

Yehor Dremluha

# Your Journey To Fluent Python



<Learn/Code/Repeat>

# Table of Contents

Introduction.....	7
Lesson 1: Introduction to Programming.....	8
1. What is programming?.....	8
2. Overview of Python .....	8
3. Python installation.....	10
4. IDE Installation (VSCode) .....	11
5. print().....	11
6. input().....	14
7. Quiz: Introduction to Programming .....	16
8. Homework.....	17
Lesson 2: Data Types .....	19
1. (int and float).....	19
2. Math operations.....	20
3. (str and bool) .....	21
4. Converting data types.....	23
5. Quiz .....	25
6. Homework.....	27
Lesson 3: Conditional Logic and Strings .....	29
1. Relational Operators.....	29
2. Logical Operators (if/elif/else) .....	29
3. Nested Logical Operators.....	32
4. Strings .....	37
5. Quiz .....	42
6. Homework.....	45
Lesson 4: Strings and Loops .....	46
1. Strings in the PC memory (encodings, ascii/unicode), ord(), chr().....	46
3. Slicing .....	49
4. Methods of Strings .....	51
5. Introduction to loops .....	55
6. Quiz .....	61
7. Homework.....	64
Lesson 5: Loops .....	66
1. Introduction to loops .....	66
2. break, continue, and else in Loops .....	68
3. Nested Loops.....	72
4. Quiz .....	75
5. Homework.....	77
Lesson 6: Data Structures p.1 (list) .....	78
1. Mutable and Immutable data types .....	78
2. Introduction to list .....	80
3. Functions (len(), sum(), min(), max(), sorted()).....	84
4. Methods of Lists.....	86
5. General methods of Iterations .....	87
6. Copying lists.....	88
7. List comprehensions .....	91
8. join() and split().....	93
9. Quiz .....	95
10. Homework.....	97

Lesson 7: Data Structures p.2 (tuple)	102
1. Overview of Tuples	102
2. Features Overview	105
3. Iterations	107
4. Quiz	108
5. Homework	110
Lesson 8: Data Structures p.3 (set)	112
1. Introduction to Sets	112
2. Methods of sets	117
3. Set Comprehensions	119
4. frozenset() Overview	119
5. Quiz	120
6. Homework	122
Lesson 9: Data Structures p.4 (dict)	124
1. Introduction to dict	124
2. Functions and Methods of dict	125
3. Iterations	127
4 Dictionary Equality	130
5. Nested Dictionaries	131
6. Quiz	133
7. Homework	134
Lesson 10: Functions	136
1. Introduction to Functions	136
2. Parameters and Arguments	138
3. Positional vs Key Arguments	139
4. Scopes	140
5. Return	142
6. Optional Parameters	143
7. Args and Kwargs	144
8. Argument Ordering	146
9. Quiz	147
10. Homework	151
Lesson 11: Exceptions	153
1 Introduction	153
2 The try & except Block	153
3 The else Block	156
4 The finally Block	157
5 Raising Exceptions	157
6 Exception Chaining	159
7. Quiz	160
8. Homework	162
Lesson 12: Imports	164
1. Introduction	164
2. Creating a Module	164
3. import, from, and as	166
4. if __name__ == "__main__"	167
5. Packages	168
6. venv	170
7. requests	172
8. Pillow	174
9. Choose your direction	176

10. Homework.....	176
Lesson 13: Files.....	178
1 What is a File?.....	178
2. Working with Files.....	178
3. Exception Handling.....	184
4 Full/Relative Paths .....	184
5. Context Manager with.....	185
6 Working with Files of Different Formats.....	186
7. Quiz .....	189
8. Homework.....	190
Lesson 14: Functional Programming .....	192
1. Revision of Functions .....	192
2. Programming Principles(KIS, DRY, YAGNI).....	193
3. Function pointers.....	197
4. Higher Order Functions.....	199
5. Closures.....	200
6. Decorators .....	202
7. Lambda Functions .....	203
8. Function Composition .....	204
9. Currying .....	206
10. Recursion.....	207
11. Function Memorization .....	207
12. Quiz.....	209
13. Homework.....	212
Lesson 15: OOP (Object oriented programming) .....	215
1. Intro to OOP .....	215
2. Class VS Instance Attributes and Methods .....	219
3. Class vs Instance methods + @staticmethod .....	221
4. Key Paradigms of OOP .....	223
5. Examples of good OOP designs.....	238
6. Quiz.....	240
7. Homework.....	244
Lesson 16: Intermediate OOP .....	246
1. Composition and Aggregation .....	246
2. Advanced Inheritance.....	250
3. Dunder (Magic) Methods in Python.....	252
4. Enums.....	263
5. Quiz .....	264
6. Homework.....	266
Lesson 17: SOLID.....	268
0. Definition .....	268
1. Single Responsibility Principle (SRP) .....	268
2. Open/Closed Principle (OCP) .....	270
3. Liskov Substitution Principle (LSP) .....	271
4. Interface Segregation Principle (ISP).....	273
5. Dependency Inversion Principle (DIP) .....	276
6. Summarise.....	278
7. Let's Refactor! .....	279
8. Quiz.....	284
9. Homework.....	287
Lesson 18: Logging.....	288

1. What is logging?	288
2. Python's Built-in Logging Module	288
3. Configuring Logging: Handlers, Formatters, and Config Files	290
4. Logging Best Practices	295
5. Homework	295
Lesson 19: Testing	296
1. Why do we need testing?	296
2. Types of Testing	296
3. Introduction to unittest	296
4. Introduction to pytest	301
5. Mocking and Patching	304
6. Advanced Techniques	306
7. Coverage Analysis	307
8. Applying Testing	308
9. Homework	310
Lesson 20: Iterators and Generators	311
1. Iterators	311
2. Building Your Own Iterators	312
3. Performance overview	314
4. Generators	315
5. Generators VS Lists	318
6. Best practices	319
7. Quiz	319
8. Homework	320
Lesson 21: Refactoring and Code Review	323
1. The Importance of Refactoring and Code Review	323
2. Principles of Good Refactoring	323
3. Code reviews	331
4. Application Development Life Cycle	336
5. Homework	340
Lesson 22. Documenting Python Code	341
1. Introduction	341
2. Code Comments and Docstrings	341
3. Annotations	343
4. External Documentation	348
5. Homework	349
Lesson 23: Regular Expressions	350
1. Regular Expressions	350
2. Basic Patterns	351
3. Character Classes	354
4. Quantifiers	358
5. Anchors and Boundaries	360
6. Grouping and Capturing	362
7. Practice	365
8. Homework	367
Lesson 24: Algorithms	371
1. What are Algorithms?	371
2. Algorithmic Complexity (Big O Notation)	371
3. Sorting Algorithms	378
4. Searching Algorithms	380
5. Dynamic Algorithms	382

Lesson 25: Advanced OOP .....	384
1. Mixins .....	384
2. Metaclasses .....	388
3. Type checking .....	391
4. Duck Typing.....	395
5. Quiz .....	399
6. Homework.....	401
Lesson 26: Context Managers .....	402
1. Introduction .....	402
2. contextlib.....	405
3. Nested context managers.....	406
4. Quiz .....	408
5. Homework.....	409
Lesson 27: Concurrent and Parallel Programming.....	410
1. Concurrent Programming .....	410
2. Parallel Programming.....	414
3 Key Differences .....	415
4. Homework.....	417
Lesson 28: Threading .....	418
1. Introduction to Threads .....	418
2. Threading .....	423
3. Global Interpreter Lock (GIL).....	432
4. Thread Communication.....	433
5. Daemon Threads .....	436
6. Thread Pooling .....	437
7. More Practice .....	439
8. Quiz .....	442
9. Homework.....	444
Lesson 29: Multiprocessing .....	446
1. Multiprocessing vs Multithreading .....	446
2. multiprocessing .....	447
3. Inter-process Communication (IPC) .....	448
4. Synchronisation.....	452
5. Process Pooling .....	455
6. Production Approaches .....	456
7. Practice.....	458
8. Quiz .....	461
9. Homework.....	462
Lesson 30: Asyncio .....	464
1. Introduction .....	464
2. Coroutines .....	464
3. Event Loop .....	465
4. Practice.....	467
5. Scheduling Tasks .....	469
6. Project and Debugging.....	470
7. Quiz .....	472
8. Homework.....	473
Quiz Answers .....	475

# Introduction

Welcome to the thrilling journey to Python fluency!

My name is Yehor, and I'm the CTO and co-founder of [Swetrix](#), a privacy-focused analytics platform. Additionally, I hold a Software Engineer position at a company based in Oxford, United Kingdom.

After years of studying programming, I've reached a solid level of proficiency, mainly through reading books and exploring different courses.

I've found that many books in the programming world lack practical relevance and don't give the knowledge to actually build real software. They're informative but don't have interactivity and engagement with a learner at all.

That's why I am excited to introduce a revolutionary approach to education. This isn't just another book to read. It's a comprehensive guide that will walk you through every step of your learning journey, providing interactive experiences and personalised support along the way.

In order to be able to interact with me, you will need to do the following:

1. Create a GitHub account. The detailed instructions can be found on the [official GitHub website](#). The repository with a book is located on the following [webpage](#) which might come in handy in case you need to copy or view the code directly from the book.
2. Join the [Discord server](#) where you can submit your H/W for my personal review, ask general questions about the topic you are struggling with, and integrate into our Python society to excel your knowledge and skills together.

I believe that it is possible to change traditional approaches to learning through books with constant support from the narrator, where a person can interact with material and have real-time practice and understanding, instead of getting frustrated when something is not going as it was planned.

This book is suitable for both beginners and experienced programmers. It covers a wide range of advanced topics as well, including detailed overviews of OOP, Testing, Documentation, Code Review with Refactoring, and Asynchronous Programming.

You don't have to be alone, please, once you get stuck ask me and other members of the community to help you!

In case you have any questions or would like to contact me directly, just do it!

Email: [yehordreliuha@gmail.com](mailto:yehordreliuha@gmail.com)

# Lesson 1: Introduction to Programming

Welcome to the exciting journey of programming! Get ready to learn, create, and have some fun along the way. Let's turn your ideas into reality!

## 1 What is programming?

Programming is the process of creating a set of instructions that tell a computer how to perform a task. It's like writing a recipe, but instead of making a cake, you're creating different apps.

### 1.1 What can you do with programming?

Talking of all opportunities which are opened to you after learning how programming works , you will be able to do the following things:

1. **Solve Problems:** Programming can be used to model and solve complex problems in fields like finance, engineering, and medicine.
2. **Develop Software and Applications:** From the operating system on your computer to the apps on your phone, programming is the backbone of software development.
3. **Create Websites:** Web development involves programming both the visual front-end that users interact with and the back-end that processes data.
4. **Automate Tasks:** You can write scripts to automate repetitive tasks, saving time and reducing errors. This could be anything from sorting files to generating reports.
5. **Analyze Data:** With programming, you can process large sets of data to find patterns and insights, a practice commonly known as data science.
6. **Build Games:** Game development is another avenue, where programming brings to life the mechanics, graphics, and interactivity of a game.

## 2. Overview of Python

Choosing Python as your first language opens a world where coding is not just accessible, but also very enjoyable and rewarding.

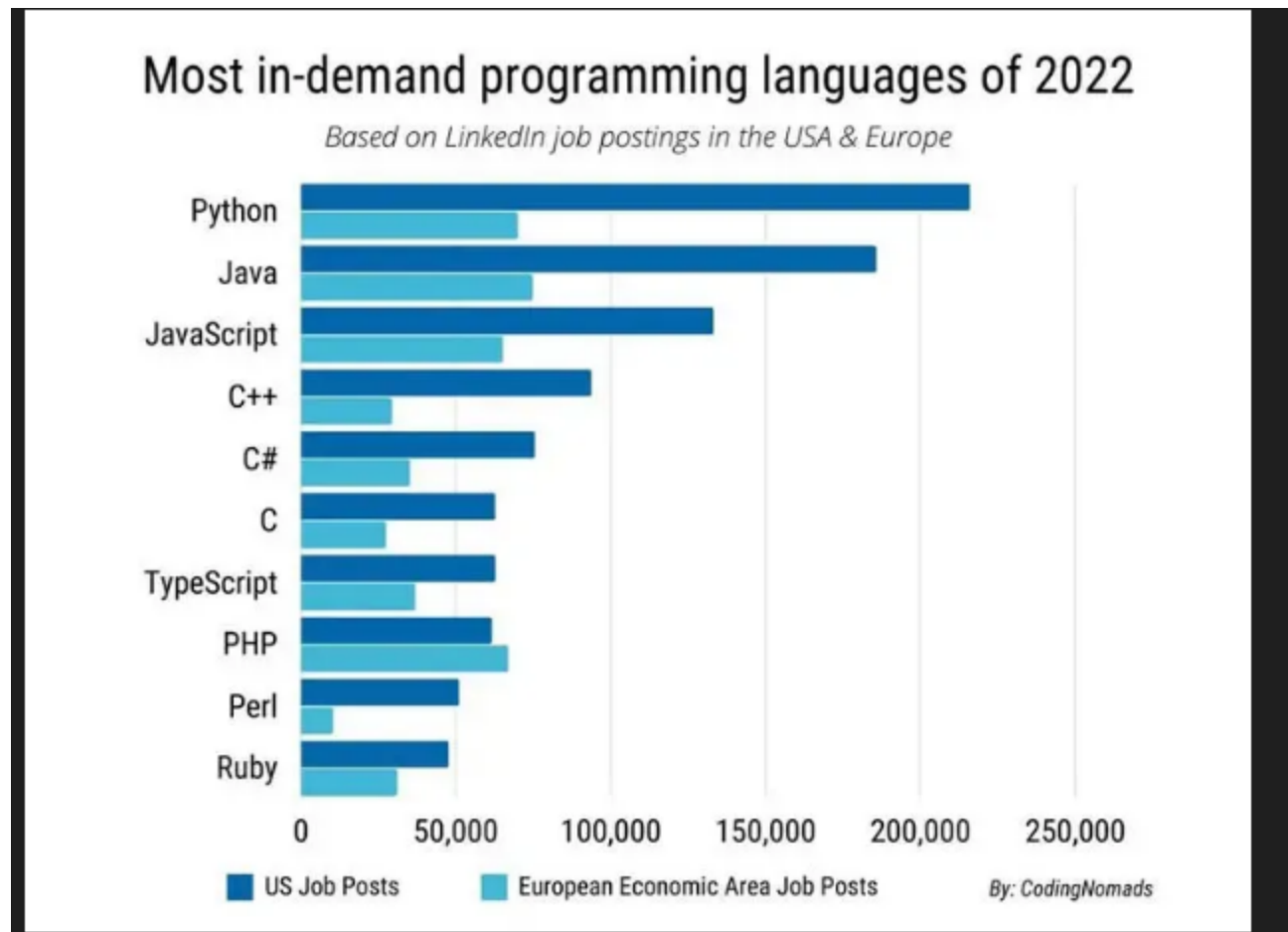
Aspect	Details
Features	<ul style="list-style-type: none"><li>- Easy to Learn</li><li>- Interpreted</li><li>- Versatile</li><li>- Extensive Libraries</li><li>- Cross-Platform</li><li>- Open Source</li><li>- Community Support</li></ul>
Common Uses	<ul style="list-style-type: none"><li>- Web Development (Django, Flask)</li><li>- Data Science (NumPy, SciPy, Pandas)</li><li>- Automation (Scripting)</li><li>- Scientific Computing</li><li>- Education (Teaching Programming)</li></ul>



Aspect	Details
Advantages	<ul style="list-style-type: none"> <li>- Readable syntax encourages good practices</li> <li>- Low maintenance cost</li> <li>- Free for all major platforms</li> </ul>
Disadvantages	<ul style="list-style-type: none"> <li>- Slower execution compared to compiled languages</li> <li>- Less prevalent in mobile computing compared to web development</li> </ul>

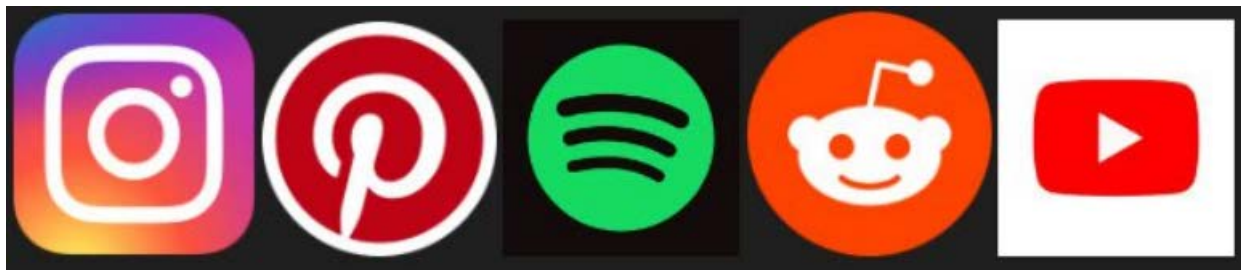
## 2.1 Sample Programs

Over its almost 30 years of existence, Python has become one of the most popular programming languages.



To give you the big picture, let's take a look at some applications and companies who use this language on everyday basis

Each of these application are using python in a certain way, and as you can see from the statistics it becomes much popular every day



### 3. Python installation

#### For Windows Users

##### Step 1: Download Python

- Visit the official Python website at [python.org](https://python.org).
- Navigate to the Downloads section and download the latest version of Python for Windows.

##### Step 2: Download Python

- Once the installer is downloaded, run it.
- Make sure to check the box that says "Add Python to PATH" before installation.
- Click "Install Now" to start the installation process.

##### Step 3: Verify the Installation

- Open Command Prompt and type `python --version` and press Enter.
- If Python is installed correctly, you should see the version number.

```
C:\Users\PC>python --version
Python 3.12.0
```

#### For macOS Users

##### Step 1: Download Python

- Visit the official Python website at [python.org](https://python.org).
- Under the Downloads section, choose the macOS version and download it.

##### Step 2: Run the installer

- Click "Install Now" to start the installation process.

##### Step 3: Verify the Installation

- Open Terminal and type `python --version` and press Enter.
- The version number should appear if the installation was successful.

```
IDLE Shell 3.10.1
Python 3.10.1 (v3.10.1:2cd268a3a9, Dec 6 2021, 14:28:59) [Clang 13.0.0 (clang-1300.0.29.3)] on darwin
Type "help", "copyright", "credits" or "license()" for more information.
>>>
```

## 4. IDE Installation (VSCode)

To start coding, we need two main things: a development environment where we can write our code, and the Python to execute it.

### Step 1: Downloading vsCode

- Select the version that matches your operating system—Windows, Linux, or macOS are all supported.
- Click the download button and install IDE

### Step 2: Installing Python in VSCode

- Look for the Extensions icon on the left-hand sidebar—it looks like four squares with one square detached.
- Search for 'Python' and select the one published by Microsoft.
- Click 'Install' and give it a moment to set up.

### Step 3: Writing Your First Python Code

- With the Python extension installed, you can now write Python code.
- Open a new file, save it with the .py extension, and start typing your Python program.
- You can run your code directly in VSCode. Just hit the green play button on the top right or press F5.

That's it! You've set up your IDE, and you're ready to dive into Python programming.

### Output

```
python --version
Python 3.8.7
```

## 5. print()

The `print()` function is used in Python in order to send data to the console. This is the primary way to output data from your program to the user.

In order to see the printed statement, you have to run the code, otherwise it won't be executed and there will be nothing inside the console.

### Example

```
print("We are learning Python!")
```

Create a file `main.py`, save it in VScode (using shortcut `Ctrl + s`) or for Mac users (`Cmd + s`) and run the following command

```
python main.py
```

## Output

```
We are learning Python!
```

**NOTE:** You can use the single quotes and the output will remain the same.

```
print('We are learning Python!') == print("We are learning Python!")
```

The `print()` command allows you to specify **multiple arguments**, in which case they must be separated by commas. If you leave out commas between arguments, Python will treat it as a syntax error.

## Example

```
print("I", "will" "become", "a" "software", "engineer")
```

## Output

```
I will become a software engineer!
```

Each subsequent `print()` will create output the message on the next line

## Example

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

## Output

```
Hello  
  
World  
  
!
```

## 5.1 `print()` with and without arguments

The `print()` function with an empty argument list simply inserts a new empty line.

## Example

```
print("What a lovely day!")
print()
print("Naah, I am too lazy to work, let's go to pub!")
```

## Output

```
What a lovely day!
Naah, I am too lazy to work, let's go to pub!
```

## 5.2 `sep` and `end` arguments

The `print()` function in Python has optional arguments `sep` and `end` which offer *more control* over the formatting of the output.

Argument	Purpose
<code>sep</code>	Specifies the separator <i>between</i> the values. By default, it is a space.
<code>end</code>	Specifies what to print at the end. By default, it is a newline character ( <code>\n</code> ).

## Example

```
# Case #1 -> dash symbol is added between all string we pass into print()
print("There", "are", "5", "apples", sep="-")
# Case #2 -> instead of printing ``world!`` on the new (separate) we print it
within one line
print("Hello", end=" ")
print("world!")
# Case #3 separate all worlds with a coma and add `and more` to the sentence
print("Python", "Java", "C++", sep=", ", end=" ")
print("and more!")
```

## Output

```
# Case #1
There-are-5-apples
# Case #2
Hello world!
# Case #3
```

## 5.3 Escaping characters

In Python, certain characters can be 'escaped' to achieve special formatting in strings.

Escape Sequence	Meaning
\n	Newline - moves to the next line
\t	Horizontal Tab - adds a tab space
\\	Backslash - to use a backslash itself
\'	Single Quote
\"	Double Quote

### Example

```
print("This is a line.\nAnd this is another line.")
print("Name:\tJohn")
print("Path to the folder: C:\\Users\\John")
```

### Output

```
This is a line.
And this is another line.
Name:   John
Path to the folder: C:\Users\John
```

## 6. input()

The main goal as a software engineer is to make an interactive application which is user friendly and matches the highest standards in the world of programming.

The `input()` function in Python allows your program to collect data entered by the user. This function waits for the user to type something into the console and then press Enter [↵].

It treats the incoming data as a string (`str`) and can be stored in a *variable* for further use.

In Python, declaring variables is pretty straightforward, you just need to assign a value to a variable with the equals sign (`=`)

### Example

```
print("Hello, what is your name?")

# At this stage the program is awaiting the input from user which we store in
the variable called ``name`` and we can use it later
name = input()
print('Hello,', name, "it is nice to see you!")
```

## Explanation

In the example above we can see that the program is executed line by line, because Python is an interpreted language and it is very important to understand, that before executing the line `Hello <name>`, it is nice to see you! it freezes, until the user is prompted to input their name

## Output

```
Hello, what is your name?
>> (Adam)
Hello Adam, it is nice to see you!
```

## 6.2 Rules how to name variables

- Use only letters (a-z, A-Z), digits, and underscores (\_).
- A variable name **cannot start with a digit**.
- The name **must ideally reflect its purpose**.
- Remember, **Python is case-sensitive: name and Name are different variables**.
- The convention is to use `lower_case_with_underscores`.

## 6.3 Assignments

### Assignment 1: Personalized Greeting

**Objective:** Write a program that asks for the user's name and favorite color, then prints a personalized greeting that includes their name and favorite color.

```
Input:
- User's name
- User's favorite color
Output:
- A personalized greeting that includes the name and favorite color.
```

### Assignment 2: Order Summary

**Objective:** Create a program that asks for the user's address, the item they are purchasing, and the cost. Then, print a summary of their order.

Input:

- User's address
- Item name
- Item cost

Output:

- A summary of the user's order, including their address, the item, and the cost.

### Assignment 3: Daily Reflection

**Objective:** Prompt the user to input what they learned today.

Input:

- Something you learned today.

Output:

- A reflection statement.

## 7. Quiz: Introduction to Programming

### Question 1:

What is the main purpose of programming?

- A) To create documents and presentations.
  - B) To instruct the computer to perform tasks.
  - C) To browse the internet.
  - D) To play computer games.
- 

### Question 2:

Which statement about Python is true?

- A) Python is only good for web development.
  - B) Python is a high-level programming language that is easy to learn.
  - C) Python cannot be used for data analysis.
  - D) Python programs can only run on Windows operating systems.
- 

### Question 3:



What is an IDE in the context of programming?

- A) A set of rules that govern the syntax of a programming language.
  - B) A program that provides facilities like code editing and debugging to programmers.
  - C) A type of computer specially made for coding.
  - D) An internet service that speeds up your code.
- 

#### **Question 4:**

Which of the following is a common use of Python?

- A) Cooking recipes automation.
  - B) Changing the weather.
  - C) Web development and data science.
  - D) Time travel.
- 

#### **Question 5:**

After writing a Python script, what is the file extension you should save it with?

- A) .txt
  - B) .xls
  - C) .py
  - D) .ppt
- 

## **8. Homework**

### **Task 1: Investigate Python's History**

- Research the history of Python. When was it created and by whom? What is PEP?

### **Task 2: Real-World Python Applications**

- Find three examples of how Python is used in different industries. Look for specific use cases such as web development, data analysis or AI

### **Task 3: Discover IDE Features**

- Try some hotkeys in Visual Studio Code! Explore your IDE!

### **Task 4: My favourite football team**

**Objective:** Write a Python program that asks for the name of a soccer team and then prints a cheer for that team.

**Requirements:** The program should take the name of the soccer team as input. It should then output the name of the team followed by the cheer "are the champions!". Example:

```
# Sample Input
Barcelona
# Sample Output
Barcelona is a champion!
```

### Task 5: Repeat after me

**Objective:** Create a Python program that captures three lines of text, one at a time, and then prints them out in reverse order.

```
# Sample Input
I love
Python
so much
# Sample Output
I love
Python
so much
```

### Task 6: Reverse Echo

**Objective:** Write a Python program that takes in three separate lines of text and then prints them out in reverse order.

```
# Sample Input
I love
Python
so much
# Sample Output
so much
Python
I love
```

# Lesson 2: Data Types

In the world of programming, data types are the building blocks that shape the digital landscape."

## 1. (int and float)

Variables in Python are like containers for storing data values. Python has various data types including integers, float (decimal numbers), strings, and boolean values.

### 1.1 int

Integers are the data type used to represent **whole numbers**. Unlike strings, integers *are not* surrounded by quotes.

In order to check what is the type of the variable we can use function `type()` and print its output.

#### Example

```
# Int
number_of_apples = 5
my_age = 22
print("There are", number_of_apples, "apples in the basket")
print(type(number_of_apples))
print("My age is:" my_age)
print(type(my_age))
```

#### Output

```
The are 5 apples in the basket
<class 'int'>
My age is: 22
<class 'int'>
```

As python is a dynamical language (it means that the programmer doesn't declare types *explicitly*) this can cause some issues in the future with advanced data types. So if you are not sure what type is stored in the variable don't hesitate to call `type()`.

### 1.2 float

Floats are numbers that have a *decimal point*. They can show fractions as well as whole numbers, like the change you get back from a dollar or how much of a cake is left.

Sometimes, you'll see really big or small floats written with an "e" which is just a shortcut way of writing zeros.

#### Example

```
# Float
apple_price = 0.99
print("Apple price:", apple_price, "pounds")
print("Type of apple_price variable:", type(apple_price))
```

## Output

```
Apple price: 0.99 pounds
Type of apple_price variable: <class 'float'>
```

## 2. Math operations

In Python, you can perform *all* the basic mathematical operations you're familiar with from arithmetic.

This includes addition, subtraction, multiplication, division, and even more complex operations like exponentiation.

Here are the the table where each operator described within Python syntax"

Operator	Description
+	addition
-	subtraction
*	multiplication
/	division (always returns a float)
//	Floor division (divides and rounds down to nearest integer)
**	Exponentiation (raises a number to the power of another)
%	Modulo (gives the remainder of a division)

## Example

```
# Two numbers
num1 = 15
num2 = 4
# Operations
sum_result = num1 + num2
difference_result = num1 - num2
division_result = num1 / num2
integer_division_result = num1 // num2
exponentiation_result = num1 ** num2
modulo_result = num1 % num2
# Displaying the results
```

```

print("Addition of", num1, "and", num2, "=", sum_result)
print("Subtraction of", num1, "from", num2, "=", difference_result)
print("Division of", num1, "by", num2, "=", division_result)
print("Integer division of", num1, "by", num2, "=", integer_division_result)
print("Exponentiation of", num1, "to the power of", num2, "=",
exponentiation_result)
print("Modulo of", num1, "by", num2, "=", modulo_result)

```

## Output

```

Addition of 15 and 4 = 19
Subtraction of 15 from 4 = 11
Division of 15 by 4 = 3.75
Integer division of 15 by 4 = 3
Exponentiation of 15 to the power of 4 = 50625
Modulo of 15 by 4 = 3

```

Here is a practical task which we remember from school, where we need to calculate the circumference of the circle

## Example

```

# Constants
PI = 3.14159
# Radius of a circle
radius = 5
# Calculating area (PI * r^2)
area = PI * (radius ** 2)
# Calculating circumference (2 * PI * r)
circumference = 2 * PI * radius
print("Given a circle with a radius of:", radius)
print("Area of the circle:", area)
print("Circumference of the circle:", circumference)

```

## Output

```

Given a circle with a radius of: 5
Area of the circle: 78.53975
Circumference of the circle: 31.4159

```

## 3. (str and bool)

### 3.1 String Data Type (str)

A string in Python is a *series of characters*. It is used to represent text.

As you know already, strings in Python are enclosed either in single quotes (') or double quotes ("), and they can include *letters, numbers, and various symbols*. Also when we ask a user for input, we store the `str` type into the variable.

### Example

```
# String Examples
greeting = "Hello, Python learners!"
course_name = 'The best Python Course in the United Kingdom, or even in the whole world!'
student_name = input("Input your name:")
print(greeting)
print("The course you are taking is called:", course_name)
print("Name of the student is", student_name)
```

### Output

```
Hello, Python learners!
The course you are taking is called: The best Python Course in the United Kingdom, or even in the world!
>> Adam
Name of the student is Adam
```

## 3.2 Boolean Data Type (`bool`)

Boolean/`bool`, is a data type that can only have two values: `True` or `False`.

Booleans are often the result of comparisons or conditions in Python and are **extremely important** *for decision-making in code*.

### Example

```
is_student = True
is_weekend = False
print("Is student:", is_student)
print("Is it the weekend:", is_weekend)
```

### Output

```
Is student: True
Is it the weekend: False
```

## 4. Converting data types

In all programming languages, sometimes there is a need to convert data from one type to another. This process is known as type *conversion* or *typecasting*.

It's especially important when the data type of a value **is not suitable for a specific operation**.

### 4.1 Implicit Conversion

In this example, Python automatically converts data types after arithmetical operation.

#### Example

```
num_int = 6          # Integer type
num_float = 1.5      # Float type
# Python automatically converts num_int to a float
total = num_int + num_float
print("Total:", total)
print("Type of total:", type(total))
```

#### Output

```
Total: 7.5
Type of total: <class 'float'>
```

### 4.2 Explicit Conversion

Explicit conversion requires the programmer to convert data types manually. Python provides functions like `int()`, `float()`, `str()`, etc.. **for explicit conversions**.

There are several examples provided below which explain the need of such conversions.

If we try to make math operations between strings, it will result in an error:

#### Incorrect Usage, **Avoid doing this!**

```
# Both variables are of type 'str' (string)
salary = "1000"
tax = "200"
# Attempting to subtract/multiply/divide two strings - will cause an error!!
net_income = salary - tax
print("Net income: ", net_income)
```

#### Output

```
ERROR!
Traceback (most recent call last):
  File "<string>", line 5, in <module>
TypeError: unsupported operand type(s) for -: 'str' and 'str'
```

**Note:** In such cases we would want to convert the types of variables

**Correct Usage. This is a recommended approach**

```
# These variables are ``strings`` representing ``numbers``
salary_str = "1000"
tax_str = "200"
# Converting ``str`` variables to ``int``
salary_int = int(salary_str)
tax_int = int(tax_str)
# Now we can perform the subtraction with ``int`` variables
net_income = salary - tax
print("Net income: ", net_income)
```

## Output

```
Net income: 800
```

We can convert user's input into the integers while using `input()` function.

## Example

```
# Asking the user to input their age and converting it into the integer
age= int(input("Please enter your age: "))
print("Type of age is: ", type(age))
print("You are", age, "years old.")
```

## Output

```
Please enter your age:
>> 21
Type of age is:  <class 'int'>
You are 21 years old.
```

## 4.3 Assignments



## Assignment 1: Cube Measurements

**Objective:** Write a program that calculates the volume and total surface area of a cube from the length of its edge.

Input:

One integer: the length of the cube's edge.

Output:

The program should output the volume and the total surface area of the cube.

## Assignment 2: Computer Set Cost Calculator

**Objective:** Create a program that calculates the total cost of three computer sets. Each set consists of a monitor, system unit, keyboard, and mouse.

Input:

Four integers on separate lines: the cost of the monitor, system unit, keyboard, and mouse, respectively.

Output:

The total cost of three computer sets.

## Assignment 3: Basic Arithmetic Operations

**Objective:** Write a program that computes the sum, difference, and product of two integers entered by the user.

Input:

Two integers on separate lines.

Output:

The sum, difference, and product of the two integers, each on a separate line.

## 5. Quiz

### Question 1:

What data type would you use to represent a person's name in Python?

- A) int
- B) str
- C) bool
- D) float

---

### Question 2:

Which of the following is the correct way to convert the string '123' to an integer?

- A) `int("123")`
  - B) `"123".int()`
  - C) `str(123)`
  - D) `integer("123")`
- 

### Question 3:

What is the result of this operation: `10 // 3`?

- A) `3.33`
  - B) `3`
  - C) `3.0`
  - D) `4`
- 

### Question 4:

Which function would you use to read a user's input as a string in Python?

- A) `input()`
  - B) `print()`
  - C) `read()`
  - D) `getString()`
- 

### Question 5:

How would you print the type of a variable `x` in Python?

- A) `print(x)`
  - B) `print(type(x))`
  - C) `type(print(x))`
  - D) `print(x.type)`
- 

### Question 6:

What will be the output of the following code?

```
x = "10"  
y = 5
```

```
print(x + y)
```

---

### Question 7:

What does the % operator do in Python?

- A) Multiplies two numbers
  - B) Divides two numbers and returns the integer part
  - C) Adds two numbers and returns their modulo
  - D) Divides two numbers and returns the remainder
- 

### Question 8:

What will be the output of the following Python code?

```
temperature_str = "25.5"  
temperature = float(temperature_str) + 10  
print(temperature)
```

- A) 35.5
  - B) 25.510
  - C) "35.5"
  - D) TypeError
- 

## 6. Homework

### Task 1: Unit Converter

**Objective:** Develop a program to convert inches to centimeters.

```
Input: The user inputs a length in inches.  
Output: The program outputs the equivalent length in centimeters.  
Conversion: 1 inch = 2.54 cm.  
Input: 10  
Output: 25.4 cm
```

### Task 2: Temperature Converter

**Objective:** Write a program that converts a temperature from Fahrenheit to Celsius.

Input: The user enters a temperature in Fahrenheit.

Output: The program prints the temperature in Celsius.

Formula:  $\text{Celsius} = (\text{Fahrenheit} - 32) * 5/9$ .

Input: 32

Output: 0.0 Celsius

### Task 3: Simple Interest Calculator

**Objective:** Create a script to calculate simple interest.

Input: The user enters the principal amount, the rate of interest (as a percentage), and the time period in years.

Output: The script outputs the calculated simple interest.

Formula:  $\text{Simple Interest} = (\text{Principal} * \text{Rate} * \text{Time}) / 100$ .

### Example

Input: Principal = 1000, Rate = 5, Time = 3

Output: Simple Interest is 150.0

# Lesson 3: Conditional Logic and Strings

"In the world of code, a single `if` statement can alter the flow just as a single choice can change a life."

## 1. Relational Operators

Relational operators in `Python` are used to compare the values on either side of them and determine the relation between them.

**Table of Relational Operators**

Operator	Description	Example	Result
<code>&gt;</code>	Greater than	<code>5 &gt; 3</code>	True
<code>&lt;</code>	Less than	<code>5 &lt; 3</code>	False
<code>==</code>	Equal to	<code>5 == 5</code>	True
<code>!=</code>	Not equal to	<code>5 != 3</code>	True
<code>&gt;=</code>	Greater than or equal to	<code>5 &gt;= 5</code>	True
<code>&lt;=</code>	Less than or equal to	<code>5 &lt;= 3</code>	False

The order of calculation is the same as in math

## 2. Logical Operators (`if/elif/else`)

Logical operators in `Python` are used to **control the flow of your program based on certain conditions**. These operators allow you to create conditional statements to execute *specific* blocks of code.

### 2.1 Intro (Indentation)

In some programming languages, *indentation is a matter of personal preference*, and code can be written without it.

However, in `Python`, indentation is an integral part of the code structure. Incorrect indentation can lead to syntax errors and incorrect program behavior.

#### Example

The correct indentation should be 4 spaces according to the PEP-8 standart.

```
if condition:
    print("This code is part of the 'if' block.")
```

Incorrect indentation - will result in an `IndentationError`.

### Example

```
if condition:
print("This code is not indented properly.")
# OR
if condition:
    print("This code is not indented properly.")
```

### Output

```
# IndentationError: expected an indented block
```

Follow the rules of indentations during this course, and you will get the general idea of constructions which require them.

Don't worry if this seems to be hard for now, with more practice indentation will never be a problem even in complex applications.

## 2.1. `if` Statement

The `if` statement is used to execute a block of code only if a condition is `True`. It is very common to use relational operators in conditional statements to control the flow of execution.

### Example

```
x = 10
y = 5
if x > 5: # it is ``True`` --> the code within ``if`` block WILL BE executed
    print("x is greater than 5")
if y < 5: # is not ``True`` --> the code within ``if`` block WON'T BE executed
    print("y is less than 5")
```

### Output

```
x is greater than 5
```

Let's write a mini program to verify, that you actually learning Python and allowed to attend this course.

## Example

```
answer = input('Which programming language are we learning?')
if answer == 'Python':
    print('Correct', 'We are learning Python', 'It is an excellent language',
sep='\n', end='!\n')
```

## Output

```
Which programming language are we learning?
>> Python
Correct
We are learning Python
It is an excellent language!
```

## 2.2 elif and else

The `elif` statement is used to execute a block of code, **only if the previous conditions** in an `if` statement **are not met**, and **a new condition is True**

The `else` statement is used in conjunction *with an if statement* to specify a block of code that should be executed **when the conditions in the if and elif statements are not True**

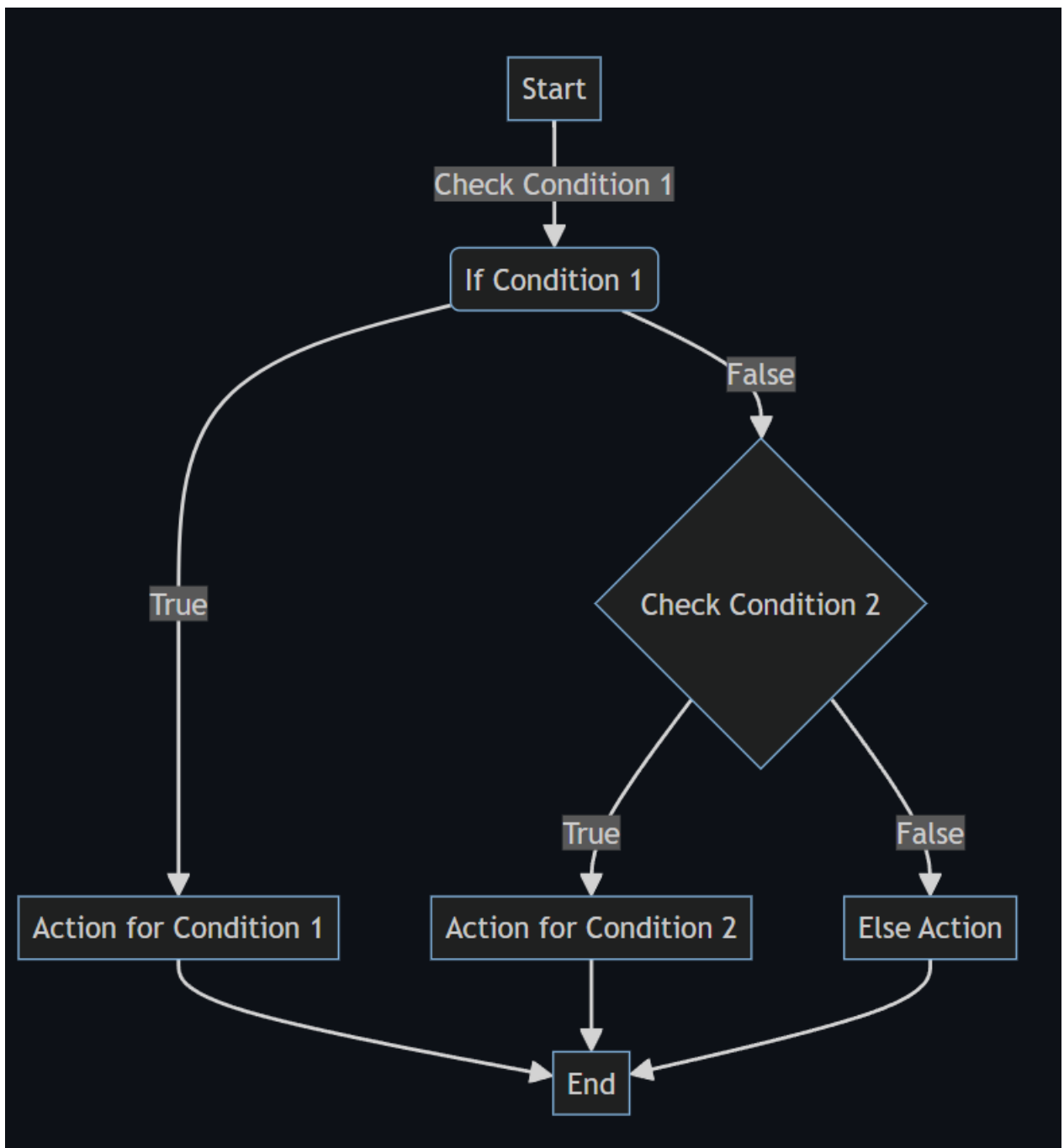
## Example

```
answer = input('Which programming language are we learning?')
if answer == 'Python':
    print('Correct!')
elif answer == 'Java':
    print('Ha-ha, Nice try, but we are learning Python!')
else:
    print('Sorry, that\'s not the language we are learning ?? -> ', answer)
```

## Output

```
Which programming language are we learning? # Asking for user's input
# Case 1
>> Python
Correct
# Case 2
>> Java
Ha-ha, Nice try, but we are learning Python!
# Case 3
>> Cooking
Sorry, that's not the language we are learning ?? ->  Cooking
```

You can see the diagram down below which explains how conditionals work



### 3. Nested Logical Operators

What should we do when we have multiple conditions? In Python, there are *three* logical operators that allow us to create complex conditions:

- `and` — logical multiplication;
- `or` — logical addition;
- `not` — logical negation.



## 3.1 Table of Logical Operators

Logical operators are used to combine conditional statements.

Operator	Description	Example	Result
and	True if both the operands are true	5 < 10 and 3 > 1	True
or	True if at least one of the operands is true	5 < 3 or 3 > 1	True
not	True if operand is false	not(5 < 3)	True

## 3.2 Operator and

Suppose we need to write a program for students who are: **at least twelve years old, are studying in at least the 7th grade, and live in the UK**

Access to it should be restricted for those who are younger and not local.

### Example

We combined three conditions using the and operator. It means that in this branching, the code block is executed only **if all conditions are met simultaneously!**

```
age = int(input('How old are you?: '))
grade = int(input('What grade are you in?: '))
country = input('Where are you from?')
if age >= 12 and grade >= 7 and country == 'United Kingdom':
    print('Access granted.')
else:
    print('Access denied.')
```

### Output

```
# Case # 1 --> All conditions are correct
How old are you?: 13
What grade are you in?: 7
Where are you from? United Kingdom
Access granted.
# Case # 2 --> One condition is incorrect
How old are you?: 12           # Correct
What grade are you in?: 6      # Incorrect
Where are you from? United Kingdom # Correct
Access denied.
```

## The truth table for and operator

a	b	a and b
False	False	False
False	True	False
True	False	False
True	True	True

### 3.3 Operator or

Let's create a program that grants access to a library **if the user is either a teacher, a student over 18, or has a special access card.**

#### Example

```
role = input('Are you a student or a teacher?: ')
age = int(input('How old are you?: '))
has_access_card = input('Do you have an access card? (yes/no): ')
if role == 'teacher' or age > 18 or has_access_card == 'yes':
    print('Welcome to the library!')
else:
    print('Sorry, access is restricted.')
```

#### Output

```
# Case # 1 --> User is a teacher
Are you a student or a teacher?: teacher
How old are you?: 30
Do you have an access card? (yes/no): no
Welcome to the library!
# Case # 2 --> User is a student under 18 without an access card
Are you a student or a teacher?: student
How old are you?: 17
Do you have an access card? (yes/no): no
Sorry, access is restricted.
```

## The truth table for or operator

a	b	a or b
False	False	False
False	True	True
True	False	True
True	True	True

### 3.4 Operator not

Suppose we want to restrict access to a certain feature in a program to users who **are not administrators**.

#### Example

```
user_role = input('What is your role? (user/admin): ')
if not (user_role == 'admin'):
    print('Feature restricted. Only administrators have access.')
else:
    print('Feature accessible.')
```

#### Explanation

```
# Case # 1 --> User is not an admin
What is your role? (user/admin): user
Feature restricted. Only administrators have access.
# Case # 2 --> User is an admin
What is your role? (user/admin): admin
Feature accessible.
```

## The truth table for not operator

a	not a
True	False
False	True

## 3.5 Resolution of operators

Understanding the order of *resolution*, or *precedence*, of logical operators in Python is crucial. Python follows a specific order when **evaluating logical expressions**, which **can significantly impact the outcome of these expressions**.

- not — highest precedence
- and — after not and before or
- or — lowest precedence

### Example

```
result = not 5 > 3 and 3 < 2 or 5 != 4
# Breakdown:
# 1. not 5 > 3 evaluates to False
# 2. 3 < 2 evaluates to False
# 3. 5 != 4 evaluates to True
# Final result: False and False or True, which evaluates to True
print(result) # Output: True
```

You can experiment creating the different logical statements and see the outcome after evaluation.

## 3.6 Assignments

### Assignment 1: Interval Membership

**Objective:** Write a program that takes an integer  $x$  and determines whether  $x$  falls within specified intervals.

```
Input:
One integer: the value of `x`.
Output:
The program should output text in accordance with the task's condition.
Example: If the intervals are -3 to 7 (inclusive), and the user enters 5, the
output should be 'The number 5 belongs to the interval [-3, 7]'.
```

### Assignment 2: Interval Membership p.2

**Objective:** Create a program that accepts an integer  $x$  and determines whether  $x$  belongs to any of the given intervals.

```
Input:
One integer: the value of `x`.
Output:
The program should output a text message indicating all intervals the number
belongs to.
```

Example: If the intervals are `[-30, -2]` and `[7, 25]`, and the user enters 8, the output should be `'The number 8 belongs to the interval [7, 25]'`.

### Assignment 3: Weight Category Determination

**Objective:** Write a program that categorizes the weight of a boxer into one of three weight categories.

Input:  
One integer: the boxer's weight in kilograms.  
Output:  
The program should output the name of the weight category.  
Example: If the categories are 'Lightweight' up to 60 kg, 'First Middleweight' up to 64 kg, and 'Middleweight' up to 69 kg, and the user enters 65, the output should be 'Middleweight'.

### Assignment 4: Triangle Inequality Theorem

Input:  
Three positive integers, representing the lengths of the sides of a triangle.  
Output:  
The program should output 'YES' if a triangle can exist with those sides according to the triangle inequality theorem, or 'NO' otherwise.  
Example: If the user enters 3, 4, and 5, the program should output 'YES'.

## 4. Strings

Previously we saw `strings` and know how to declare them, but how to work with them?

### 4.1 Basic `str` operations

Strings can be concatenated (glued) with the `+` operator, and repeated with `*`:

#### Example

```
# Concatenation
full_greeting = greeting + ', ' + name + '!'
# Repetition
laugh = 'Ha' * 3
print(full_greeting)
print(laugh)
```

#### Output

```
Hello, World!
```

It's very useful and can improve the readability of your code while working in team!

**NOTE:** Strings are immutable data types, which means that we can't change them!

We have to create a new variable, but trying to change strings can result into the following error:

### Example

```
original_string = "Hello, World"
original_string[7] = 'w' # Attempting to change 'W' to 'w'
```

### Output

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

## 4.2 len() and operator in

The len() function is used to find out how many characters are in a string.

### Example

```
greeting = "Hello, World!"
str_len = len(greeting)
print(str_len)
print(type(str_len))
```

### Output

```
13
<class 'int'>
```

The in operator checks if a certain substring exists within another string.

### Example

```
phrase = "The quick brown fox"
print('quick' in phrase)
# You can use the ``not`` operator combining with ``in``
print('quick' not in phrase)
```

## Output

```
True
False
```

### 4.3 indexing and slicing

Often, you need to access a specific character in a string. In Python, this is done using square brackets `[]` with the `index (number)` of the desired character.

**NOTE:** The count in Python starts from 0 and the 1st element of any data structure we will learn will be 0 element!

Let's say we have the following string:

#### Example

```
s = 'Python'
```

The image and table below shows how indexing works:

Expression	Result	Explanation
<code>s[0]</code>	P	1st character of string
<code>s[2]</code>	t	3rd character of string
<code>s[5]</code>	n	6th character of string

In Python, strings support *negative indexing*.

Negative indices start from -1 for the **last character of the string**. This can be particularly useful when you want to access the elements of a string from the end without calculating its length.

Expression	Result	Explanation
<code>s[-1]</code>	n	6th character of string
<code>s[-4]</code>	t	3rd character of string
<code>s[-3]</code>	h	6th character of string

## Output

```
print(s[0])           # The first character --> P
```

```
print(s[-1])           # The last character --> n
```

## 4.4 Assignments

### Assignment 1: Interval Membership

**Objective:** Write a program that determines the shortest and longest names among three given city names.

Input:  
Three lines, each containing the name of a city.  
Output:  
The program should print the shortest and longest city names on separate lines.  
Note: It is guaranteed that the lengths of all three city names will be different.

### Assignment 2: Resting Query

**Objective:** Create a program that reads a single line of text and then decides if the text suggests resting.

Input:  
A single line of text.  
Output:  
The program should output "YES" if the line contains the word "Saturday" or "Sunday", and "NO" otherwise.

### Assignment 3: Validating an Email Address

**Objective:** Write a program that checks if an email address is correct, assuming correctness requires the presence of the '@' symbol and a .

Input:  
A single line containing an email address.  
Output:  
The program should print "YES" if the email address is considered correct, and "NO" otherwise.  
Note: The presence of symbols '@' and '.' is necessary for an email address to be correct, but their absence does not guarantee the address is incorrect.





## 5. Quiz

### Question 1:

What data type would you use to represent a person's name in Python?

- A) `int`
  - B) `str`
  - C) `bool`
  - D) `float`
- 

### Question 2:

Which of the following is the correct way to convert the string '123' to an integer?

- A) `int("123")`
  - B) `"123".int()`
  - C) `str(123)`
  - D) `integer("123")`
- 

### Question 3:

What is the result of this operation: `10 // 3`?

- A) `3.33`
  - B) `3`
  - C) `3.0`
  - D) `4`
- 

### Question 4:

Which function would you use to read a user's input as a string in Python?

- A) `input()`
  - B) `print()`
  - C) `read()`
  - D) `getString()`
-

### Question 5:

How would you print the type of a variable `x` in Python?

- A) `print(x)`
  - B) `print(type(x))`
  - C) `type(print(x))`
  - D) `print(x.type)`
- 

### Question 6:

What will be the output of the following code?

```
x = "10"  
y = 5  
print(x + y)
```

---

### Question 7:

What does the `%` operator do in Python?

- A) Multiplies two numbers
  - B) Divides two numbers and returns the integer part
  - C) Adds two numbers and returns their modulo
  - D) Divides two numbers and returns the remainder
- 

### Question 8:

What will be the output of the following Python code?

```
temperature_str = "25.5"  
temperature = float(temperature_str) + 10  
print(temperature)
```

- A) 35.5
- B) 25.510
- C) "35.5"
- D) `TypeError`



## 6. Homework

### Task 1: Secure Password Generator

**Objective:** Create an app which checks user's password and based on your rules states if it's secure or not.

```
Input: The user inputs a password.
Output: The program outputs whether the password is 'Strong', 'Medium', or 'Weak'.
Input:
Pass123!
Output:
Strong password
```

### Task 2: Custom Text-based Game

**Objective:** Develop an **interactive** game with a mini plot twist.

```
Input: The user makes choices at story junctures.
Output: The program narrates the consequence of the choices, leading to a unique story ending.
Choose your path (forest/mountain):
>> forest
Output:
You walk into the forest and find a hidden treasure chest!
```

### Task 3: Travel Itinerary Planner

**Objective:** Plan your future holiday, you might be tired already :)

```
Input: The user enters three cities they plan to visit.
Output: The program outputs a travel itinerary in the order entered.
Input:
>> London
>> Paris
>> Rome
Output:
Your travel itinerary: London -> Paris -> Rome
```

# Lesson 4: Strings and Loops

"Strings hold characters, loops repeat actions. That's coding in a nutshell."

## 1. Strings in the PC memory (encodings, ascii/unicode), ord(), chr()

We have already worked with strings a lot, but really, how does it work under the hood?

### 1.1 How it works

Strings in a computer's memory are stored as sequences of bytes. To represent strings of text, a computer has a system that maps these byte sequences to characters. This system is called a character encoding.

Two common character encoding systems are ASCII and Unicode:

### 1.2 ASCII (American Standard Code for Information Interchange)

It uses 7 bits to represent 128 unique characters, which includes letters, numbers, and control characters.

**NOTE:** ASCII is limited to English characters and *does not support* international text.

## ASCII TABLE

Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	[NULL]	32	20	[SPACE]	64	40	@	96	60	`
1	1	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	2	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	3	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	4	[END OF TRANSMISSION]	36	24	\$	68	44	D	100	64	d
5	5	[ENQUIRY]	37	25	%	69	45	E	101	65	e
6	6	[ACKNOWLEDGE]	38	26	&	70	46	F	102	66	f
7	7	[BELL]	39	27	'	71	47	G	103	67	g
8	8	[BACKSPACE]	40	28	(	72	48	H	104	68	h
9	9	[HORIZONTAL TAB]	41	29	)	73	49	I	105	69	i
10	A	[LINE FEED]	42	2A	*	74	4A	J	106	6A	j
11	B	[VERTICAL TAB]	43	2B	+	75	4B	K	107	6B	k
12	C	[FORM FEED]	44	2C	,	76	4C	L	108	6C	l
13	D	[CARRIAGE RETURN]	45	2D	-	77	4D	M	109	6D	m
14	E	[SHIFT OUT]	46	2E	.	78	4E	N	110	6E	n
15	F	[SHIFT IN]	47	2F	/	79	4F	O	111	6F	o
16	10	[DATA LINK ESCAPE]	48	30	0	80	50	P	112	70	p
17	11	[DEVICE CONTROL 1]	49	31	1	81	51	Q	113	71	q
18	12	[DEVICE CONTROL 2]	50	32	2	82	52	R	114	72	r
19	13	[DEVICE CONTROL 3]	51	33	3	83	53	S	115	73	s
20	14	[DEVICE CONTROL 4]	52	34	4	84	54	T	116	74	t
21	15	[NEGATIVE ACKNOWLEDGE]	53	35	5	85	55	U	117	75	u
22	16	[SYNCHRONOUS IDLE]	54	36	6	86	56	V	118	76	v
23	17	[END OF TRANS. BLOCK]	55	37	7	87	57	W	119	77	w
24	18	[CANCEL]	56	38	8	88	58	X	120	78	x
25	19	[END OF MEDIUM]	57	39	9	89	59	Y	121	79	y
26	1A	[SUBSTITUTE]	58	3A	:	90	5A	Z	122	7A	z
27	1B	[ESCAPE]	59	3B	;	91	5B	[	123	7B	{
28	1C	[FILE SEPARATOR]	60	3C	<	92	5C	\	124	7C	
29	1D	[GROUP SEPARATOR]	61	3D	=	93	5D	]	125	7D	}
30	1E	[RECORD SEPARATOR]	62	3E	>	94	5E	^	126	7E	~
31	1F	[UNIT SEPARATOR]	63	3F	?	95	5F	_	127	7F	[DEL]

## 1.3 Unicode

It is a comprehensive encoding system designed to represent text in most of the world's writing systems. Unlike ASCII, Unicode uses a variable-length encoding system, which can be 8, 16, or 32 bits long.

**NOTE:** This encoding allows to represent over a million unique characters.

	20	21	22	23	24	25	26	27	28	29	2a	2b	2c	2d	2e	2f	30	31	32	33	34	35	36	37	38	39	3a	3b	3c	3d	3e	3f	40
00	!	"	#	\$	%	&	'	(	)	*	+	,	-	.	/	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?	@	
01	À	Á	Â	Ã	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì	Í	Î	Ï	Ð	Ñ	Ò	Ó	Ô	Õ	Ö	×	Ø	Ù	Ú	Û	Ü	Ý	Þ	ß	
02																																	
03																																	
04	Р	С	Т	У	Ф	Х	Ц	Ч	Ш	Щ	Ъ	Ы	Ь	Э	Ю	Я	а	б	в	г	д	е	ж	з	и	й	к	л	м	н	о	п	
05																																	
06																																	
07																																	

## 1.4 ord(), chr()

Python provides two functions to interact with character encodings:

- `ord(c)`: Given a string representing one Unicode character, `ord()` returns an integer representing the Unicode code point of that character. For example, `ord('a')` returns the integer 97.
- `chr()` Given an integer representing a Unicode code point, `chr()` returns a string representing the character.

**NOTE:** This is not used in the production environments, but we will have a homework based on this material, so get acquainted.

### Example

```
print(ord('a'))  
  
print(chr('97'))
```

### Output

```
97  
a
```

These functions allow us to convert between a character's `byte` and its `str` representations.

It will be very helpful once we start working with files later.

## 2. String Formatting

Python provides several ways to format strings. Two common methods are raw strings and formatted string literals.

### 2.1 Raw Strings (r strings)

Raw strings are useful when you need to use lots of backslashes, like defining a file path on Windows/Linux/Mac OS. In a raw string, backslashes are treated as literal characters and not as escape characters, but as a part of the string.

#### Example

```
path = r"C:\Users\Name\Documents\file.txt"

print(path)
```

#### Output

```
C:\Users\Name\Documents\file.txt
```

### 2.2 Formatted strings Literals (f strings)

Formatted string literals, or `f strings`, are a way to embed expressions inside string literals using curly braces `{}`. They are concise and easy to use for embedding variables and expressions within a string.

It really simplifies life using them, just compare two `print()` statements shown in the example below:

#### Example

```
name = "Alice"
age = 25
# Without ``f`` strings
print("Hello, ", name, ".", "You are ", age, "years old.")
# With ``f`` strings
greeting = f"Hello, {name}. You are {age} years old."
print(greeting)
```

#### Output

```
Hello, Alice. You are 25 years old.
```



## 3. Slicing

Slicing is a mechanism in Python which allows you to **extract a part of a string**, also known as a *substring*.

### 3.1 The syntax

A slice is created by specifying two indices in **square brackets**, separated by a colon `[start:stop]` where `start` is the index where the slice starts and `stop` is the index where the slice ends.

**NOTE:** The start index is included in the slice, but the stop index is not.

You can also include a 3rd parameter, `step`, which lets you skip characters within the slice `[start:stop:step]`.

Let's take a look closer on the table below:

Index	0	1	2	3	4	5	6	7	8	9
Char	a	b	c	d	e	f	g	h	i	j
Index	-1 0	-9	-8	-7	-6	-5	-4	-3	-2	-1

We can omit some indexes, and the count will be started from the start/end

### Example

```
s = "abcdefghij"
print(s[1:5]) # Extracts from index 1 to 4
print(s[:5])  # Extracts from the beginning to index 4
print(s[7:])  # Extracts from index 7 to the end
print(s[:])   # Extracts the whole string
print(s[::2]) # Extracts every second character from the string
print(s[::-1]) # Reverses the string
```

### Output

```
bcde
abcde
hij
abcdefghij
```

```
acegi
jihgfedcba
```

**NOTE:** Python also supports negative indices in slicing, which counts **from the end** of the string **backwards**

### Example

```
s = "abcdefghij"
print(s[-4:-1]) # Extracts from the fourth-last to the second-last character
print(s[:-5])   # Extracts from the beginning to the fifth-last character
print(s[-3:])   # Extracts the last three characters
print(s[::-2])  # Extracts every second character from the string in reverse
```

### Output

```
ghi
abcde
hij
jhfdb
```

## 3.2 Assignments

There will be only 2 assignments in this section, but they are a little bit tricky, good luck!

### Assignment 1: Reverse a String

**Objective:** Write a program that takes a string as input and returns it in reverse order using slicing.

```
# Sample Input: 'Python'
# Sample Output: 'nohtyP'
```

### Assignment 2: Palindromic Phrase Verifier

**Objective:** Develop a program that checks whether a word is a palindrome (reads the same backward as forward).

```
Input: The user inputs a word.
Output: The program states whether it is a palindrome.
Example:
Input: radar / level
Output:
This word is a palindrome.
```

## 4. Methods of Strings

In this section we will find out the real power of strings in Python.

### 4.1 Methods vs Functions

In Python, **functions** and **methods** both refer to blocks of code that perform tasks. **The key difference** between them lies in *how they are used and invoked within the code*.

#### 4.1.1 Functions

A **function** is a standalone unit of code that performs a specific task. It can be called directly by its name and can work on different types of data. Functions can take inputs (arguments) and return outputs (results).

##### Example

```
name = "Hello"

print(len(result)) # len() --> function
```

#### 4.1.2 Methods

A method is similar to a function, but **it is associated with an object** and is called on that object using `dot ( . )` notation.

```
name = "hello"

print(result.capitalize()) # capitalize() --> method
```

You have to understand the following key points:

1. Functions are called by name and can be used independently of objects.
2. Methods are called on objects and often work with the data within those objects.

### 4.2 `dir()` and `help()`

The `dir()` function will list all the attributes of any object in Python.

##### Example

```
print(dir(str))
```

## Output

```
['__add__', '__class__', '__contains__', ... 'capitalize', 'casefold',  
'center', 'count', 'encode', 'endswith', 'expandtabs', 'find', 'format', ...]
```

The `help()` function can be used to understand what a particular method does.

```
help(str.capitalize)
```

## Output

```
Help on method_descriptor:  
capitalize(self, /)  
    Return a capitalized version of the string.  
  
    More specifically, make the first character have upper case and the rest  
    lower case.
```

These outputs will provide detailed information about the methods, which can be very helpful when you're trying to figure out how to use a particular string method or which method can be used on the object.

**Note:** Don't forget to refer to [the official documentation for Python](#)

## 4.3 Methods of strings

Method	Description	Example	Output
<code>capitalize()</code>	Capitalizes the first letter of the string.	<code>'hello'.capitalize()</code>	<code>'Hello'</code>
<code>casefold()</code>	Converts string to lower case for caseless matching.	<code>'HELLO'.casefold()</code>	<code>'hello'</code>
<code>center(width)</code>	Centers the string within a specified	<code>'hello'.center(10)</code>	<code>' hello '</code>

Method	Description	Example	Output
	width.		
<code>count(sub)</code>	Returns the number of occurrences of a substring.	<code>'hello'.count('l')</code>	2
<code>endswith(suffix)</code>	Checks if the string ends with the specified suffix.	<code>'hello'.endswith('o')</code>	True
<code>find(sub)</code>	Searches the string for a specified substring and returns the index.	<code>'hello'.find('e')</code>	1
<code>format(*args)</code>	Formats the string into a nicer output.	<code>'{} world'.format('hello')</code>	'hello world'
<code>replace(old, new)</code>	Replaces occurrences of a substring with another.	<code>'hello'.replace('e', 'a')</code>	'hallo'
<code>strip(chars)</code>	Trims leading and trailing characters (whitespace by default).	<code>' hello '.strip()</code>	'hello'
<code>upper()</code>	Converts all characters of the string to uppercase.	<code>'hello'.upper()</code>	'HELLO'

Methods below are used tightly with a `list` data structure, which we will learn during next lessons. We will take a closer look on them later, just bear in mind that they exist!

Method	Description	Example	Output
<code>join(iterable)</code>	Joins the elements of an iterable by the	<code>'-'.join(['1',</code>	'1-2'

Method	Description	Example	Output
	string.	'2'])	
split(sep)	Splits the string at the specified separator.	'1,2,3'.split(',')	['1', '2', '3']

## 4.4 Assignments

### Assignment 1: The Name Corrector

**Objective:** Create a program that takes a user's name and ensures the first letter is capitalized and the rest are lowercase, regardless of how the user enters it.

```
Input: aLiCe           # Note, that you might want to use 2 methods to achieve the
expected output
Output: Alice
```

### Assignment 2: The Substring Counter

**Objective:** Write a script that counts the number of times a substring appears in a given string.

```
Input:
Enter the main string: hellohellohello
Enter the substring to count: ello
Output: The substring 'ello' appears 3 times in 'hellohellohello'.
```

### Assignment 3: The Alignment Formatter

**Objective:** Ask the user to enter a sentence and then display it centered within a frame of a specified width.

```
Input:
Enter your sentence: hello
Set the frame width: 4
Output:
    hello
```

### Assignment 4: The Case Converter

**Objective:** Create a program that can convert a given string into either uppercase or lowercase based on the user's choice.

```
Input:
```

```
Enter your text: Python is Fun!
Choose 'upper' or 'lower': upper
Output: PYTHON IS FUN!
```

## Assignment 5: The URL Corrector

**Objective:** Write a program that ensures all URLs entered by a user start with `https://`. If it doesn't, add it to the beginning.

```
Input: Enter the URL: www.swetrix.com

Output: Corrected URL: https://www.swetrix.com
```

## 5. Introduction to loops

Loops are a powerful feature of computers, giving them the ability to perform *repetitive tasks*.

Python provides two primary types of loops:

- **Counting Loops** (`for`): Ideal for situations where **the number of iterations is known in advance**.
- **Conditional Loops** (`while`): These loops continue to execute **until a certain condition is met**.

### Syntax

```
for variable_name in range(number_of_iterations):

    code_block

# Note that we need to use indentation for this construction
```

### Example

```
for i in range(10):

    print("Hello")
```

### Output

```
# 10 times
Hello
Hello
...
Hello
```

The purpose and function of the `loop variable` may not be immediately obvious. Let's look at an example to clarify

### Example

```
for i in range(10):  
    print(i)
```

### Explanation

When the loop starts, Python assigns **the initial value of the loop variable `i` to 0**.

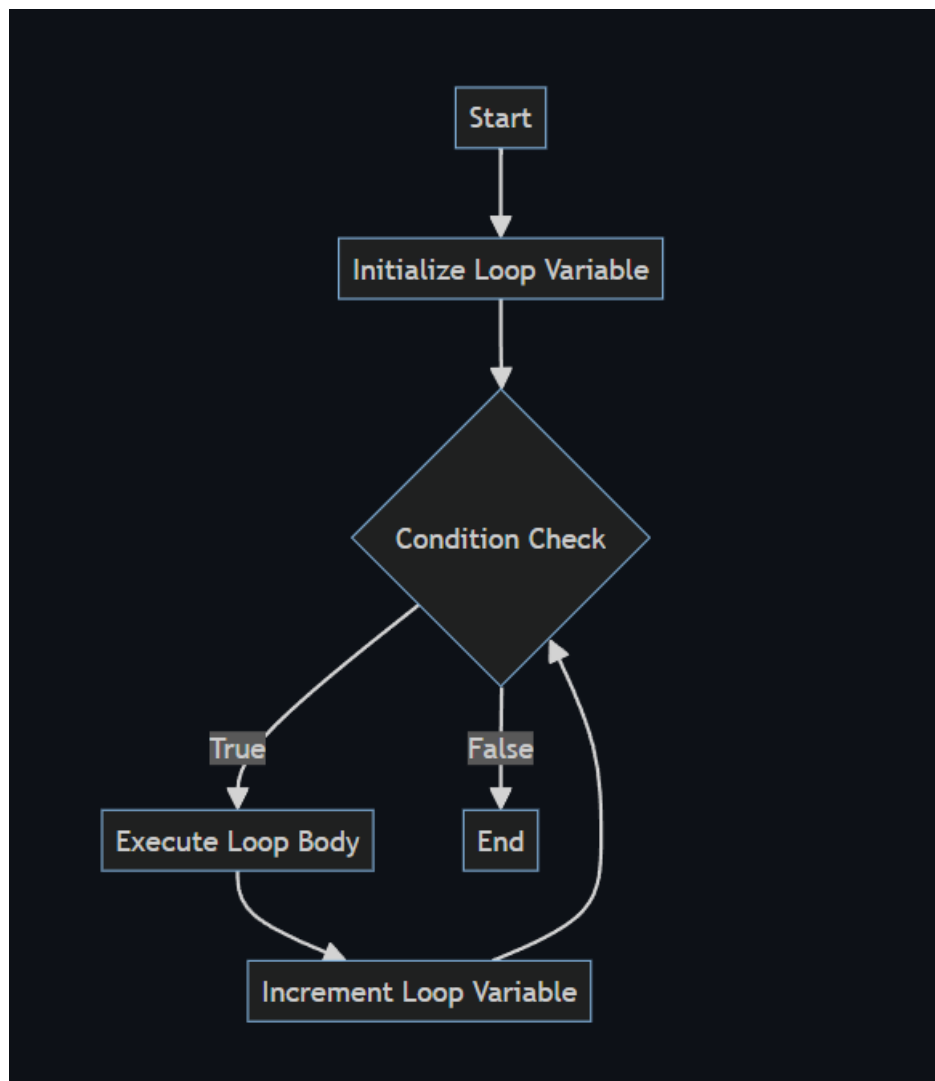
With each iteration of the loop body, **it increments the value of `i` by 1**.

### Output

```
0  
1  
2  
3  
4  
5  
6  
7  
8  
9
```

The diagram shows how `for` loop works in Python.





## 5.1 Naming conventions in Python

Variable names for loops should be:

1. Meaningful and descriptive
2. Shorter names are common as well. Generally, programmers use the letters *i*, *j*, *k* for loop variables.

### Example

```
# Common use
for i in range(10):          # j, k , etc..
    print(i)

# Meaningful and descriptive
for number in range(10):
```

```
print(number)
```

It is important you understand that the *main goal* is to **make code understandable** for people you are working with.

## 5.2 range()

The `range()` function in Python is used to generate a sequence of numbers. It is often used with the `for` loop to iterate over a sequence of numbers as you could see from the examples above.

To be more precise, we can say that the `range(n)` function generates a sequence of numbers from 0 to  $n-1$ , and the `for` loop iterates through this sequence.

### Example

```
print(range(10))
```

### Output

```
range(0, 10)
```

## 5.3 Overloading of range()

The `range()` function in Python can take up to three parameters: `range(start, stop, step)`.

Here's what each parameter represents:

- `start`: The **beginning** of the sequence.
- `stop`: The **end** of the sequence (exclusive -->  $(stop-1)$ ).
- `step`: The **increment** between each number in the sequence.

Function Call	Description	Generated Sequence
<code>range(1, 10, 2)</code>	Starts at 1, ends before 10, steps by 2	1, 3, 5, 7, 9
<code>range(5, 30, 5)</code>	Starts at 5, ends before 30, steps by 5	5, 10, 15, 20, 25
<code>range(10, 1, -1)</code>	Starts at 10, ends before 1, steps by -1	10, 9, 8, 7, 6, 5, 4, 3, 2
<code>range(20, 10, -2)</code>	Starts at 20, ends before 10, steps by -2	20, 18, 16, 14, 12

Function Call	Description	Generated Sequence
<code>range(3, 8, 1)</code>	Starts at 3, ends before 8, steps by 1	3, 4, 5, 6, 7
<code>range(0, -10, -1)</code>	Starts at 0, ends before -10, steps by -1	0, -1, -2, -3, -4, -5, -6, -7, -8, -9

## 5.4 for in / for in range()

There are two different ways to iterate through objects in python ->

for variable in .. and for variable in range()

The `for in` loop is used to iterate over **elements of a sequence**, such as a str, list, string, tuple, or any iterable object which you will learn during the next lessons.

This type of loop goes through each item in the sequence, **one by one**.

### Syntax

```
for element in iterable:
    # Do something with element
```

### Example

```
s = 'Python'

for char in s:
    print(char)          # Directly accessing the element
```

We can iterate through the string using the following construction and `len()` function

```
s = 'Python'

for index in range(len(s)):
    print(s[index])     # Accessing the element of the string using indexing
```

### Output

```
P
Y
```

```
t
h
o
n
```

## 5.5 When to Use Each Type of Loop?

Use `for in` when you have an iterable (like a list or string) and you want to perform actions **on each element**.

Use `for in range()` when you need to **repeat actions a specific number of times**, or when you need to **iterate using a counter**.

### Example

```
# Don't forget that we can pas 1-3 arguments into the `range()` function

for i in range(2, 10, 2):

    print(i)
```

## Assignment 1: The Countdown Timer

**Objective:** Imagine that you are defusing a bomb which will explode in `n` seconds.

1. The program starts with a countdown from a given number `n` to 1.
2. When the countdown reaches 1, prompt the user to choose a wire to cut: red or blue.
3. If the user chooses `red` -> the bomb explodes.
4. If the user chooses `blue` -> the bomb is defused.
5. Make the application very interactive.

```
Input:
Enter the countdown time: 5
Output:
The bomb will explode in 5 seconds!
5
4
3
2
1
Quick! Which wire to cut? Red or Blue?
User Input: Blue
Output:
You cut the blue wire...
The bomb has been defused. Congratulations, you saved the day!
# Alternatively
```

```
User Input: Red
Output:
You cut the red wire...
BOOM!!! The bomb exploded
```

## Assignment 2: The Fibonacci Sequence Generator

**Objective:** Write a program that generates the Fibonacci sequence up to a user-defined number.

Fibonacci Sequence Example: 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89,...

```
Input: The user enters a number `n`.
Output: The program outputs the first `n` numbers of the Fibonacci sequence.
Example:
Input: 5
Output: 0, 1, 1, 2, 3
```

## Assignment 3 The Multiplication Table Printer

**Objective:** Create a script that prints the multiplication table for a number provided by the user.

```
Input: The user inputs a number.
Output: The program prints the multiplication table for that number.
Example:
Input: 3
Output:
3 x 1 = 3
3 x 2 = 6
3 x 3 = 9
...
3 x 10 = 30
```

## 6. Quiz

### Question 1:

Which function returns the Unicode code point for a single character?

- A) `chr()`
- B) `ord()`
- C) `len()`
- D) `str()`

## Question 2:

What does the following string slicing return? `s = 'HelloWorld'; print(s[2:7])`

- A) Hello
  - B) World
  - C) lloWo
  - D) HelloWorld
- 

## Question 3:

Which of the following is the correct way to iterate through the string "Python" using a for loop?

- A) `for i in 'Python': print(i)`
  - B) `for i in range('Python'): print(i)`
  - C) `for i in len('Python'): print(i)`
  - D) `for 'Python': print(i)`
- 

## Question 4:

Which statement about `raw` strings in Python is true?

- A) They treat backslashes as escape characters.
  - B) They cannot contain special characters.
  - C) They treat backslashes as normal characters.
  - D) They start with the letter `r` instead of `R`.
- 

## Question 5:

What will be the output of the following code?

```
x = "awesome"

print(f"Python is {x}")
```

- A) Python is x
  - B) Python is awesome
  - C) Python is {awesome}
  - D) x is awesome
-

### Question 6:

What is the result of the following?

```
print("Hello, World!".find("W"))
```

- A) 6
  - B) 7
  - C) Hello
  - D) -1
- 

### Question 7:

How do you print numbers from 1 to 5 using a for loop in Python?

A)

```
for i in range(1, 6):  
    print(i)
```

B)

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

C)

```
for i in 1 to 5:  
    print(i)
```

D)

```
for i in range(6):  
    print(i)
```

---

### Question 8:

What does the following code return?

```
print("PYTHON".casefold())
```

- A) PYTHON
- B) python

- C) An error
- D) PYthon

## 7. Homework

### Task 1: ASCII/Unicode Converter

**Objective:** Develop an application that can encrypt and decrypt multiple messages using ASCII and Unicode code points.

The program should allow the user to choose between encryption and decryption and specify the number of messages.

```
# Example of how the program should work
# Menu:
# 1. Encrypt Messages
# 2. Decrypt Messages
# Encrypt Input:
Enter the number of messages to encrypt: 2
Enter message 1: Hello
Enter message 2: World
# Encrypt Output:
Encrypted Message 1: 72 101 108 108 111
Encrypted Message 2: 87 111 114 108 100
# Decrypt Input:
Enter the number of messages to decrypt: 2
Enter message 1: 72 101 108 108 111
Enter message 2: 87 111 114 108 100
# Decrypt Output:
Decrypted Message 1: Hello
Decrypted Message 2: World
```

### Task 2: The Custom String Slicer

**Objective:** Develop a program that allows users to input a string and then perform various slicing operations based on the user input. Ask if the user wants to add a step and process the request accordingly

```
Input:
Enter a string: Hello World
Enter start index: 2
Enter stop index: 8
Do you want to add a step (yes/no)? yes
Enter step: 2
Output:
The sliced string with step is: l o
```

### Task 3: The Custom String Slicer



**Objective:** Write a program that builds a story based on the user's choices. Use f strings for dynamic storytelling.

```
Input:
Choose your character's name: Alice
Choose a companion (dog/cat): dog
Choose a destination (forest/beach): forest
Output:
Alice, along with her loyal dog, set out on an adventure to the forest.
[Continue the story...]
# Be creative! You can add more variables and sentences to the story!
```

#### Task 4: The URL Shortener

**Objective::** Develop a simple URL shortener. The program will take a URL and provide a shortened version using slicing and concatenation.

```
Input: Enter a URL: www.example.com

Output: Shortened URL: www.exa...com
```

# Lesson 5: Loops

"Loops are engines of automation in the world of code."

## 1. Introduction to loops

Let's take a look on `while` loop. It runs until the condition for the loop is `True`.

### 1.1 Syntax

```
while condition:

    # Code to execute while the condition is True
```

### 1.2 Practical Use Cases

**Scenario #1:** Validation of the user input.

#### Example

```
# The loop keeps asking for input until the user enters a positive number.
number = 0
while number <= 0:
    number = int(input("Enter a number > 0: "))    # Note: that indent is
                                                    # required to use ``while``
# Exit the loop in case we entered the ``number > 0``
print(f"Your number is > 0 and it is {number}")
```

#### Output

```
Enter a number > 0: -2
Enter a number > 0: 0
Enter a number > 0: 11
Your number is > 0 and it is 11
```

**Scenario #2:** Repeat until specific input

#### Example

```
response = ''

while response.lower() != 'exit':

    response = input("Type 'exit' to stop: ")

print("Exited successfully!")
```

## Output

```
Type 'exit' to stop: what?
Type 'exit' to stop: I don't understand
Type 'exit' to stop: exit
Exited successfully!
```

Loops are extremely powerful tools in programming languages and the examples provided are not *single possible*. Be creative! And come up with solutions, which will satisfy your needs!

### 1.2 while vs for()

- while loops are best used when the number of iterations is not known in advance.
- for loops are ideal when iterating over a sequence or range.

The table provided below helps to make a decision based on the appropriateness of usage for and while

Aspect	while	for
Control	Based on a boolean condition.	Based on iterating over a sequence.
Usage	Unknown number of iterations.	Known number of iterations.
Flexibility	More flexible, can lead to infinite loops.	Less prone to errors, more structured.

Sometimes we can achieve the identical result using either `while` or `for` loops, but with different syntax.

### Example

```
counter = 0                # Define a counter
while counter < 5:
    print(counter)
    counter += 1           # Each iteration add ``1`` to its value
# Add empty line between outputs
print()
# The ``for`` loop serves the same purpose as `while`
for counter in range(5):
    print(counter)
```

## Output

```
0
1
2
3
4
```

```
0
1
2
3
4
```

### 1.3 Infinite loop

Infinite loops occur when the **loop condition is always True**. They should be used extremely **cautiously**. As this can block the program from I/O (input/output) operations.

Make sure that you always exit the infinite loop based on some conditions, using the `break` keyword described in section 2.

#### Example

```
while True:

    print("This loop will run forever...")
```

#### Output

```
# The program prints "This loop will run forever..." indefinitely
```

## 2. `break`, `continue`, and `else` in Loops

Loops in Python can be controlled using `break`, `continue`, and `else` statements. These control statements help in managing loop execution more precisely based on certain conditions.

Keyword	Description
<code>break</code>	Exits the loop immediately, skipping the remaining iterations.
<code>continue</code>	Skips the current iteration and continues with the next one.
<code>else</code>	Executes a block of code after the loop completes normally.

### 2.1 `break`

The `break` statement is used to exit a loop prematurely when a certain condition is met.

#### When to use:

- To stop the loop when a specific condition occurs.
- In search operations when the item is found.

- To avoid unnecessary processing once the goal is achieved.

### Example (for)

**Objective:** We want to exit the loop once number equals to 5.

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

It is important to understand, that we can use these keywords, in any type of loop we want. And put the condition which suits the technical task.

### Example (while)

```
count = 0  
while True:  
    print(count)  
    count += 1  
    if count >= 5:  
        break
```

### Output

```
0  
1  
2  
3  
4
```

## 2.2 continue

The `continue` is used to skip the rest of the loop block and move to the next iteration.

### When to use:

- To skip specific iterations that do not meet certain criteria.
- To avoid executing certain parts of the loop for specific conditions.

### Example (for)

**Objective:** We want to output only odd numbers within `range(0,10)`.

```
for number in range(10):  
    if number % 2 == 0:  
        continue  
    # Skip the iteration once we meet the criteria in ``if`` statement
```

```
print ( number )
```

### Example (while)

```
count = 0
while count < 10:
    count += 1
    if count % 2 == 0:
        continue          # Skip the iteration once we meet the criteria in ``if`` statement
    print ( count )
```

### Output

```
1
3
5
7
9
```

## 2.3 else

The `else` clause is executed after the loop completes its execution without hitting a `break` statement.

#### When to use:

- To check if a loop has completed normally without any `break`.
- To perform certain final actions after loop completion.

### Example

**Objective:** We want to assure that loop has completed without exiting.

```
# Example of else with a for loop
for number in range(3):
    print(number)
    if number >= 2:
        print('123213')
        break
else:
    print("No Break")
0
1
2
```

With control statements, you can create loops that handle a wide range of dynamic conditions and scenarios. Try it yourself!

## 2.4 Assignments

### Assignment 1: The Menu Navigator

**Objective:** Write a program that simulates a menu navigation system using a while loop and break statement. The user can select from a list of options, and the loop terminates when the user selects the 'Exit' option.

```
Input:
1. Start Game
2. Load Game
3. Exit
Choose an option: 3
Output:
Exiting the menu...
```

### Assignment 2: The Odd Number Skipper

**Objective:** Create a program that prints out all numbers from 1 to 20, but skips odd numbers using one of control statements.

```
Output:
2
4
6
8
10
12
14
16
18
20
```

### Assignment 3: The Prime Number Identifier

**Objective:** Develop a script that identifies whether a number is prime. Use a for else construction. If the loop completes without breaking, the number is prime.

```
Input: Enter a number: 11
```

```
Output: 11 is a prime number.
```

## Assignment 4: The Escape Room

**Objective:** Simulate an escape room challenge where the user must find the correct code to "escape the room." The loop should break once the correct code is entered.

```
Input:
Enter the escape code: 1234
Wrong code, try again.
Enter the escape code: 9999
Correct! You've escaped!
Output:
You've successfully escaped the room!
```

## Assignment 5: The Input Validator

**Objective:** Write a program that asks users to enter their age and ensures the input is valid using a while loop. If the input is not a number, use continue to prompt them again until a valid number is entered.

```
Input:
Enter your age: twenty
Invalid input, please enter a number.
Enter your age: 20
Output:
Age entered: 20
```

# 3. Nested Loops

Nested loops in Python are when you have *a loop running inside another loop*.

Sounds scary, right? I remember myself, when I started learning programming, the concept of nested loops was one of the hardest to understand.

I wanted to quit programming studying this section. But now, I will try to explain everything to you to make nested loops clear! Don't be afraid of *nested loops*!

## Example

The great example can be a clock where:

1. The hour hand *completes one full rotation* while the minute hand *completes 12 full*.



2. The minute hand *completes one full rotation (outer loop)*, while the second hand completes 60 full rotations (inner loop).

**NOTE:** The inner loop completes its execution for every iteration of the outer loop.

```
# Outer loop for the hour hand
for hour in range(12):
    # Inner loop for the minute hand
    for minute in range(60):
        # Another inner loop for the second hand
        for second in range(60):
            print(f"{hour} hours {minute} minutes {second} seconds")
```

## Output

```
# Firstly the ``second's`` loop is going to be executed
0 hours 0 minutes 0 seconds
0 hours 0 minutes 1 seconds
0 hours 0 minutes 2 seconds
0 hours 0 minutes 3 seconds
0 hours 0 minutes 4 seconds
# Secondly, ``minutes`` inner loop and lastly, the ``hours``
```

## Example

Imagine fans in a stadium chanting. For each round (outer loop) they clap several times (inner loop).

```
# Outer loop for each chanting round
for round in range(3):
    print("Let's go, team!")
    # Inner loop for clapping
    for clap in range(3):
        print("Clap!", end=' ')
    print() # New line after the claps
```

## Output

```
Let's go, team!
Clap! Clap! Clap!
Let's go, team!
Clap! Clap! Clap!
Let's go, team!
Clap! Clap! Clap!
```

## Example

Let's create a multiplication table for numbers 1 to 3, up to the times 10.

The outer loop can iterate over the numbers you want to multiply, and the inner loop can iterate over the range of multipliers.

```
# For numbers 1, 2, 3
for i in range(1, 4):
    # From 1 times to 10 times
    for j in range(1, 11):
        print(f"{i} x {j} = {i*j}")
    print() # New line
```

## Output

```
# 1st work of the outer loop
1 x 1 = 1
1 x 2 = 2
...
1 x 10 = 10
# 2nd work of the outer loop
2 x 1 = 2
2 x 2 = 4
...
2 x 10 = 20
# 3rd work of the outer loop
3 x 1 = 3
3 x 2 = 6
...
3 x 9 = 27
```

You can use [Python Visualizer](#) in order to understand the concept of nested loops better.

There are lots of practical cases where software engineer can use nested loops. You will notice, that we sometimes coming back to them working with advanced data structures.

## 3.1 Assignments

### Assignment 1: The Nested Countdown

**Objective:** Write a program that simulates a digital clock counting down hours, minutes, and seconds from a given time.

```
Input:
Enter the countdown start time (hh mm ss): 01 30 00
Output:
1 hours 29 minutes 59 seconds
1 hours 29 minutes 58 seconds
```

```
...
0 hours 0 minutes 1 seconds
Time's up!
```

## Assignment 2: The Pattern Printer

**Objective:** Create a program that prints out a pattern of stars, forming a right-angled triangle.

```
Input:
Enter the number of rows for the triangle: 5
Output:
*
**
***
****
*****
```

## Assignment 3: The Table Matrix

**Objective:** Develop a script that prints out a matrix of numbers, where each row contains numbers incremented by 1 from the previous row.

```
Input:
Enter the number of rows: 3
Enter the number of columns: 4
Output:
1 2 3 4
2 3 4 5
3 4 5 6
```

# 4. Quiz

## Question 1:

What is the output of the following code snippet?

```
i = 3
while i > 0:
    i -= 1
    if i == 2:
        break
else:
    print("Loop ended without break.")
```

---

## Question 2:

Which of the following is an infinite loop?

A)

```
i = 5
while i > 0:
    print(i)
    i -= 1
```

B)

```
while True:
    print("Python")
```

C)

```
for i in range(5, 0, -1):
    print(i)
```

---

### Question 3:

What will be printed by the following code?

```
for i in range(3):
    for j in range(2):
        print(j, end='')
    print(i)
```

---

### Question 4:

What does the continue keyword do in a loop?

- A) Exits the loop
  - B) Skips to the next iteration of the loop
  - C) Does nothing
  - D) Restarts the loop
- 

### Question 5:

When will the 'else' part of a 'loop' execute?

- A) When the loop finishes normally without encountering a break statement
- B) When the loop encounters a break statement
- C) At the end of each loop iteration
- D) The else part is not valid for loops

## 5. Homework

### Task 1: The Guessing Game

**Objective::** Write a program that selects a random number and asks the user to guess it. Use a while loop to allow multiple attempts until they guess correctly or choose to exit.

```
Input:
Guess the number: 8
Wrong! Try again or type 'exit' to stop: 5
Wrong! Try again or type 'exit' to stop: exit
Output:
The correct number was 7. Better luck next time!
```

### Task 2: Nested loops

**Objective::** Write a program using nested loops to match the following output.

```
Output:
1
22
333
4444
55555
```

### Task 3: The Factorial Calculator

**Objective::** Create a program that computes the factorial of a given number using a for loop.

```
Input:
Enter a number to calculate the factorial: 5
Output:
The factorial of 5 is 120.
```

# Lesson 6: Data Structures p.1 (`list`)

"Lists are the backbone of data organization."

## 1 Mutable and Immutable data types

Mutable data types - are those that *can be changed after they are created*.

This means you can change, add, or remove elements from these data types *without creating a new object*.

In programming, understanding the difference between mutable and immutable data types is *extremely crucial* because it affects how you work with data and how your program behaves.

This lesson we will be focusing on the `list`. But during next lessons we will get acquainted with `tuple`, `set` and `dict` data structures. So it will be the best to know which of them are mutable/immutable now.

### 1.1 Mutable

Data Structure	Description	Example Operations
List	Ordered and changeable collection of items	<code>append()</code> , <code>remove()</code> , <code>pop()</code>
Dictionary	Unordered collection of key-value pairs	<code>update()</code> , <code>clear()</code> , <code>pop()</code>
Set	Unordered collection of unique items	<code>add()</code> , <code>discard()</code> , <code>pop()</code>

### 1.2 Immutable

Immutable data types cannot be changed after their creation. Any "modification" **creates a new object** instead of changing the old one.

Data Type	Description	Example Operations
Tuple	Ordered and unchangeable collection	Accessing items, <code>index()</code> , <code>count()</code>
String	Sequence of characters	Accessing characters, concatenation creates new string
Integer	Whole number without decimal	Arithmetic operations create new integers

Data Type	Description	Example Operations
Float	Number with decimal point	Arithmetic operations create new floats
Boolean	Represents True or False	Used in logical operations

We have already seen the error trying to modify strings in the Lesson #3. Instead of this we created a **new** variable to store a **new** object.

### Example

```
greeting = "Hello, world!"
print(f"Original string: {greeting}")
# Attempting to change a character
greeting[0] = 'J'          # This line will actually raise an error: TypeError
# Concatenation creates a new string and
greeting = 'J' + greeting[1:]
print(f"Modified string: {greeting}")
```

### Output

```
Original string: Hello, world!
Traceback (most recent call last):
  File "<string>", line 5, in <module>
TypeError: 'str' object does not support item assignment
```

But how to verify if we created a new object? This can be done using `id()` function.

## 1.3 `id()`

The `id()` function in Python returns a **unique identifier for an object**, which corresponds to its location in memory. Actually, almost any operation that seems to modify the object actually creates a new object, if you work with *immutable* object.

If the `id()` is different, it confirms that a new object was created. This can be helpful while debugging the project.

### Example

```
# Immutable data type
greeting = "Hello, world!"
original_id = id(greeting)
greeting = 'J' + greeting[1:]
new_id = id(greeting)
# Comparing identifiers
print(f"Original ID: {original_id}")
print(f"Modified ID: {new_id}")
```

## Output

```
Original ID: 140353776296400  
Modified ID: 140353776295680
```

## 2. Introduction to `list`

In Python a `list` - is a *mutable, ordered sequence of items*.

It is one of the most versatile data types and can contain items of different data types, though typically, lists contain items of the same type.

### 2.1 Syntax

In order to create an empty `list` we can use the following syntax:

#### Example

```
list_1 = []  
list_2 = list()  
print(list_1, list_2)  
print(type(list_1), type(list_2))
```

## Output

```
[] []  
<class 'list'> <class 'list'>
```

### 2.2 Data Type Conversion

Most iterable types in Python can be converted to a list using the list constructor.

Simply speaking, `iterable` types - objects that can give you elements one by one, which allows you to iterate through them using loops or advanced techniques such as iterators/generators.

#### Example

```
string_to_list = list('hello')  
print(string_to_list)
```

## Output

```
['h', 'e', 'l', 'l', 'o']
```



## 2.3 Indexing and slicing

Indexing and slicing work the same way for lists as they do with strings.

Indexing -> access *individual* items.

slicing -> access *a range of* items.

### Example

```
my_list = [10, 20, 30, 40, 50]
# Indexing
print(my_list[1]) # 20
# Slicing
print(my_list[1:4]) # [20, 30, 40]
```

## 2.4 Concatenation

Lists can be concatenated using the + operator, which combines them into a new list.

### Example

```
list_one = [1, 2, 3]
list_two = [4, 5, 6]
list_three = [7, 8, 9]
# Concat
new_list = list_one + list_two + list_three
print(new_list)
```

### Output

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

## 2.5 Multiplication

In Python, lists can be *multiplied by an integer, which repeats the list's elements*.

### Example

```
# Example 1
numbers = [1, 2, 3]
new_numbers = numbers * 3
print(new_numbers)
# Example 2
names = ["Sasha", "Yehor"]
triple_names = names * 3
print(triple_names)
```

## Output

```
[1, 2, 3, 1, 2, 3, 1, 2, 3]
['Sasha', 'Yehor', 'Sasha', 'Yehor', 'Sasha', 'Yehor']
```

In some cases this can be very useful and it's very cool that Python provides such functionality.

## 2.6 Operators += / \*= and unpacking

The += and \*= operators can modify lists in place. These operators perform the same operations as `list + list` and `list * int`. respectively.

### Example

```
# +=
nums = [1, 2, 3]
nums += [4, 5] # Same as ``nums = nums + [4,5]``
print(nums)
# *=
nums = [1, 2, 3]
nums *= 2 # Same as ``nums = nums * 2``
print(nums)
```

## Output

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

## 2.7 Unpacking

Unpacking is the way of extracting values from iterable objects.

There are several types of unpacking presented in Python.

### 2.7.1 Basic Unpacking

The number of variables *on the left side of the assignment operator* **should match** the number of elements in the iterable.

### Example

```
my_list = [1, 2, 3]
a, b, c = my_list # Unpacking list into three variables
print(a) # 1
```

```
print(b) # 2
print(c) # 3
```

## 2.7.2 Extended Unpacking

Using a star `*` expression, a.k.a (asterix), allows you to assign *a portion of an iterable to a variable and the rest to another variable*.

### Output

```
my_list = [1, 2, 3, 4, 5]
first, *middle, last = my_list
print(first) # 1
print(middle) # [2, 3, 4]
print(last) # 5
```

## 2.7.3 Swapping Variables

This lifehack can be used to swap the values of two variables efficiently without needing a temporary variable.

### Example

```
x, y = 10, 20
x, y = y, x # Swapping values
print(x) # 20
print(y) # 10
```

## 2.7.4 Unpacking with `for`

As it was mentioned before, `list` can consist of different types of elements and it being a `list` of `lists`. That's where nested loops come in handy. Alternatively, you can use that to unpack variables with the same format.

### Example

```
nested_lists = [[1, 'apple'], [2, 'banana'], [3, 'cherry']]
for number, fruit in nested_lists:
    print(f"Number: {number}, Fruit: {fruit}")
```

## Output

```
Number: 1, Fruit: apple
Number: 2, Fruit: banana
Number: 3, Fruit: cherry
```

## 2.8 Operator `in`

The `in` operator in Python is used to check if a value exists within all iterable objects.

### Example

```
my_list = [1, 2, 3, 4, 5, 'banana']
print(3 in my_list) # True
print(6 in my_list) # False
print("banana" in my_list) # True
print("a" in my_list[-1]) # True
```

## 3. Functions (`len()`, `sum()`, `min()`, `max()`, `sorted()`)

Python provides several built-in functions that are readily available for common list operations. These functions can quickly perform actions on list items without the need for loops.

They work with *all iterables* because they use polymorphism concept which we will explore in OOP (Object Oriented Programming).

### 3.1 `len()`

The `len()` function returns the number of items in a list (the length of the list).

### Example

```
numbers = [1, 2, 3, 4, 5]
print(len(numbers))
matrix = [[1,2,3], [4,5,6]] # we have 2 lists in the list, heh
print(len(matrix))
```

## Output

```
5
2
```

### 3.2 `sum()`

The `sum()` function calculates the total sum of *all numerical* items in a list.

#### Example

```
numbers = [1, 2, 3, 4, 5]
print(sum(numbers))
# This won't work, as we need only ``numerical data types``
imposter_numbers = ["STRING???", 15]
print(sum(imposter_numbers))
```

#### Output

```
15
Traceback (most recent call last):
  File "<string>", line 5, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

### 3.3 `min()`

The `min()` function returns the smallest item in a list.

#### Example

```
numbers = [5, 1, 8, 3, 2]
print(min(numbers))
# This won't work, as well
imposter_numbers = [[1, 2, 3], 15]
print(min(imposter_numbers))
```

#### Output

```
1
Traceback (most recent call last):
  File "<string>", line 2, in <module>
TypeError: '<' not supported between instances of 'list' and 'int'
```

### 3.4 `max()`

The `max()` function returns the biggest item in a list.

#### Example

```
numbers = [5, 1, 8, 3, 2]
print(max(numbers)) # Output: 8
```

```
# Nope, won't work, don't even try to do this!
imposter_numbers = ['s', 15]
print(min(imposter_numbers))
```

## Output

```
8
Traceback (most recent call last):
  File "<string>", line 2, in <module>
TypeError: '<' not supported between instances of 'list' and 'int'
```

## 4. Methods of Lists

Suppose we have the following list:

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

### 4.1 Adding Elements

These methods let you add elements to the list, either at the end, a specific position, or extending the list by appending all elements from another iterable (str, list, tuple, dict, etc..) / numbers.

Method	Description	Example	Output
<code>append(x)</code>	Adds an item to the end of the list.	<code>x.append(4)</code>	<code>[1, 2, 3, 4]</code>
<code>extend(iterable)</code>	Extends the list by appending elements from the iterable.	<code>x.extend([5, 6])</code>	<code>[1, 2, 3, 4, 5, 6]</code>
<code>insert(i, x)</code>	Inserts an item at a given position.	<code>x.insert(2, 'a')</code>	<code>[1, 2, 'a', 3, 4]</code>

### 4.2 Removing Elements

Method	Description	Example	Output
<code>remove(x)</code>	Removes the first item from the list whose value is x.	<code>x.remove('a')</code>	<code>[1, 2, 3, 4]</code>

Method	Description	Example	Output
<code>pop([i])</code>	Removes the item at the given position in the list, and returns it.	<code>x.pop(1)</code>	<code>[1, 3, 4]</code>
<code>clear()</code>	Removes all items from the list.	<code>x.clear()</code>	<code>[]</code>

### 4.3 Utility Methods

Method	Description	Example	Output
<code>index(x)</code>	Returns the index of the first item whose value is x.	<code>x.index(3)</code>	<code>2</code>
<code>count(x)</code>	Returns the number of times x appears in the list.	<code>x.count(1)</code>	<code>1</code>
<code>sort()</code>	Sorts the items of the list in place.	<code>x.sort()</code>	<code>[1, 2, 3, 4]</code>
<code>reverse()</code>	Reverses the elements of the list in place.	<code>x.reverse()</code>	<code>[4, 3, 2, 1]</code>
<code>copy()</code>	Returns a <i>shallow copy</i> of the list.	<code>new_x = x.copy()</code>	<code>new_x</code> is <code>[1, 2, 3, 4]</code>

Don't hesitate to use `dir()`, and `help()` functions or refer to the official Python documentation working with list.

## 5. General methods of Iterations

Iterating through lists is a fundamental aspect of working with data in Python, as it allows you to access and manipulate each item in a list.

As with strings, Python provides several ways to iterate over lists.

### 5.1 Using `for in` Loop

The `for in` loop is the most common way to iterate through each item in a list directly.

#### Example

```
fruits = ['apple', 'banana', 'cherry']
for fruit in fruits:
    print(fruit)
```

#### Output

```
apple  
banana  
cherry
```

### Example

```
cities = ['London', 'Kyiv', 'Washington']  
for city in cities:  
    print(city)
```

### Output

```
London  
Kyiv  
Washington
```

Don't forget to call variables which make sense for data structures like in the examples above.

## 5.2 Using `for index in range(len(list))`

Sometimes, you need the index of the items while iterating through a list. In this case, you can use `range()` along with `len()` to iterate through the list indexes.

### Example

```
fruits = ['apple', 'banana', 'cherry']  
for index in range(len(fruits)):  
    print(f"Index: {index}, Fruit: {fruits[index]}")
```

### Output

```
Index: 0, Fruit: apple  
Index: 1, Fruit: banana  
Index: 2, Fruit: cherry
```

This method is useful when you need to modify the list items since you can use the index to set new values.

## 6. Copying lists



Copying lists can be tricky, especially for beginners, because of the way Python handles object references.

There are two primary ways how we can copy list objects:

## 6.1 Shallow Copying

Shallow copying **creates a new list object**, but **it doesn't create new objects for the elements inside the list**. It only copies references to those elements.

Be carefull using this copying.

### Example

```
original_list = [1, 2, 3]
copied_list = original_list.copy()
print("Original List:", id(original_list))
print("Copied List:", id(copied_list))
```

In this case, `original_list` and `copied_list` have different ids, **meaning they are different objects**.

### Output

```
Original List: 139827368587200
Copied List: 139827364976256
```

However, the elements **inside them** are **still referencing the same objects**.

### Example

```
print(id(original_list[1]))
print(id(copied_list[1]))
```

### Output

```
139827364976123
139827364976123
```

## 6.2 Assigning a reference

Assigning a reference to a list in Python is an operation that can lead to **unintended consequences**. It's often confused with creating a copy of the list, but in reality, it's quite different.

### Example

What Happens When You Assign a Reference?

```
original_list = [1, 2, 3]
reference_list = original_list    # assign a list to one and another variable
# Output the ids
print("Original_list_id: ", id(original_list))
print("Reference_list_id: ", id(reference_list))
# Made a change to both ``original_list`` and ``reference_list``
original_list += [4]
reference_list += [5]
# Output lists
print("Original List:", original_list)
print("Reference List:", reference_list)
```

Any changes made to `original_list` will also appear in `reference_list`, and vice versa. This is because they are **two names for the same object**.

If your intention was to work with two separate lists, this behavior could lead to bugs. Changes you thought were made to only one list can unexpectedly affect the other.

### Output

```
Original_list_id: 140583473131392
Reference_list_id: 140583473131392
Original List: [1, 2, 3, 4, 5]
Reference List: [1, 2, 3, 4, 5]
```

## 6.3 Deep Copying Lists

Imagine, you have lists containing other mutable objects (e.g list of lists), it's crucial to understand the concept of deep copying, when working with more complex data structures.

A deep copy creates a new list with entirely new objects, *even for nested mutable objects*.

### How it works?

1. Creating a New List: A **completely new list object** is created.

2. Copying Nested Objects: Even the *nested objects within the list are copied and recreated* in the new list.
3. Independent Objects: Changes made to the deep-copied list (or its nested objects) *do not affect the original list, and vice versa*.

### Example

```
"""Python provides a module named `copy`, which has a method `deepcopy()` for
this purpose."""
import copy
# Original list with nested mutable objects
original_list = [[1, 2, 3], [4, 5, 6]]
# Creating a deep copy
deep_copied_list = copy.deepcopy(original_list)
# Modifying the deep copied list
deep_copied_list[0][1] = 'Changed'
print("Original List:", original_list)
print("Deep Copied List:", deep_copied_list)
```

### Output

```
Original List: [[1, 2, 3], [4, 5, 6]]
Deep Copied List: [[1, 'Changed', 3], [4, 5, 6]]
```

Deep copying is more memory-intensive and can be slower for large lists. So use it wisely. For example, when your list contains nested mutable objects, and you need complete independence between the original and copied data.

## 7. List comprehensions

List comprehensions provide a more readable and concise way to create lists by transforming each element of an iterable.

They can be used in place of for loops for simplicity and efficiency, particularly when you're applying a single, straightforward transformation or condition to each element. Btw, they work faster than default loops.

### 7.1 Syntax

```
# Default option
[expression for item in iterable]
# If clause
[expression for item in iterable if condition]
# If-else clause
[if expression else expression for item in itreable]
```

Basically it is the same as for loops and each list comprehension *can be rewritten* into the for loop.

### Example

```
# List comprehension
squares = [i ** 2 for i in range(5)]
# For loop equivalent
squares = []
for i in range(5):
    squares.append(i ** 2)
```

### Output

```
[0, 1, 4, 9, 16]
```

### Example

```
# List comprehension
even_squares = [i ** 2 for i in range(10) if i % 2 == 0]
# For loop
even_squares = []
for i in range(10):
    if i % 2 == 0:
        even_squares.append(i ** 2)
```

### Output

```
[0, 4, 16, 36, 64]
```

### Example

```
# List comprehension
even_or_odd = ["even" if i % 2 == 0 else "odd" for i in range(5)]
# For loop
even_or_odd = []
for i in range(5):
    if i % 2 == 0:
        even_or_odd.append("even")
    else:
        even_or_odd.append("odd")
```

### Output

```
["even", "odd", "even", "odd", "even"]
```

But there is some known rules between programmers where we **should** or **shouldn't** use them. In some cases for simple operation it is a great tool which working much faster in terms of efficiency, but in other cases it can lead to unnecessary complexon which will result in bad maintainability and readability of the code. Please, be careful using list comprehensions.

## 7.2 For-Loop vs List Comprehension

	Loops	List Comprehension
Advantages	<ul style="list-style-type: none"><li>- More flexible</li><li>- Easier to read with complex logic</li><li>- Allows complex operations</li></ul>	<ul style="list-style-type: none"><li>- More concise and readable for simple cases</li><li>- Can be written as a single line</li><li>- Often faster for creating a list</li></ul>
Disadvantages	<ul style="list-style-type: none"><li>- Can be verbose for simple operations</li></ul>	<ul style="list-style-type: none"><li>- Can be less readable with complex expressions</li><li>- Not suitable for multi-step operations</li></ul>
When to Use	<ul style="list-style-type: none"><li>- When you have complex logic</li><li>- When the loop involves nested loops, conditions, or other complex operations</li></ul>	<ul style="list-style-type: none"><li>- When you're applying a single operation to all items</li><li>- When you need a quick and readable one-liner to create/update a list</li></ul>

**Note:** Always prioritize *readability* and *maintainability* of your code when choosing between a for-loop and a list comprehension.

## 8. `join()` and `split()`

In Python, `join()` and `split()` are two powerful methods that bridge the gap between strings and lists. They are widely used in data manipulation, especially in tasks involving *text processing* and *data formatting*.

### 8.1 `join()`

The `join()` method in Python is a string method that takes all items in an iterable and joins them into one string.

```
words = ['Python', 'is', 'extremely', 'powerful', 'and', 'complicated'
         'language']

s = ' '.join(words)

print(s)
```

### Output

```
Python is extremely powerful and complicated language
```

```
# I'm kidding Assembler is harder :)
```

A string can be specified as the separator, by default it is a whitespace as you have seen from the example above.

### Example

```
separator = ', '  
my_list = ['apple', 'banana', 'cherry']  
joined_string = separator.join(my_list)  
print(joined_string)
```

### Output

```
apple, banana, cherry
```

## 8.2 split()

The `split()` method *splits a string into a list where each word is a list item*.

The splitting is done at the specified separator. If no separator is defined, whitespace is used by default.

### Example

```
text = 'apple, banana, cherry'  
split_list = text.split(', '  
print(split_list)
```

### Output

```
['apple', 'banana', 'cherry']
```

### Example

```
text = 'one two three four'  
split_text = text.split(' ', 2) # You can the max number of splits as well.  
print(split_list)
```

## Output

```
['one', 'two', 'three four']
```

These methods are especially powerful when used together, allowing for back-and-forth transformations between strings and lists.

Sure, here's the corrected and reformatted list for your quiz:

## 9. Quiz

### Question 1:

What will be the output of the following code?

```
my_list = [10, 20, 30, 40, 50]
print(my_list[-2])
```

---

### Question 2:

What will be the output of the following code snippet?

```
my_list = [1, 2, 3, 4, 5]
my_list.append(6)
print(my_list)
```

---

### Question 3:

Which of the following is a correct way to create a deep copy of a list?

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

- A) `new_list = original_list`
- B) `new_list = original_list.copy()`
- C) `new_list = list(original_list)`
- D)

```
import copy
```

```
new_list = copy.deepcopy(original_list)
```

---

#### Question 4:

What does the `extend()` method do to a list?

- A) Extends the list by adding a single element at the end.
  - B) Increases the size of the list without adding elements.
  - C) Adds multiple elements at the specified index of the list.
  - D) Appends all elements of an iterable to the end of the list.
- 

#### Question 5:

What is the output of the following list comprehension?

```
numbers = [x for x in range(10) if x % 2 == 1]
```

- A) [0, 2, 4, 6, 8]
  - B) [1, 3, 5, 7, 9]
  - C) [2, 4, 6, 8, 10]
  - D) [1, 2, 3, 4, 5]
- 

#### Question 6:

What is the result of the following operation?

```
my_list = [1, 2, 3]
my_list.insert(1, 'a')
print(my_list)
```

- A) [1, 'a', 2, 3]
  - B) ['a', 1, 2, 3]
  - C) [1, 2, 'a', 3]
  - D) An error occurs
- 

#### Question 7:



What is the purpose of the `reverse()` method in a list?

- A) To sort the list in ascending order.
- B) To randomly shuffle the elements of the list.
- C) To reverse the order of the elements in the list.
- D) To remove the last element of the list.

### Question 8:

Which of the following is a valid example of slicing a list?

- A) `my_list[1:]`
  - B) `my_list[:1]`
  - C) `my_list[1:3]`
  - D) All of the above
- 

### Question 9:

Which of the following statements about list comprehensions is NOT true?

- A) They are more memory-efficient than for-loops.
  - B) They can include conditional logic.
  - C) They can make code less readable if overused.
  - D) They cannot be used to create lists from other iterables.
- 

### Question 10:

How does `pop()` differ from `remove()` when used on a list?

- A) `pop()` deletes an item by value, whereas `remove()` deletes by index.
  - B) `pop()` returns the removed item, whereas `remove()` does not.
  - C) `remove()` can delete multiple items, whereas `pop()` can only delete one.
  - D) There is no difference in their functionality.
- 

This format maintains consistency and ensures each question and set of options is clearly defined for participants in the quiz.

## 10. Homework

### Task 1: List Manipulator

**Objective:** Create a program that allows the user to manipulate a list in various ways. The user can choose to add, remove, or modify items in the list.

```
# Example of how the program should work
# Menu:
# 1. Add an item
# 2. Remove an item
# 3. Modify an item
# User chooses option 1:
Enter the item to add: Apple
Output: List: ['Apple']
# User chooses option 2:
Enter the item to remove: Apple
Output: List: []
# User chooses option 3:
Enter the item to modify: Apple
Enter the new item: Banana
Output: List: ['Banana']
```

## Task 2: Slicer and Dicer

**Objective:** Develop a program that allows users to input a list and then perform various slicing operations based on the user input.

```
Input:
Enter a list of numbers (separated by space): 10 20 30 40 50
Enter start index for slicing: 1
Enter stop index for slicing: 4
Output:
The sliced list is: [20, 30, 40]
```

## Task 3: Shopping List Organizer

**Objective:** Develop a program that helps users organize their shopping list by adding, viewing, removing and sort items alphabetically.

Additionally, we would want to check if we add strings to the list. You can use `isinstance()` function documented [here](#).

```
# Example of how the program should work
# Menu:
# 1. Add item
# 2. View list
# 3. Remove item
# 4. Exit
```

```
# User Interaction Example:
```

```
Choose an action (add/view/remove/exit): add
Enter an item to add: Milk
Item added. Your current list is: ['Milk']
Choose an action (add/view/remove/exit): add
Enter an item to add: Eggs
Item added. Your current list is: ['Eggs', 'Milk']
```

"The code will be continued on the following page..."

```
Choose an action (add/view/remove/exit): add
Enter an item to add: 25
Sorry, it is not a string
Choose an action (add/view/remove/exit): view
Shopping List: ['Eggs', 'Milk']
Choose an action (add/view/remove/exit): remove
Enter an item to remove: Milk
Item removed. Your current list is: ['Apples']
Choose an action (add/view/remove/exit): exit
Exiting the program.
# Don't forget to check if item exists in the list using operator ``in``
```

## Task 4: Average of Evens

**Objective:** Generate a list of even elements using range from the input and display the average of them.

```
# Write a list comprehension here:
```

## Task 5: List of numbers operations

**Objective:** Enhance the provided code to perform the following actions on a given list:

- Display the length of the list.
- Print the sum, max, min and the average.
- Print the last element of the list.
- Output the list in reverse order (remember to use slicing).
- Print "YES" if the list contains the numbers 4 and 9, and "NO" otherwise.
- Show the list with the first and last elements removed (create a new list object instead).
- Print the third element from the end of the list.
- If the second element is > 10, replace it with 10.

```
# Sample List
numbers = [3, 12, 5, 8, 4, 9, 15, 22]
# Your code here
```

## Task 6: ASCII alphabet

**Objective:** Write a program which creates alphabet and stores it into the `list`.

```
# Output

['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o',
'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z']
```

## Task 7: Mini Database

**Objective:** Develop a program that stores detailed information about a person using nested lists.

```
Input:
Enter your name: Alice
Enter your age: 30
Enter your hobbies (separated by commas): reading, gardening, gaming
Enter your favorite foods (separated by commas): pizza, salad, ice cream
Output:
Personal Information:
Name: Alice
Age: 30
Hobbies: ['reading', 'gardening', 'gaming']
Favorite Foods: ['pizza', 'salad', 'ice cream']
# Later on, the program can provide options to view, update and remove these details.
```

## Task 8: Validating IP Addresses

**Objective:** Write a program that validates if the entered string is a correct IP address.

**Note:** An IP address is considered valid if it consists of four non-negative integers separated by dots, with each integer ranging from 0 to 254 inclusive.

```
## Example
Input: 192.168.1.1
Output: Correct
Input: 256.300.2.10
Output: Incorrect
```

## Task 9: Email Extractor

**Objective:** Create a program to extract email addresses from a block of text. The program should identify strings that resemble email addresses and separate them using semicolons.

```
# Input:  
Enter the text: Please contact us at support@example.com or sales@example.com  
for assistance.  
# Output:  
Extracted Emails: support@example.com; sales@example.com
```

Don't forget that while designing your program, remember to create an interactive and user-friendly interface!

# Lesson 7: Data Structures p.2 (<sub>tuple</sub>)

Tuples are the static snapshots of data, providing a reliable and immutable record of information in code."

## 1. Overview of Tuples

In Python a tuple - is an *immutable, ordered sequence of items*.

Similar to `lists`, tuples can contain elements of different data types, but **cannot be modified after their creation**.

Why do we need them, if we have `lists`?

Advantage	Description
Memory Efficiency	Have a fixed size, which makes them more memory-efficient than <code>lists</code> .
Faster than Lists	Due to their immutability, tuples can be slightly faster than <code>lists</code> when iterating through large datasets.
Fixed Data	Ideal for storing data that shouldn't change over time, such as configuration values or constants.
Easier Debugging	With immutability, it's much easier to track changes and debug your code, as tuples don't change state unexpectedly.

There are more advantages which we will see during expanding of our knowledge into the next lessons:

Advantage	Description
Suitable as Dictionary Keys	Tuples can be used as keys in dictionaries due to their immutability.
Functionality in Functions	Useful for passing multiple values to and from functions, ensuring they remain unaltered.
Hashable	Tuples are hashable, which means they can be used as keys in sets or as unique identifiers.

## 1.1 Syntax and Creation

In order to create an empty `tuple` we can use the following syntax:

### Example

```
tuple_1 = ()
tuple_2 = tuple()
print(tuple_1, tuple_2)
print(type(tuple_1), type(tuple_2))
```

### Output

```
() ()
<class 'tuple'> <class 'tuple'>
```

## 1.2 Convert iterables into `tuple`

As with lists you can convert other iterable data types to a `tuple` using the `tuple()` constructor.

This feature is particularly useful when you need the immutability of `tuples`, or when a specific API requires a `tuple` instead of another iterable, or you want to have a representation of some data in this format.

### Example

```
# Converting a list to a tuple
my_list = [1, 2, 3]
my_tuple = tuple(my_list)
print(my_tuple)
# Converting a string to a tuple
my_string = "hello"
string_tuple = tuple(my_string)
print(string_tuple)
```

### Output

```
(1, 2, 3)

('h', 'e', 'l', 'l', 'o')
```

You can create a `tuple` just assigning a few variables into with coma and parentheses.

### Example

```
another_tuple = 2, 'world', 1.618
print("Another tuple:", another_tuple)
print("Type:", type(another_tuple))
```

### Output

```
Another tuple: (2, 'world', 1.618)
Type: <class 'tuple'>
```

## 1.3 Immutability

Once a `tuple` is created, its elements **cannot be changed, removed, or added**. This immutability makes `tuples` a reliable data structure for storing *unchangeable* data.

**IMPORTANT:** Though, you have to understand that if the `tuple` has mutable objects inside, you can access and modify them there.

### Example

```
tuple_1 = ([1,2,3], [5,6])
print("Tuple before:", tuple_1)
print("ID before:", id(tuple_1))
# Modifying a mutable object inside the tuple
tuple_1[-1].append(4)
print("Tuple after:", tuple_1)
print("ID after:", id(tuple_1))
```

### Output

```
Tuple before: ([1, 2, 3], [5, 6])
ID before: 139653455025920
Tuple after: ([1, 2, 3], [5, 6, 4])
ID after: 139653455025920
```

Yes, it's Python baby, but I promise, you will get used to this once understand the concept of objects and how do they work.

Note that `id` of the `tuple` hasn't changed, but `id` of the objects inside has.



## 2. Features Overview

Same as

lists Python supports indexing, slicing, concatenation, multiplication, unpacking and some built-in functions.

Suppose we have the following two tuples:

```
first_tuple = (1, 'hello', 3.14)
second_tuple = 2, 'world', 1.618
```

### 2.1 Indexing and Slicing in Tuples

#### Example

```
# Indexing
print(first_tuple[1])
# Slicing
print(second_tuple[1:])
```

#### Output

```
hello
('world', 1.618)
```

### 2.2 Concatenation and Multiplication

#### Example

```
# Concatenation
combined_tuple = first_tuple + second_tuple
# Multiplication
repeated_tuple = first_tuple * 2
print("Combined:", combined_tuple)
print("Repeated:", repeated_tuple)
```

#### Output

```
Combined: (1, 'hello', 3.14, 2, 'world', 1.618)
```

```
Repeated: (1, 'hello', 3.14, 1, 'hello', 3.14)
```

## 2.3 Unpacking

### Example

```
x, y, z = first_tuple
print(f"x={x},y={y},z={z}")
```

### Output

```
x=1,y=hello,z=3.14
```

## 2.4 Operator in

### Example

```
if 'hello' in first_tuple:
    print("Found 'hello' in the tuple!")
```

### Output

```
Found 'hello' in the tuple!
```

## 2.5 Functions

### Example

```
constants = (3.14, 2.7, 36.6)
length = len(constants)
max_value = max(constants)
min_value = min(constants)
sum_value = sum(constants)
sorted_list = sorted(constants)
print("Lenght:", length)
print("Max:", max_value)
print("Min:", min_value)
print("Sum:", sum_value)
print("Sorted:", sorted_list)      # Note that ``sorted()`` returns a ``list``
instead!
```

### Output

```
Lenght: 3
Max: 36.6
```

```
Min: 2.7
Sum: 42.44
Sorted: [2.7, 3.14, 36.6]
```

Basically, it's very similar to `lists`, but don't forget that Python doesn't allow modification of immutable objects.

## 2.6 Tuple Methods

There are only two methods for `tuples`.

```
nums = (1, 2, 3)
```

Method	Description	Example	Output
<code>index(x)</code>	Returns the index of the first item whose value is <code>x</code> .	<code>nums.index(3)</code>	2
<code>count(iterable)</code>	Returns the number of times <code>x</code> appears in the tuple.	<code>nums.count(1)</code>	1

## 3. Iterations

Typically we use tuples for storing the *constants*.

### 3.1 Using `for in`

#### Example

```
# Rainbow is a constant so that ``tuple`` is a great choice to store its colors
rainbow = ('red', 'orange', 'yellow', 'green', 'blue', 'indigo', 'violet')
for color in rainbow:
    print(color)
```

#### Output

```
red
orange
yellow
green
blue
indigo
violet
```

### 3.2 `for index in range(len(tuple))`

```
# Tuple of prime numbers
primes = (2, 3, 5, 7, 11)
for index in range(len(primes)):
    print(f"Prime number at index {index} is {primes[index]}")
```

## Output

```
Prime number at index 0 is 2
Prime number at index 1 is 3
Prime number at index 2 is 5
Prime number at index 3 is 7
Prime number at index 4 is 11
```

### 3.3 Using enumerate()

There is a built-in function called `enumerate()`, that adds a counter to an iterable.

It can be used with **any iterable**. and be particularly useful with when **both the element and its index** are needed.

```
# Tuple of weekdays
weekdays = ('Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday')
for index, day in enumerate(weekdays):
    print(f"{day} is the {index+1}-th day of the workweek.")
Monday is the 1-th day of the workweek.
Tuesday is the 2-th day of the workweek.
Wednesday is the 3-th day of the workweek.
Thursday is the 4-th day of the workweek.
Friday is the 5-th day of the workweek.
```

## 4. Quiz

### Question 1:

What will be the output of the following code?

```
my_tuple = (10, 20, 30, 40, 50)

print(my_tuple[-2])
```

---

### Question 2:

What happens when you try to modify an element in a tuple?

```
my_tuple = (1, 2, 3, 4, 5)
```

```
my_tuple[0] = 6
```

---

### Question 3:

Which of the following is the correct way to create a tuple?

A) `my_tuple = tuple(1, 2, 3)`

B) `my_tuple = 1, 2, 3`

C) `my_tuple = (1,)`

D) `my_tuple = [1, 2, 3]`

---

### Question 4:

What does the `count()` method do in a tuple?

A) Counts the number of occurrences of an element in the tuple.

B) Counts the total number of elements in the tuple.

C) Returns the first occurrence of an element in the tuple.

D) None of the above.

---

### Question 5:

What is the output of the following tuple slicing?

```
my_tuple = (0, 1, 2, 3, 4, 5)
```

```
print(my_tuple[1:4])
```

A) `(1, 2)`

B) `(1, 2, 3)`

C) `(2, 3, 4)`

D) `(0, 1, 2)`

---

### Question 6:

How does `enumerate()` enhance the iteration process in tuples?

- A) It sorts the `tuple` before iteration.
  - B) It adds a counter as part of the `tuple` element during iteration.
  - C) It reverses the `tuple` during iteration.
  - D) It performs unpacking of the `tuple` elements.
- 

### Question 7:

Which of the following is true about `tuple` concatenation?

- A) It modifies the original `tuple`.
  - B) It creates a new `tuple`.
  - C) It's not possible to concatenate `tuples`.
  - D) It removes duplicates while concatenating.
- 

### Question 8:

Which of the following statements about `tuples` is NOT true?

- A) `Tuples` are immutable.
- B) `Tuples` can contain elements of different data types.
- C) `Tuples` do not support slicing.
- D) `Tuples` can be used as keys in dictionaries.

## 5. Homework

### Task 1: Tuple Analyzer

**Objective:** Write a program that allows the user to input a sequence of numbers, store them in a `tuple`, and analyze the `tuple` to provide *insights*.

```
Input: Enter numbers separated by commas: 1, 2, 3, 4, 5
Output: Tuple: (1, 2, 3, 4, 5)
        Sum: 15
        Average: 3
        Maximum: 5
        Minimum: 1
```

### Task 2: Tuple Sorter

**Objective:** Develop a program to sort elements of multiple `tuples` based on user preference (ascending or descending order).

```
Input: Enter elements of the tuple: 5, 1, 9, 3
```

```
Sort order (asc/desc): asc
Output: Sorted Tuple: (1, 3, 5, 9)
```

### Task 3: Tuple Element Finder

**Objective:** Create a program that finds specific elements within a `tuple` based on user queries.

```
Input: Tuple: (1, 2, 3, 4, 5, 6)
       Enter element to search: 4
Output: Element 4 found at index: 3
```

### Task 4: Enumerate Sports Teams

**Objective:** Use the `enumerate()` function to list sports teams and their ranking based on user input.

```
Input: Enter teams (separated by commas): Lakers, Bulls, Celtics
Output: Team Rankings:
        1. Lakers
        2. Bulls
        3. Celtics
```

### Task 5: Mini Code Review

**Objective:** Try rewriting some programs from the previous homework where it's applicable and where `tuples` are a better choice.

# Lesson 8: Data Structures p.3 (set)

Sets are the guardians of uniqueness, ensuring each element stands out. Be like a set, stand out from the crowd!"

## 1. Introduction to Sets

A set - is a **mutable, unordered collection of items in Python**.

Every element **is unique (no duplicates)**. and **must be immutable (cannot be changed)**.

### 1.1 Syntax

Sets can be created using curly braces {} or the set() function. However, an empty set can only be created using the set() function, using {} as it creates an empty dictionary.

#### Example

```
my_set = {1, 2, 3}
print(my_set)
print("Type is:", type(my_set))
empty_set = set()
print(empty_set)
print("Type of empty set is:", type(empty_set))
dictionary = {}          # Be careful it is NOT a ``set``
print(dictionary)
print("Type is:", type(dictionary))
```

#### Output

```
{1, 2, 3}
Type is: <class 'set'>
set()
Type of empty set is: <class 'set'>
{}
Type is: <class 'dict'>
```

### 1.2 Convert iterables into sets

Converting other iterables to sets can be useful for several reasons:

- **Eliminating Duplicates:** Sets automatically remove duplicate elements, which is useful for getting *unique items*.
- **Efficient Membership Testing:** Checking whether an item exists in a set is faster compared to lists or tuples.



As for me, it is very underestimated data structure among `py` devs' community, because based on my current experience I haven't seen many solutions where sets were used in production code.

The syntax for converting an iterable to a set involves the `set()` constructor.

### Example

```
# Converting a list to a set
my_list = [1, 2, 3, 2, 1]
my_set = set(my_list)
print(my_set)
# Converting a tuple to a set
my_tuple = (4, 5, 6, 5, 4)
my_set = set(my_tuple)
print(my_set)
# Converting a string to a set
my_string = "hello"
my_set = set(my_string)
print(my_set)
```

### Output

```
{1, 2, 3}
{4, 5, 6}
{'e', 'h', 'l', 'o'}
```

Bear in mind that the order of elements in sets can be different from time to time you output the elements.

## 1.3 Indexing and Slicing

Sets **do NOT** support indexing or slicing.

This sounds logically, as there is no order in how elements are stored and they can't be accessed by index.

### Example

```
my_set = {1, 2, 3}
# The following lines will raise an error
print(my_set[0])
print(my_set[0:2])
```

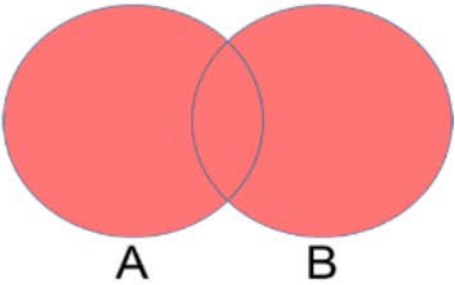
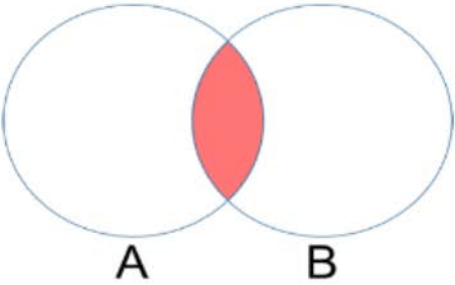
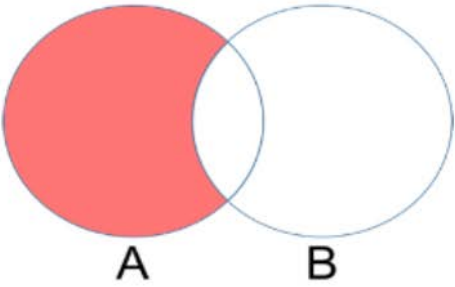
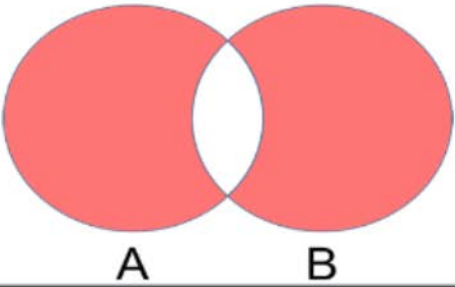
## Output

```
TypeError: 'set' object is not subscriptable
```

### 1.4 Math operations and `built_in` functions.

You can do various math operations with `sets` in Python.

Python's built-in functions like `len()`, `max()`, `min()`, `sum()`, and `sorted()` can also be applied to sets.

Set Operation	Venn Diagram	Interpretation
Union		$A \cup B$ , is the set of all values that are a member of $A$ , or $B$ , or both.
Intersection		$A \cap B$ , is the set of all values that are members of both $A$ and $B$ .
Difference		$A \setminus B$ , is the set of all values of $A$ that are not members of $B$
Symmetric Difference		$A \triangle B$ , is the set of all values which are in one of the sets, but not both

### Example

```
# Define two sets
A = {1, 2, 3}
B = {3, 4, 5}
# Perform set operations
union = A | B # or A.union(B)
intersection = A & B # or A.intersection(B)
difference = A - B # or A.difference(B)
symmetric_difference = A ^ B # or A.symmetric_difference(B)
# Display the results
print("Union:", union)
```

```

print("Intersection:", intersection)
print("Difference:", difference)
print("Symmetric Difference:", symmetric_difference)
# Apply built-in functions
length = len(A)
max_value = max(A)
min_value = min(A)
sum_value = sum(A)
sorted_list = sorted(A) # Note: sorted() returns a list
# Display the results
print("Length:", length)
print("Max:", max_value)
print("Min:", min_value)
print("Sum:", sum_value)
print("Sorted:", sorted_list)
# Membership test
print("-" * 10)
print(1 in A)
print("A" in A)

```

## Output

```

Union: {1, 2, 3, 4, 5}
Intersection: {3}
Difference: {1, 2}
Symmetric Difference: {1, 2, 4, 5}
Length: 3
Max: 3
Min: 1
Sum: 6
Sorted: [1, 2, 3]
-----
True
False

```

## 1.5 Set Comparisons

Set comparison operators in Python enable you to compare sets based on their contents, allowing you to check for subset, superset, or equality relationships. These operations are particularly useful when dealing with collections of elements where the focus is on the presence or absence of items rather than their order or frequency.

### 1.5.1 Subset and Superset

- A **subset** (<) is a set where all its elements are contained within another set.
- A **superset** (>) is a set that contains all elements of another set.

Python provides the `<`, `<=`, `>`, and `>=` operators, along with the methods `.issubset()`, `.issuperset()` for these comparisons.

## 1.5.2 Equality

- Two sets are considered **equal** (`==`) if they contain the same elements, regardless of their order.

### Example

```
A = {1, 2, 3}
B = {1, 2, 3, 4, 5}
C = {1, 2, 3}
# Checking for subset
is_subset = A < B # or A.issubset(B)
print(f"A is a subset of B: {is_subset}")
# Checking for superset
is_superset = B > A # or B.issuperset(A)
print(f"B is a superset of A: {is_superset}")
# Checking for equality
is_equal = A == C
print(f"A is equal to C: {is_equal}")
# Using <= and >= for subset and superset including equality
is_subset_or_equal = A <= C
is_superset_or_equal = B >= C
print(f"A is a subset or equal to C: {is_subset_or_equal}")
print(f"B is a superset or equal to C: {is_superset_or_equal}")
# Non-equivalence
is_not_equal = B != C
print(f"B is not equal to C: {is_not_equal}")
```

### Output

```
A is a subset of B: True
B is a superset of A: True
A is equal to C: True
A is a subset or equal to C: True
B is a superset or equal to C: True
B is not equal to C: True
```

## 2. Methods of `sets`

Python sets come with methods that allow you to modify set elements and compare sets in various ways.

### 2.1 Adding and Removing Elements

Method	Description
--------	-------------

Method	Description
<code>add(element)</code>	Adds an element to the set.
<code>remove(element)</code>	Removes an element from the set. Raises a <code>KeyError</code> if the element is not present.
<code>discard(element)</code>	Removes an element from the set if it is a member. If the element is not a member, does nothing.
<code>pop()</code>	Removes and returns an arbitrary element from the set. Raises a <code>KeyError</code> if the set is empty.
<code>clear()</code>	Removes all elements from the set.

## 2.2 Set Operations

Method	Description
<code>union(*others)</code>	Returns a new set with elements from the set and all others.
<code>intersection(*others)</code>	Returns a new set with elements common to the set and all others.
<code>difference(*others)</code>	Returns a new set with elements in the set that are not in the others.
<code>symmetric_difference(other)</code>	Returns a new set with elements in either the set or other but not both.
<code>update(*others)</code>	Updates the set, adding elements from all others.
<code>intersection_update(*others)</code>	Updates the set, keeping only elements found in it and all others.
<code>difference_update(*others)</code>	Updates the set, removing elements found in others.
<code>symmetric_difference_update(other)</code>	Updates the set, keeping only elements found in either set, but not in both.

### Example

```
A = {1, 2, 3}
B = {3, 4, 5}
# Add and remove elements
A.add(4)
A.discard(2)
```

```
print(f"After add and discard: {A}")
# Set operations
union_set = A.union(B)
intersection_set = A.intersection(B)
difference_set = A.difference(B)
print(f"Union: {union_set}")
print(f"Intersection: {intersection_set}")
print(f"Difference: {difference_set}")
```

## Output

```
After add and discard: {1, 3, 4}
Union: {1, 3, 4, 5}
Intersection: {3, 4}
Difference: {1}
```

## 3. set Comprehensions

Just like for list comprehensions, there is a support for set comprehensions as well.

### Example

```
# Create a set of squares using set comprehension
squares = {x**2 for x in range(10)}
print(squares)
# Create a set from a list where only even numbers are stored
evens = {x for x in range(10) if x % 2 == 0}
print(evens)
```

## Output

```
{0, 1, 64, 4, 36, 9, 16, 49, 81, 25}

{0, 2, 4, 6, 8}
```

## 4. frozenset() Overview

A frozenset is the immutable version of a set. Once created, items cannot be added or removed from a frozenset.

This immutability makes `frozenset` instances hashable, so they can be used as keys in dictionaries or as elements in other sets.

### Example

```
immutable_set = frozenset([1, 2, 3])
print(immutable_set)
# Attempt to add an element (This will raise an error)
try:
    immutable_set.add(4)
except AttributeError as e:
    print(e)
```

### Output

```
frozenset({1, 2, 3})

'frozenset' object has no attribute 'add'
```

## 5. Quiz

### Question 1:

What method would you use to add an element to a set in Python?

- A) `push()`
  - B) `append()`
  - C) `add()`
  - D) `insert()`
- 

### Question 2:

Which set method is used to remove an element that might not be present in the set?

- A) `remove()`
  - B) `discard()`
  - C) `pop()`
  - D) `delete()`
- 

### Question 3:

What will be the result of the following code snippet?



- A) `{2, 3}`
  - B) `{1, 2, 3, 4}`
  - C) `{1, 4}`
  - D) `TypeError`
- 

#### Question 4:

Which of the following is true for the `frozenset` type?

- A) It is mutable.
  - B) It can contain mutable items.
  - C) It can be used as a dictionary key.
  - D) Elements can be added to it using `add()`.
- 

#### Question 5:

What does the `union()` method do when applied to two sets, A and B?

- A) Finds elements that are in both A and B.
  - B) Creates a new set with elements from both A and B, excluding duplicates.
  - C) Removes from A all elements that are also in B.
  - D) Creates a new set with elements that are unique to each set.
- 

#### Question 6:

How would you remove all elements from a set A in Python?

- A) `A.delete()`
  - B) `A.clear()`
  - C) `A.removeAll()`
  - D) `A.discardAll()`
- 

#### Question 7:

Given `A = {1, 2, 3}` and `B = {3, 4, 5}`, what is the result of `A - B`?

- A) {1, 2}
  - B) {3}
  - C) {4, 5}
  - D) {1, 2, 4, 5}
- 

### Question 8:

Which of the following will create a set containing the elements 1, 2, and 3?

- A) `set(1, 2, 3)`
- B) `set([1, 2, 3])`
- C) `{[1, 2, 3]}`
- D) `{1: 'a', 2: 'b', 3: 'c'}`

## 6. Homework

### Task 1: Methods

#### Objective:

1. Create two sets, `A` and `B`, where `A` contains numbers from 1 to 10 and `B` contains numbers from 5 to 15.
2. Perform the following operations and print the results:
3. Find the union of `A` and `B`.
4. Find the intersection of `A` and `B`.
5. Find the difference between `A` and `B`.
6. Find the symmetric difference between `A` and `B`.
7. Add a new element 16 to set `B` and remove element 5.
8. Check if `A` is a subset of `B` and vice versa.

### Task 2: More Methods

#### Objective:

1. Create three sets, `x`, `y`, and `k`, with random numbers and some common elements among them.
2. Update set `x` with the intersection of `x`, `y`, and `k`.
3. Perform a symmetric difference update between set `y` and set `k`.
4. Check if `x` is now a subset of `y` or `k` and print the result.
5. Convert set `k` into an immutable set and attempt to add another element to demonstrate the immutable property.

### Task 3: Guess a Letter

**Objective:** Create a game where players guess letters of a secret word, using sets to track guessed letters and determine the game's progress.

```
## Example
```

```
Input: Secret Word: "banana", Guessed Letters: {'a', 'n'}, Guess: 'b'
```

```
Output: Correct
```

```
Input: Secret Word: "apple", Guessed Letters: {'a', 'e', 'p'}, Guess: 'c'
```

```
Output: Incorrect
```

# Lesson 9: Data Structures p.4 (`dict`)

Can't come up with a quote :) But dicts rules, yeah!

## 1. Introduction to `dict`

In Python, a `dictionary` is a mutable, unordered collection of items. While other compound data types have only value as an element, a dictionary has a key-value pair.

Dictionaries are optimized to retrieve values when the key is known. Consider them as a mini database for your application.

### 1.1 Syntax

Creating a dictionary is as simple as placing items inside curly braces `{ }` separated by commas or using `dict()` function. An item has a key and a corresponding value that is expressed as a pair (key: value).

#### Example

```
# Option 1: Explicitly define a ``dict`` object
my_dict = {
    'name': 'John',
    'age': 30,
    'occupation': 'developer'
}
# Option 2: Pass keyword parameters
my_dict = dict(name='John', age=30, occupation='developer')
```

#### Output

```
{'name': 'John', 'age': 30, 'occupation': 'developer'}
```

Or you can create an empty dictionary:

#### Example

```
my_dict = {}
my_dict = dict()
print(my_dict)
```

## Output

```
{}
```

## 1.2 Accessing and Modifying Dictionary

Accessing elements can be done by referring to its key name, enclosed in square brackets `[]` or using the `.get()` method. Adding or modifying elements can be done by using the assignment operator (`=`) along with the key in square brackets.

### Example

```
my_dict = {
    'name': 'John',
    'age': 30,
    'occupation': 'developer'
}
# Access
print(my_dict['name'])
print(my_dict.get('age')) # 30
# Modify
my_dict['age'] = 31
# Add new key-value
my_dict['hobby'] = 'painting'
print(my_dict)
```

## Output

```
John
30
{'name': 'John', 'age': 31, 'occupation': 'developer', 'hobby': 'painting'}
```

## 2. Functions and Methods of `dict`

### 2.1 `min()` and `max()`

The `min()` and `max()` functions can be used with dictionaries to find the minimum or maximum key or value.

**Note:** By default, these functions operate on the dictionary keys, but you can specify to operate on the values by using the `key` parameter.

## Example

Finding minimum and maximum keys

```
my_dict = {'apple': 3, 'banana': 2, 'cherry': 5}
print(min(my_dict)) # Output: 'apple'
print(max(my_dict)) # Output: 'cherry'
```

## Output

apple

cherry

## Example

Finding minimum and maximum values, instead of keys.

```
my_dict = {'apple': 3, 'banana': 2, 'cherry': 5}
print(min(my_dict.values()))
print(max(my_dict.values()))
```

## Output

2

5

## 2.2 sorted()

The `sorted()` function returns a new sorted list from the items in any iterable. When used with dictionaries, you can sort by keys or values.

## Example

```
my_dict = {'apple': 3, 'banana': 2, 'cherry': 5}
sorted_keys = sorted(my_dict)
print(sorted_keys)
```

**IMPORTANT:** Output - is a list: ['apple', 'banana', 'cherry']

## Output

['apple', 'banana', 'cherry']

## 2.3 Methods of Dictionary

Method	Description	Exaple Code	Example Output
<code>keys()</code>	Returns a view object containing the keys of the dictionary.	<code>my_dict.keys()</code>	<code>dict_keys(['name', 'age', 'occupation'])</code>
<code>values()</code>	Returns a view object containing the values of the dictionary.	<code>my_dict.values()</code>	<code>dict_values(['John', 30, 'developer'])</code>
<code>items()</code>	Returns a view object containing a tuple for each key-value pair.	<code>my_dict.items()</code>	<code>dict_items([('name', 'John'), ('age', 30), ('occupation', 'developer')])</code>
<code>update()</code>	Updates the dictionary with the elements from another dictionary object or from an iterable of key/value pairs.	<code>my_dict.update({'hobby': 'painting'})</code>	<code>{'name': 'John', 'age': 30, 'occupation': 'developer', 'hobby': 'painting'}</code>
<code>pop()</code>	Removes a specified key and returns the corresponding value.	<code>removed_age = my_dict.pop('age')</code>	Removes 'age' from <code>my_dict</code> and returns 30
<code>clear()</code>	Removes all items from the dictionary.	<code>my_dict.clear()</code>	<code>my_dict</code> becomes <code>{}</code>

## 3. Iterations

Iterating over dictionaries can be done in several ways, such as by iterating over keys, values, or both.

### 3.1 Iterating Over Keys

Iterating over keys is the default behavior when iterating through a dictionary directly in a for loop.

#### Example

```
for key in my_dict:
    print(f"Key: {key}")
```

## Output

```
Key: name  
Key: age  
Key: occupation
```

### 3.2 Iterating Over Values

If you're only interested in values, use the `.values()` method.

#### Example

```
for value in my_dict.values():  
    print(f"Value: {value}")
```

## Output

```
Value: John  
Value: 30  
Value: developer
```

### 3.3 Iterating Over Items

To get both keys and values simultaneously, use the `.items()` method. This method is particularly useful when you need to work with both elements.

#### Example

```
for key, value in my_dict.items():  
    print(f"Key: {key}, Value: {value}")
```

## Output

```
Key: name, Value: John  
Key: age, Value: 30  
Key: occupation, Value: developer
```

### 3.4 Using Dictionary Comprehensions

Dictionary comprehensions are not just for creating dictionaries; they can also be used to iterate over an existing dictionary.



## Example

Filter out items where the value is below a certain threshold. This is useful for data processing where you only want to keep items that meet certain criteria.

```
my_dict = {'apple': 1, 'banana': 2, 'cherry': 3, 'date': 4}
filtered_dict = {key: value for key, value in my_dict.items() if value > 2}
print(filtered_dict)
```

## Output:

```
{'cherry': 3, 'date': 4}
```

## Example

Inverting the keys and values of a dictionary can be useful in scenarios where you need to look up keys by their associated values.

```
my_dict = {'apple': 1, 'banana': 2, 'cherry': 3, 'date': 4}
inverted_dict = {value: key for key, value in my_dict.items()}
print(inverted_dict)
```

## Output:

```
{1: 'apple', 2: 'banana', 3: 'cherry', 4: 'date'}
```

## 3.5 Iterating with `enumerate()`

While `enumerate()` is typically used with lists, it can also be applied to the `.items()` of a dictionary to get both the index and the key-value pairs during iteration. Not sure, where it can be applicable, but just FYI that such option exists.

## Example

```
for index, (key, value) in enumerate(my_dict.items()):
    print(f"Index: {index}, Key: {key}, Value: {value}")
```

## Output

```
Index: 0, Key: name, Value: John
Index: 1, Key: age, Value: 30
Index: 2, Key: occupation, Value: developer
```

## 4 Dictionary Equality

### 4.1 Equality (==)

Two dictionaries are considered equal if:

- They have the same set of keys.
- Corresponding keys have the same values.

Python checks each key-value pair during comparison, making it an efficient process.

### Example

```
dict_a = {'name': 'Alice', 'age': 25, 'city': 'Wonderland'}
dict_b = {'age': 25, 'city': 'Wonderland', 'name': 'Alice'}
print(dict_a == dict_b)
```

## Output

```
True
```

**NOTE:** Although `dict_a` and `dict_b` were defined in different orders, they are considered equal.

**NOTE:** Order does not matter since Python 3.7, dictionaries maintain the insertion order of items. However, the order does not influence the outcome of equality comparisons.

### 4.2 Inequality (!=)

The inequality operator returns `True` if the dictionaries differ in at least one key or value.

### Example

```
dict_a = {'name': 'Alice', 'age': 25}
dict_b = {'name': 'Alice', 'age': 30}
# The values for the key 'age' are different, so dict_a and dict_b are not
equal.
```

```
print(dict_a != dict_b)
```

## Output

```
True
```

**Note:** Deep comparison, the comparison is "deep" meaning nested dictionaries will also be compared accurately.

## 5. Nested Dictionaries

Nested dictionaries allow you to store and organize complex data structures.

### 5.1 Accessing Elements

To access elements within a nested dictionary, you chain square brackets `[]` or use the `.get()` method for safer access.

#### Example

```
family = {  
    'john': {'age': 30, 'job': 'developer'},  
    'jane': {'age': 28, 'job': 'designer'}  
}  
print(family['john']['job'])
```

## Output

```
developer
```

```
28
```

### 5.2 Modifying Elements

Specify the keys to the path of the item you want to modify.

#### Example

```
# Modifying Jane's job  
family['jane']['job'] = 'architect'  
print(family['jane']['job'])
```

## Output

```
architect
```

### 5.3 Adding Values

Adding a new key-value pair to a nested dictionary might require ensuring that the parent dictionary exists.

This can be done using the `.setdefault()` method or checking for the existence of the key.

#### Example

```
# Adding a new key-value pair for John
family['john']['hobby'] = 'painting'
# Adding a new nested dictionary for a new family member
family['alice'] = {'age': 24, 'job': 'engineer'}
print(family)
```

## Output

```
{'john': {'age': 30, 'job': 'developer', 'hobby': 'painting'},
 'jane': {'age': 28, 'job': 'designer'},
 'alice': {'age': 24, 'job': 'engineer'}}
}
```

### 5.4 Iterations

To access keys and values, you might iterate over the outer dictionary and then over each nested dictionary.

#### Example

```
for person, details in family.items():
    print(f"Name: {person}")
    for key, value in details.items():
        print(f"    {key.capitalize()}: {value}")
```

## Output

```
Name: john
    Age: 30
    Job: developer
```

```
Name: jane
Age: 28
Job: designer
```

## 5.5 Deleting Items

To remove items either from default or a nested dictionary, use the `del` statement or the `.pop()`

### Example

```
# Deleting a key-value pair
del family['john']['hobby']
# Removing an entire nested dictionary
removed_person = family.pop('alice', None)
print(f"Removed: {removed_person}")
```

### Output

```
{'john': {'age': 30, 'job': 'developer'}, 'jane': {'age': 28, 'job': 'designer'}}
Removed: {'age': 28, 'job': 'designer'}
```

## 6. Quiz

### Question 1:

What does the `pop()` method do in a Python dictionary?

- A) Adds a new item to the dictionary
- B) Returns the value of a key and removes the key-value pair from the dictionary
- C) Sorts the dictionary
- D) None of the above

---

### Question 2:

How do you access the value associated with the key 'occupation' in the dictionary `my_dict`?

- A) `my_dict(occupation)`
- B) `my_dict['occupation']`
- C) `my_dict.get(occupation)`
- D) `my_dict.get('occupation')`

### Question 3:

Which of the following statements about dictionaries in Python is true?

- A) Dictionaries are ordered collections of items.
  - B) A dictionary's keys can be mutable types.
  - C) Dictionaries can contain mixed data types for keys and values.
  - D) A dictionary cannot contain another dictionary as a value.
- 

### Question 4:

How would you create a new dictionary that contains all the items from `dict1` and `dict2`, where items in `dict2` overwrite items in `dict1` for matching keys?

- A) `dict1.update(dict2)`
  - B) `{**dict1, **dict2}`
  - C) `dict1 + dict2`
  - D) `dict2.update(dict1)`
- 

### Question 5:

Which method would you use to safely retrieve a value from a dictionary, providing a default value if the key does not exist?

- A) `fetch()`
- B) `get()`
- C) `retrieve()`
- D) `pull()`

## 7. Homework

### Task 1 Movie Recommendation System

**Objective:** Create a simple movie recommendation system that suggests movies to users based on their preferences.

#### Requirements:

1. **Movie Database:** Create a dictionary named `movie_database` where keys are movie titles and values are lists containing the genre(s) of each movie.
2. **Recommendation Algorithm:** Suggest a movie to a user based on their preferred genres and the movie database.

The format should be something like this:

```
movie_database = {  
    "Interstellar": ["Adventure", "Drama", "Sci-Fi"]  
}
```

## Task 2: Quiz Game

**Objective:** Create a flashcard quiz game that helps users learn and test their knowledge on various topics.

### Requirements:

1. **Data:** Define a dictionary named `flashcards` where keys are questions and values are answers.
2. **Quiz:** Implement a quiz functionality that presents users with a random flashcard question and prompts them to input their answer.
3. **Repeat:** Allow the user to continue the quiz until they decide to quit.

```
flashcards = {  
    "What is the chemical simbol for water?": "H2O",  
}  
  
# We haven't learnt modules yet, but you will need to figure out how to get the  
random question from ``flashcards``, I believe in you!
```

# Lesson 10: Functions

"Functionality is the heart of programming, functions are the veins."

## 1. Introduction to Functions

### 1.1 What is a function?

A function in programming is a **self-contained, reusable block of code** which acts as a mini-program within a larger program.

Functions allow you to segment your code into **modular, manageable** pieces, thereby enhancing **readability**, simplifying **debugging** and improving **coding experience** overall.

### 1.2 Real world examples

Let's say we have a complex repetitive task, baking cakes. And let's say that in order to bake a singular cake we have to run this code:

```
print("1. Preheat the oven to 350°")
print("2. Mix flour, sugar, and eggs.")
print("3. Bake for 30 minutes.")
print("4. Let the cake cool and serve.")
```

So if we had to bake 4 cakes in different times, and we had to do it *without* using functions, our code would look like this:

```
print("1. Preheat the oven to 350°")
print("2. Mix flour, sugar, and eggs.")
print("3. Bake for 30 minutes.")
print("4. Let the cake cool and serve.")
print("1. Preheat the oven to 350°")
print("2. Mix flour, sugar, and eggs.")
print("3. Bake for 30 minutes.")
print("4. Let the cake cool and serve.")
print("1. Preheat the oven to 350°")
print("2. Mix flour, sugar, and eggs.")
print("3. Bake for 30 minutes.")
print("4. Let the cake cool and serve.")
print("1. Preheat the oven to 350°")
print("2. Mix flour, sugar, and eggs.")
print("3. Bake for 30 minutes.")
print("4. Let the cake cool and serve.")
```



Let's write the same code, but **with functions** this time.

## 1.3 Syntax

In Python functions are defined by `def` keyword *followed by the name* of the function , then curly brackets `()` and semicolon `:`.

**NOTE:** Take a look at *indentation*, in case it's wrong the Python interpreter will not be able to compile the code.

### Example

```
def print_cake_recipe():
    print("1. Preheat the oven to 350°")
    print("2. Mix flour, sugar, and eggs.")
    print("3. Bake for 30 minutes.")
    print("4. Let the cake cool and serve.")
    print("\n")
print_cake_recipe() # we call the function in order to execute code inside it
print_cake_recipe()
print_cake_recipe()
print_cake_recipe()
```

### Explanation

We created a function `print_cake_recipe()` and then called it 4 times, lets see what happens when we run it.

### Output

```
1. Preheat the oven to 350°
2. Mix flour, sugar, and eggs.
3. Bake for 30 minutes.
4. Let the cake cool and serve.
1. Preheat the oven to 350°
2. Mix flour, sugar, and eggs.
3. Bake for 30 minutes.
4. Let the cake cool and serve.
1. Preheat the oven to 350°
2. Mix flour, sugar, and eggs.
3. Bake for 30 minutes.
4. Let the cake cool and serve.
1. Preheat the oven to 350°
2. Mix flour, sugar, and eggs.
3. Bake for 30 minutes.
4. Let the cake cool and serve.
```

Now you can see that it is a *very convenient* way to write the programs, **as you can collect chunks of your code into functions** and **call them every time you need**.

**NOTE:** If we try to assign function value to a variable, it will be `None`, more on this in "10.5 Return".

## 2. Parameters and Arguments

Often you need to work with dynamic values within your function and the optimal approach to this challenge would be implementing *parameters*.

### 2.1 Syntax

**Objective:** Write an `addition` function that would take two values and print the sum of these two values. Here is an example of how this code would look like:

#### Example

```
def addition(a, b):  
    print(a + b)  
addition(1, 3)
```

#### Output

```
4
```

#### Explanation

Define function `addition` and in curly brackets we declared the parameters: `a` and `b`.

Then we called the function and passed arguments `1` and `3`, as `a` and `b` accordingly

### 2.2 Parameters vs Arguments

As irrelevant as it might seem, there is a difference between these two key terms. The function **parameters are the names listed in the function's definition** and the function **arguments are the real values passed to the function**.

You just need to understand that once you pass value `11` to the parameter `a` into the function, it becomes an argument.

## 3. Positional vs Key Arguments

When calling functions in Python, the arguments you pass can be either **positional** or **keyword** arguments. **Understanding the difference and the proper usage of each type is crucial for writing clear and error-free code.**

### 3.1 Positional Arguments

Positional arguments are arguments that need to be included in the correct order. The order in which you pass the values when calling the function should match the order in which the parameters were defined in the function.

#### Example

```
def create_profile(name, age, profession):  
    print(f"Name: {name}, Age: {age}, Profession: {profession}")  
create_profile("Alice", 30, "Engineer")
```

#### Output

```
Name: Alice, Age: 30, Profession: Engineer
```

#### Explanation

In this example, "Alice" is passed as the name, 30 as the age, and "Engineer" as the profession, **following the order they were defined in the function.**

We did it sequentially, so that params are in the same order. Never mix the order of your arguments, it can blow out your code!

### 3.2 Keyword Arguments

**Keyword arguments**, on the other hand, **are arguments passed to a function, accompanied by an identifier.**

You explicitly state which parameter you're passing the argument to by using the name of the parameter. **This means the order of the arguments does not matter**, as long as all required parameters are provided.

#### Example

```
def create_profile(name, age, profession):  
    print(f"Name: {name}, Age: {age}, Profession: {profession}")
```

```
create_profile(age=30, profession="Engineer", name="Alice")
```

## Output

```
Name: Alice, Age: 30, Profession: Engineer
```

## Explanation

Here, even though the order of arguments is **different** from the order of parameters in the function definition, Python knows which argument corresponds to which parameter, thanks to the keyword parameters.

Personally I prefer the following approach, following it, your codebase becomes much cleaner.

## 4. Scopes

The scope of a variable refers to **the context in which it is visible or accessible in the code**. In Python, scopes help manage and isolate variables in different parts of the program, ensuring that variable names don't clash and create unexpected behaviors.

The two main types of scopes are **local** and **global**.

### 4.1 Local scope

Variables declared **within** a function have **local** scope, they are created when function is called and destroyed when it finishes its execution. **Local variables declared in a functions can only be accessed from within this function.**

## Example

```
def addition():  
    suma = 10 + 15  
    print(suma)  
addition()
```

## Output

```
25
```

## Example

```
def addition():
```

```
summ = 10 + 15
print(summ)
print(summ)
```

## Output

```
NameError: name 'summ' is not defined.
```

## Explanation

In this case we declare variable `suma` and since we created it **inside** the function it has **local** scope, therefore we wouldn't be able to call it from outside:

## 4.2 Global scope

Variables declared **outside** all the functions have a global scope. They can be **accessed** from any part of the code, **including** inside functions, **unless overshadowed by a local variable with the same name**.

## Example

```
total = 50
def print_added_total():
    print(total + 50)
print_added_total()
```

## Output

```
100
```

## Explanation

Here, we declared variable **outside** the function, therefore it has global scope and can be *accessed* and *modified* from *wherever* one might need.

However, if we attempt to **reassign** a different value to the global variable from within the function the program will not work as intended.

## Example

```
total = 50
def update_total():
```

```
total += 50
update_total()
```

## Output

```
UnboundLocalError: local variable 'total' referenced before assignment
```

This happens because you **can't reassign** value to a variable, unless you use keyword `global`, however it is strongly advised not to use it, as this will introduce redundant complexity to your code, making it less clear and more bug prone.

Here it is important to understand the difference between reassigning a value and modifying an object as if we attempt to append an element to a list declared with global scope it won't cause any problems.

## Example

```
list1 = ["a", "b"]
def update_list():
    list1.append("c")
update_list()
print(list1)
```

## Output

```
['a', 'b', 'c']
```

## 4.3 Best practices

1. **Use Few Global Variables:** Try to use global variables (those outside functions) as little as possible to avoid confusion.
2. **Keep Variables Local:** Use variables inside functions for things that only matter in that function. This helps keep your code clean and easy to understand.
3. **Be Careful with Same Names:** If you use the same name for a variable inside and outside a function, the function **will only know about the inside one**.

## 5. Return

The `return` statement in a function **sends a value from within the function's local scope to where the function has been called**. It's a powerful way to pass data *out of a function* and can be used to send **any type** of object back to the caller.

## 5.1 Syntax

The `return` statement is followed by the **value** or **expression** you want to return. **If no value or expression is specified, the function will return `None`.**

### Example

```
def addition(a, b):  
    return a + b  
sum_of_two_numbers = addition(1, 3)  
print(sum_of_two_numbers)
```

### Output

```
4
```

In this example, the `addition` function takes two positional arguments `a` and `b` and returns their sum returning the value to `sum_of_two_numbers` variable.

**Note:** Any operation after the `return` statement will not be executed.

### Example

```
def addition(a, b):  
    return a + b  
    print("this message will never be output")  
sum_of_two_numbers = addition(1, 3)  
print(sum_of_two_numbers)
```

### Output

```
4
```

## 6. Optional Parameters

In Python functions **can be called with a varying number of parameters**. This feature enhances **flexibility** and **usability** of the code depending on the scenario. Optional parameters have **default values**, which are used if **no argument** is passed during the function call.

## 6.1 Benefits of Optional Parameters

Optional parameters make your functions more **flexible**. They allow you to create more generalized functions that can handle a wider range of inputs.

Thanks to optional parameters you will be able to use the same function for slightly different purposes **without** overloading it with arguments or creating multiple, nearly identical functions.

## 6.2 Syntax

To define an optional parameter, you **assign it a default value** in the function's definition using operator =. This default value is used if the caller does not provide a value for that parameter.

### Example

```
def greet(name, message="Hello"):
    print(f"{message}, {name}!")
greet("Alice")
greet("Bob", "Good morning")
```

## Output

Hello, Alice!

Good morning, Bob!

### Explanation

In the `greet` function above, `name` is a **mandatory** parameter, while `message` is **optional** with a default value of `"Hello"`. If no `message` is provided when the function is called, it uses the default value.

**Important:** You can only assign default values to parameters **After** you have declared your positional arguments, more on this in **args/kwargs** section

## 7. Args and Kwargs

In Python, `*args` and `**kwargs` are **special operators** used in function definitions. They allow a function to accept a variable number of arguments, making your functions more flexible. `*args` is used for **positional** arguments, while `**kwargs` is used for **keyword** arguments.

## 7.1 \*args

The `*args` parameter allows a function to take **any number** of **positional** arguments without having to define each one individually.



**Note** The arguments passed to `*args` are accessible as a **tuple**. Therefore, contents of `args` **can not** and **should not** be altered in traditional ways.

### Example

```
def calculate_sum(*numbers):  
    total = sum(numbers)  
    return total  
print(calculate_sum(10, 20, 30)) # Output: 60
```

### Explanation

In this example, `calculate_sum()` can take **any number** of numerical arguments, sum them up, and return the total. The `*numbers` parameter collects all the positional arguments into a tuple called `numbers`.

#### 7.1.1 Unpacking parameters

As we learned previously in data types, you can use `*` to unpack the elements of an iterable, it is relevant as if you will try to pass an iterable to a function without unpacking it, it will be treated as a whole object.

**NOTE:** Same applies for `**kwargs`

### Example

```
list1 = ["a", "b", "c"]  
def example(*args):  
    for argument in args:  
        print(argument)  
  
example(list1)
```

### Output

```
['a', 'b', 'c']
```

Therefore, if you want to pass all the elements of `list1` we will need to unpack them first.

### Example

```
list1 = ["a", "b", "c"]  
def example(*args):  
    for argument in args:  
        print(argument)
```

```
example(*list1)
```

## Output

```
a  
b  
c
```

## 7.2 \*\*kwargs

The `**kwargs` parameter allows a function to accept any number of keyword arguments. This is useful when you want to handle named arguments in your function. The keyword arguments passed to `**kwargs` are stored in a **dictionary**.

### Example

```
def student_info(**details):  
    for key, value in details.items():  
        print(f"{key}: {value}")  
student_info(name="John", grade="A", subject="Mathematics")
```

### Output:

```
name: John  
grade: A  
subject: Mathematics
```

### Explanation

In this example, `student_info()` can accept **any number** of keyword arguments. The `**details` parameter collects all the keyword arguments into a dictionary called `details`.

## 8. Argument Ordering

When defining a function, **it's important to follow the correct order of parameters to avoid syntax errors**. The order should be:

- Standard arguments
- `*args`
- `**kwargs`

This order ensures that your function can handle a mix of standard, positional, and keyword arguments effectively.

### Example of Correct Syntax

```
def mix_and_match(a, b, *args, **kwargs):  
    pass # Function implementation
```

### Example

Incorrect Syntax - Will Cause an Error!

```
def mix_and_match(a, b, **kwargs, *args):  
    pass
```

### Output:

```
SyntaxError: invalid syntax
```

## 9. Quiz

### Question 1:

What will the output be for the following function call?

```
def greet(name):  
    return "Hello, " + name  
print(greet("Alice"))
```

- A) Hello, Alice
- B) Hello,
- C) Alice
- D) It will raise an error.

---

### Question 2:

Given this coffee-making function, what will the following function call output?

```
def make_coffee(size="Medium", type="Cappuccino"):
    return f"Making a {size} {type} coffee."
print(make_coffee("Large"))
```

- A) Making a Large Cappuccino coffee.
  - B) Making a Large coffee.
  - C) Making a Medium Cappuccino coffee.
  - D) It will raise an error.
- 

### Question 3:

What does the `*args` parameter in a function allow you to do?

- A) It allows the function to accept any number of keyword arguments.
  - B) It allows the function to accept a list of arguments.
  - C) It allows the function to accept any number of positional arguments.
  - D) It unpacks the arguments passed to the function.
- 

### Question 4:

What will be the output of the following code?

```
def user_profile(**details):
    return details.get("name", "Anonymous") + " - " + details.get("role", "Guest")
print(user_profile(name="John", age=30, role="Admin"))
```

- A) John - 30
  - B) John - Admin
  - C) Anonymous - Guest
  - D) It will raise an error.
- 

### Question 5:

Consider this function. What is true about the `return` statement in this function?

```
def add_numbers(a, b):
```

```
result = a + b
return result
print("Calculation has been completed")
```

- A) It outputs the result of the function.
  - B) It stops the function's execution and returns the result.
  - C) It prints the result before ending the function.
  - D) It is optional and can be omitted.
- 

### Question 6:

Given this code, what will the output be?

```
def calculate_difference(a, b):
    result = a - b
    return result
calculate_difference(10, 5)
print(result)
```

- A) 5
  - B) 10
  - C) result
  - D) It will raise a `NameError`.
- 

### Question 7:

Consider the function and its call below. What will the output be?

```
def display_info(name, age):
    return f"Name: {name}, Age: {age}"
print(display_info(age=25, name="Emma"))
```

- A) Name: Emma, Age: 25
- B) Name: 25, Age: Emma
- C) Name: name, Age: age
- D) It will raise an error.

---

### Question 8:

In the function definition below, which parameters are considered optional?

```
def func(a, b=5, c=10):  
    return
```

- A) Only a
- B) Both b and c
- C) Only b
- D) All a, b, and c

---

### Question 9:

Which of the following is the correct way to define a function with all types of arguments?

- A) `def func(*args, a, b, **kwargs):`
- B) `def func(a, *args, b, **kwargs):`
- C) `def func(a, b, *args, **kwargs):`
- D) `def func(**kwargs, *args, a, b):`

---

### Question 10:

Given the function and call below, what will the function return?

```
def multiply_numbers(*args):  
    result = 1  
    for number in args:  
        result *= number  
    return result  
print(multiply_numbers(2, 3, 4))
```

- A) 24
- B) 9
- C) `6`
- D) It will raise an error.

## Question 11:

What will be the output of the following code?

```
x = 10
def print_number():
    x = 5
    print("Inside function:", x)
print_number()
print("Outside function:", x)
```

- A) Inside function: 5, Outside function: 5
- B) Inside function: 10, Outside function: 10
- C) Inside function: 5, Outside function: 10
- D) It will raise a `NameError`.

## 10. Homework

### Task 1: Star Rectangle

**Objective:** Implement a function named `draw_rectangle` that outputs a rectangle made of asterisks (\*). The rectangle should have a width of 7 characters and a height of 6 lines.

#### Requirements:

- Use nested `for` loops to generate the rectangle.
- The outer loop should iterate through the lines (height), and the inner loop should iterate through the characters (width) on each line.
- Only the border of the rectangle should be drawn with asterisks, while spaces ( ) should fill the interior.
- The function does not need to return anything; it should directly print the rectangle to the console.

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

### Task 2: Sum of Digits

**Objective:** Create a function `print_digit_sum` that calculates and prints the sum of all digits in a given integer.

#### Requirements:

- The function should accept a single integer argument, possibly negative.
- Convert the integer to its absolute value to handle negative numbers.
- Iterate over each digit in the number and calculate the total sum.
- Print the result to the console. The function returns `None`.

```
print_digit_sum(1234) # Output: 10
print_digit_sum(-567) # Output: 18
```

### Task 3: Find All Factors

**Objective:** Develop a function `get_factors` that returns a list of all the divisors of a given natural number.

**Requirements:**

- The function should accept a single integer argument, `num`.
- If `num` is less than 1, return an empty list to reflect the definition of natural numbers.
- Efficiently find and return a list of all divisors of `num`.
- Ensure correct functionality for both small and large values of `num`.

```
print(get_factors(28)) # Output: [1, 2, 4, 7, 14, 28]
print(get_factors(13)) # Output: [1, 13]
print(get_factors(0)) # Output: []
```

### Task 4: Temperature Converter

**Objective:** Enhance the `convert_temperature` function to support optional parameters for conversion direction.

**Requirements:**

- The function should accept one mandatory parameter for the temperature value and one optional parameter for the direction of conversion ('C' for Celsius to Fahrenheit, 'F' for Fahrenheit to Celsius).
- Use default arguments to assume conversion from Celsius to Fahrenheit if the direction is not specified.
- Calculate and return the converted temperature value.

```
print(convert_temperature(100)) # Assumes Celsius to Fahrenheit, Output: 212
print(convert_temperature(212, convert_to='C')) # Fahrenheit to Celsius,
Output: 100
```

### Task 5: Statistics Calculator



**Objective:** Implement a function `calculate_statistics` that computes various statistical measures (mean, median, mode, range) for a dataset, based on specified options.

### Requirements:

- The function should accept an arbitrary number of positional arguments (`*data`) representing the dataset.
- Accept keyword arguments (`**options`) to specify which statistics to calculate: mean, median, mode, range. If an option is `True`, calculate that statistic.
- Return a dictionary with keys as the names of the statistics calculated and their corresponding results as values.
- If no options are specified, calculate and return all statistics.
- Handle edge cases such as empty datasets or datasets without a mode.

```
data_points = [4, 1, 2, 2, 3, 5]
print(calculate_statistics(*data_points, mean=True, range=True))
# Output: {'mean': 2.8333333333333335, 'range': 4}
print(calculate_statistics(*data_points))
# Output: {'mean': ..., 'median': ..., 'mode': ..., 'range': ...}
```

## Lesson 11: Exceptions

"Exceptions are the gentle reminders that perfection is a journey, not a destination."

### 1 Introduction

Error handling is an important component of programming that provides opportunity for code to adequately respond to unexpected situations. In Python, errors are managed through the use of exceptions, which are special objects that represent error conditions.

When Python interpreter stumbles upon a situation, it cannot cope with, it raises an exception. If the exception isn't handled, the program will terminate abruptly, which can lead to a poor user experience or even loss of data.

### 2 The `try` & `except` Block

In python, error handling is managed with the use of `try` and `except` keywords.

#### 2.1 How do `try` and `except` work?

The `try` keyword indicates a beginning of a block. Python will first try to run the code inside the block and if an error occurs, it will look for the instructions in `except` block. The error can be specified and the code inside the block will be executed only in case exception was caught.

#### Example

```
try:
    result = 10 / 0 # This will raise a ZeroDivisionError
except ZeroDivisionError:
    print("Cannot divide by zero!")
```

## Output

```
Cannot divide by zero!
```

## Explanation

The code inside the `try` block raises a `ZeroDivisionError`. The `except` block catches this exception and prints a message, without crashing the program.

## 2.2 `except Exception as e`

For debugging purposes, or sometimes even in production, when you aren't sure about the possible errors or you simply want to ignore them you can use `except Exception as e` construction.

## Example

```
try:
    print(3 / 0) # this will raise ZeroDivisionError
except Exception as e:
    print(f"An error occurred: {e}")
```

## Output

```
An error occurred: division by zero
```

You could also use bare `except` block solely for debugging purposes, however it is highly advised against it and simply forbidden in development. It is not a good idea to use this construction because it may lead to bad overall performance of the program.

## Example

```
responses = []
for i in range(0, 9999):
    try:
        response = requests.get("http://example.com")
        responses.append(response)
    except:
```

```
pass
print(len(responses))
```

## Output

```
6785
```

## Explanation

The provided code is using `requests` module, you will learn more about it in the next lesson, for now, all you need to know is that it can be used to retrieve information from the website, in the process called parsing.

You would expect this program to print 9999 as we have tried to access "<http://example.com>" 9999 and should have gotten same amount of responses, however in reality, during the parsing process many things can go wrong, for example:

- Bad internet connection might raise a requests error, passing straight to `except`
- The website access can be limited, meaning that you can only open it 10 times in a minute, and considering that we are trying to parse it, we open it more than 1000 times per minute.
- If the domain you are accessing is changing, it can simply not exist, raising an error once again.

So all of these are possible problems that might have occurred during the execution and because you used bare `except`, you won't be able to identify what specifically went wrong and how to improve the quality of parsing.

## 2.3 Catching Multiple Exceptions

You can catch multiple exceptions by specifying multiple `except` blocks. Each `except` block can handle a different type of exception in a **different** way.

```
try:
    # Some code that might raise different types of exceptions
except ZeroDivisionError:
    # Handle division by zero
except ValueError:
    # Handle value errors
except Exception as e:
    # Handle any other exceptions
    print(f"An error occurred: {e}")
```

## 2.4 Multiple Exception Types

You can also catch multiple exception types in a single `except` block by providing a tuple of exception types.

```
try:
    # Some code that might raise different types of exceptions
except (ZeroDivisionError, ValueError) as e:
    # Handle both ZeroDivisionError and ValueError
    print(f"An error occurred: {e}")
```

## 3 The `else` Block

In Python, the `else` block can be used in conjunction with the `try` and `except` blocks to define a block of code that should only be executed if no exceptions were raised in the `try` block. The `else` block is an optional part of the error-handling mechanism and provides a clear way to separate the normal execution path from the error-handling code.

### Example

```
try:
    result = 10 / 2
except ZeroDivisionError:
    print("Cannot divide by zero!")
else:
    print(f"The result is {result}")
```

### Output

```
The result is 5
```

### Explanation

In this example, the code inside the `try` block successfully completes without raising any exceptions, so the `else` block is executed, and the result is printed.

#### 3.1 When to use `else`

The `else` block is useful when you want to separate the code that might raise an exception from the code that should only be executed if no exceptions occur. This separation can improve the readability of your code and make it easier to understand the flow of execution.

Though, I must admit that the following approach is not really popular among Python society. Have no idea why, to be honest...

#### 3.2 Note

- The `else` block must follow all `except` blocks and will only be executed if the `try` block does not raise an exception.
- If an exception is raised in the `else` block, it will not be caught by the preceding `except` blocks.

## 4 The `finally` Block

In Python, the `finally` block is an important part of exception handling that is used in combination with `try` and `except` blocks. The `finally` block contains code that is guaranteed to execute, regardless of whether an exception is raised in the `try` block or not.

### Example:

```
try:
    print(10 / int(input("input your number:")))
except ZeroDivisionError:
    print("0 is not accepted")
finally:
    print("Thank you for using this program.")
```

### Explanation

In this example, 10 is attempted to be divided by user input in the `try` block. If user inputs 0, a `ZeroDivisionError` is raised and caught in the `except` block. Regardless of whether the mathematical operation was successfully or not, the `finally` block prints the closing message.

## 5 Raising Exceptions

While Python and its libraries raise exceptions automatically under certain conditions, you also have the ability to manually raise exceptions in your code.

This is particularly useful for enforcing constraints, validating input, or signaling that a specific error condition has occurred.

### 5.1 Understanding `raise`

The `raise` statement allows you to throw an exception at any point in your program. When an exception is raised, it interrupts the normal flow of the program and transfers control to the nearest enclosing `try` block.

### 5.2 Syntax of `raise`

The basic syntax to raise an exception is:

```
raise ExceptionType("Description of the error.")
```

Where `ExceptionType` is the class of the exception you want to raise (e.g., `ValueError`, `TypeError`, `KeyError`), and the string provides a description of the error.

## 5.3 Use Cases

**1.Input Validation:** If a function requires inputs to meet certain conditions, you can raise an exception if the provided inputs are invalid.

```
def set_age(age):  
    if age < 0:  
        raise ValueError("Age cannot be negative")  
try:  
    set_age(-5)  
except ValueError as e:  
    print(e)
```

**2.Enforcing Constraints:** When certain states or conditions must not occur within a program, raising exceptions can enforce these constraints explicitly.

```
inventory = {'apple': 10, 'banana': 5}  
def add_to_cart(item, quantity, cart):  
    if item not in inventory:  
        raise KeyError("Item not available")  
    if inventory[item] < quantity:  
        raise ValueError("Insufficient stock")  
    cart[item] = quantity  
    inventory[item] -= quantity
```

**3.Signaling Unimplemented Features:** If a part of your code is not yet implemented, you can raise a `NotImplementedError` as a placeholder.

```
def future_feature():  
    raise NotImplementedError("This feature is coming soon!")  
try:  
    future_feature()  
except NotImplementedError as e:  
    print(e)
```

There are no limits for your imagination in terms of handling errors, again this will come with practice, just think about potential errors which might occur inside the application and try to handle them gracefully!

## 5.5 Best Practices

- **Be Specific:** Prefer raising and catching specific exceptions rather than the general `Exception` class. This makes error handling more predictable, robust and optimised.

- **Provide Useful Messages:** When raising exceptions, include a clear, descriptive message to make it easier to understand the cause of the error.
- **Use Exceptions Judiciously:** While exceptions are powerful, using them inappropriately can make your code harder to understand and maintain. Avoid using exceptions for normal flow control, and prefer using them for actual error conditions.

## 6 Exception Chaining

Exception chaining in Python is a mechanism that allows you to link exceptions together, making it easier to understand a sequence of errors that led to a failure.

This feature is particularly useful when an exception is raised while handling another exception.

### 6.1 Implicit Exception Chaining

Python automatically chains exceptions if an exception is raised inside an `except` block. The original exception is available in the `__context__` attribute of the new exception.

#### Example

```
try:
    # This block intentionally raises a ZeroDivisionError
    result = 1 / 0
except ZeroDivisionError:
    try:
        # This block will raise a NameError
        print(unknown_variable)
    except NameError as e:
        raise RuntimeError("A NameError occurred") from e
```

#### Explanation

In this example, the `NameError` is implicitly chained to the `RuntimeError`. Python will display both exceptions, indicating that the `RuntimeError` was directly raised while handling the `NameError`.

### 6.2 Explicit Exception Chaining with `from`

You can explicitly chain exceptions using the `from` keyword. This allows you to specify the cause of the exception, which can be either another exception instance or `None` to indicate that the chaining should be suppressed.

## Example

```
try:
    # Some operation that fails
    open("nonexistent_file.txt")
except FileNotFoundError as e:
    # Explicitly chaining the exception
    raise ValueError("Failed to open configuration.") from e
```

## Example

In this case, if the file does not exist, a `FileNotFoundError` is raised, and it is explicitly chained to a `ValueError`. When the exception is caught, Python will indicate that the `ValueError` was directly raised from the `FileNotFoundError`.

## 6.3 Suppressing Exception Chaining

You can suppress exception chaining by specifying `from None`. This is useful when the exception context is not helpful or you want to prevent the display of chained exceptions.

### Example of Suppressing Exception Chaining:

```
try:
    some_operation()
except SomeError as e:
    raise DifferentError("An error occurred") from None
```

This prevents Python from chaining the exceptions, so only the `DifferentError` is shown to the user, making the error message cleaner.

## 7. Quiz

### Question 1:

What will the following code output?

```
try:
    print(1 / 0)
except ZeroDivisionError:
    print("You cannot divide by zero!")
```

- A) You cannot divide by zero!
- B) 1



- C) An unhandled exception is thrown
  - D) 0
- 

### Question 2:

Which except clause will catch a `TypeError`?

```
try:
    '2' + 2
except ValueError:
    print("ValueError caught!")
except TypeError:
    print("TypeError caught!")
except:
    print("Some other error caught!")
```

- A) `ValueError`
  - B) `TypeError`
  - C) The generic `except` block
  - D) No `except` block catches the error
- 

### Question 3:

What is the output of the following code snippet?

```
try:
    num = int("3")
except ValueError:
    print("Not a number!")
else:
    print("It is a number!")
```

- A) Not a number!
  - B) It is a number!
  - C) Nothing is printed
  - D) An error is thrown
- 

### Question 4:

What does the `finally` block do?

```
Try:
```

```

x = 1 / 0
except ZeroDivisionError:
    print("Error!")
finally:
    print("Will this be executed? Or error anyway?")

```

- A) It will not execute if an exception is caught.
- B) It is executed only if no exceptions occur.
- C) It executes regardless of whether an exception was caught or not.
- D) It will execute before the `except` block.

---

### Question 5:

What is the purpose of specifying multiple exceptions in a single `except` clause?

```

try:
    # Code that might raise different errors
except (ZeroDivisionError, KeyError) as e:

```

- `print(f"Caught an error: {e}")`
- A) To handle different types of exceptions that might be raised in the same block of code.
  - B) To increase the processing time by handling all errors at once.
  - C) To handle only the first error that occurs and ignore others.
  - D) It's a syntax error to specify multiple exceptions.

### Question 6:

How does exception chaining help in Python?

- A) It suppresses all exceptions.
- B) It links exceptions together, helping to trace back to the initial error.
- C) It prevents exceptions from being raised.
- D) It automatically resolves exceptions without user intervention.

## 8. Homework

### Task 1: Safe Division

**Objective:** Create a function `safe_divide` that safely performs division and handles any division errors gracefully.

#### Requirements:

- The function should accept two parameters, `numerator` and `denominator`.
- Use `try` and `except` blocks to handle division errors such as `ZeroDivisionError`.
- If a division by zero occurs, print an error message and return `None`.
- If the division is successful, return the result.

- Use the `finally` block to print a message that the division attempt has been completed.

```
print(safe_divide(10, 2)) # Output: 5.0
print(safe_divide(5, 0)) # Output: Error: Cannot divide by zero
```

## Task 2: Voting

**Objective:** Implement a function `check_voter_age` that checks if a person is eligible to vote and raises an exception if the age is below the minimum voting age.

### Requirements:

- The function should accept one integer argument, `age`.
- If `age` is less than 18, raise a `ValueError` with a message indicating that the person is too young to vote.
- If `age` is 18 or above, print a message confirming that the person is allowed to vote.
- Use a `try` block to test the function with different ages and an `except` block to catch and print the `ValueError` message.

```
check_voter_age(21) # Output: You are allowed to vote.
check_voter_age(16) # Output: Error: You are too young to vote.
```

# Lesson 12: Imports

"Where Python's Collaborative Symphony Begins."

## 1. Introduction

A Python module is essentially a file that contains definitions and statements.

The module name is the same as the file name with the `.py` extension added at the end.

### 1.1 Why do we need modules?

Generally, if we were to write everything into one file, we wouldn't be able to maintain our application, making it hard for other developers to use.

Using modules helps us to do the following:

- **Organizing Code:** By dividing a program into modules you can separate parts and concerns making it easier to manage and understand the program.
- **Reusable Components:** Functions, classes and variables defined in a module can be easily reused across sections of a program or different programs.
- **Managing Namespaces:** Modules help prevent conflicts between names by providing a separate namespace, for each module.

## 2. Creating a Module

In Python, a simple `.py` file can be considered and used as a module to store specific variables, functions, runnable code or any other object, which can then be used in other modules.

1. **Create a Python File:** Start by creating a Python file. For instance, name it `mymodule.py`.
2. **Add Code to the File:** Write the necessary functions, classes, and variables in the file. Here's an example of what `mymodule.py` might look like:

```
def say_hello(name):  
    print(f"Hello, {name}!")  
def addition(a, b):  
    return a + b  
variable = 123
```

**NOTE:** Module names should conform to Python's naming conventions. They should be short, lowercase, and if necessary, use underscores to improve readability.

## Example

Let's create a mini application stored in `Base/assets/lesson_12/` directory.

We will split our logic having different `.py` files:

```
└─ lesson_12  
    │  
    └─ calculations.py  
    │  
    └─ greetings.py  
    │  
    └─ main.py  
    │  
    └─ menu.py
```

**NOTE:** Include docstrings and comments to explain what your module does, the purpose of different functions, expected parameter types, return types, and so on.

## Example

```
"""  
This modules provides several functions which interact with arithmetical  
operations between 2 numbers  
"""  
  
def add(a, b):  
    """Return the sum of a and b."""  
    return a + b  
  
def subtract(a, b):  
    """Return the difference between a and b."""  
    return a - b  
  
def multiply(a, b):  
    """Return the product of a and b."""  
    return a * b  
  
def divide(a, b):  
    """Return the quotient of a divided by b. Raises ValueError on division by  
zero."""  
    if b == 0:  
        raise ValueError("Cannot divide by zero.")  
    return a / b
```

Again, instead of having everything in a one file, we created a couple of different files where each serves its purpose.

## 2.3 How to?

Break down your application into logical components. Each component should have a well-defined responsibility.

For example, user authentication, data processing, and user interface components can be separate modules.

- **Repeated Code:** If you find yourself copying and pasting the same code across multiple parts of your project, it's a sign that you should modularize this code.
- **Reusable Components:** If your project contains components that could be reused across different parts of the application or even in different projects, these components are good candidates for modularization.

Don't get stuck or overthink how to do it in the best possible way, everything will come with practice.

## 3. `import`, `from`, and `as`

Python provides several ways to bring in external modules into your current script, each serves different needs and providing various ways to access the module's components.

### 3.1 `import`

**Basic Import:** Use the `import` statement to bring an entire module into your script. This way, you access the module's components using the dot notation, which helps in maintaining their namespace.

```
import mymodule
mymodule.say_hello("Alice")  # Calls the say_hello function from mymodule
result = mymodule.addition(10, 20)  # Calls the addition function from mymodule
print(mymodule.variable)  # Access the variable from mymodule
```

### 3.2 `from`

**Import Specific Components:** You can choose to import specific components (functions, classes, or variables) from a module. This is useful when you only need a part of the module.

```
from mymodule import say_hello, addition
say_hello("Bob")  # Directly use the imported function
result = addition(5, 15)  # Directly use the imported function
```

**Note:** Generally, it is a preferable way, as very often we don't need to import everything

**Importing All Components:** Using `from module import *` imports all public names from the module into the current namespace.

```
from mymodule import *
```

**NOTE:** It's generally discouraged as it can lead to conflicts with existing names and makes it unclear which names are present in the current namespace.

### 3.3 Module Aliasing with `as`

**Aliasing Modules:** If a module name is long or likely to conflict with an existing name in your code, you can give it an alias. This allows you to reference it with a different name.

```
import mymodule as mm
mm.say_hello("Carol")  # Use the alias to call a function from mymodule
```

**Aliasing Specific Components:** You can also alias specific components when you import them.

```
from mymodule import say_hello as greet
greet("Dave")  # Use the alias to call the imported function
```

#### 3.3.1 Why Aliasing?

For example, let's say you are trying to import a function `time` from two different modules.

The only way you can achieve this without causing conflicts within the code is by using aliasing.

#### Example

```
from x import time as x_time
from y import time as y_time
```

#### Explanation

Thanks to `as` you can use `time` from both `x` and `y` modules, without any conflicts and be able to distinct them too.

## 4. `if __name__ == "__main__"`

When writing Python scripts, you'll often see a code block guarded by at the bottom:

```
if __name__ == "__main__"
```

The special `__name__` attribute is set to `__main__` when a script is executed directly.

However, when the same script is imported as a module in another script, `__name__` is set to the script's filename.

## Example

Consider the script `mymodule.py`, which defines a function and is executed:

```
# mymodule.py
def say_hello(name):
    """Print a greeting to the named individual."""
    print(f"Hello, {name}!")
def main():
    """Run the main logic of the script."""
    say_hello("Alice")
# This condition checks if the script is the entry point to the program
if __name__ == "__main__":
    main() # Only runs if the script is executed, not if it's imported.
```

## Explanation

If `mymodule.py` is run as the main file, the `main()` function will execute and Alice will be greeted.

If `mymodule.py` is imported into another file, `say_hello` can be used, but Alice will not be greeted until `say_hello` is explicitly called.

Try it yourself and experiment with the code provided in `Base/assets/lesson_12`, and you will see the difference.

## 5. Packages

Packages allow you to group related modules under a common namespace, making it easier to understand and manage the code.

They help in avoiding conflicts between module names and provide a structured way to access the functions, classes, and variables defined in different modules.

### 5.1 `__init__.py`

The `__init__.py` file is a key ingredient in Python packages. It serves multiple roles:



- **Initialization:** Executes any code necessary to set up the package's state or initialize certain variables.
- **Namespace Management:** Declares which modules or symbols the package exports as the API, through `__all__`.
- **Hierarchy Signaling:** Signifies to Python that the directory should be treated as a package, allowing modules to be imported from it.

Instead of raw theory, let's take a look at the real application and try to come up with a good structure for better maintainability of the codebase

## Example

Consider a package named `finance` designed to handle various financial calculations and data processing

## Structure

```
finance/
  __init__.py
  calculator.py
  taxes.py
  investments/
    __init__.py
    stocks.py
    bonds.py
```

finance/`__init__.py`

```
# Specify the public API for convenience
__all__ = ['calculator', 'taxes', 'investments']
# You may also initialize some package-wide constants or setup code, though its
not a common practice nowadays
DEFAULT_TAX_BRACKET = '25%'
```

finance/calculator.py

```
def calculate_interest(principal, rate, time):
    """Compute interest earned on a principal over time at a fixed rate."""
    return principal * (rate / 100) * time
```

finance/taxes.py

```
from .calculator import calculate_interest
def calculate_tax_due(income, rate=DEFAULT_TAX_BRACKET):
    """Calculate the tax owed based on income and a tax rate."""
    pass
```

finance/investments/stocks.py

```
def analyze_stock_market():  
    """Perform analysis on stock market trends."""  
    pass
```

## 5.2 Utilizing Sub-Packages

Sub-packages enable even more granular organization within a package, grouping together related modules.

For instance, `finance/investments` is a sub-package grouping financial investment-related modules like `stocks.py` and `bonds.py`.

When the project becomes extremely big, think about how you could re-structure to make it more maintainable, reusable and readable.

## 5.3 Importing from Packages

Importing from packages or sub-packages is done with a `.dot` notation, allowing you to use specific functions or entire modules.

Examples:

```
# Importing a specific function from a module within the 'finance' package  
from finance.calculator import calculate_interest  
# Importing an entire module from a sub-package  
from finance.investments import stocks  
# Using the imported function  
interest = calculate_interest(1000, 5, 3) # Calculate interest on $1000 at 5%  
for 3 years  
# Invoking a function from the imported 'stocks' module  
stocks.analyze_stock_market()
```

## 6. venv

Python's virtual environments allow you to install and manage Python packages in an isolated environment.

Suppose we need to install a couple of external packages, which provide interfaces to carry out some specific tasks and resolve issues while it can't be done with a default Python functionality.

That's where virtual environments and external packages come in handy.

### 6.1 Benefits of Using `venv`

- **Dependency Management:** Keep project-specific dependencies in separate environments.

- **Project Isolation:** Avoid installing packages globally, which can lead to version conflicts.
- **Reproducibility:** Create an environment that can easily be replicated on other machines or deployments.

The purpose of `venv` is to ensure that each project has its own set of dependencies, which may differ in version from one project to another, without causing conflicts.

**IMPORTANT:** We could install everything globally, but such behaviour leads to conflicts among projects, and is not recommended at all.

## 6.2 Creating a Virtual Environment

You can create a virtual environment using the `venv` module that comes with Python 3.x.

1. **Setup:** Navigate to your project's directory:

```
cd my_project
```

2. **Creation:** Run the following command to create a virtual environment named `venv`:

```
python3 -m venv venv
```

**Note:** You can replace `venv` with your preferred environment name.

3. **Activation:** Run the following commands

- **On Windows:**

```
.\venv\Scripts\activate
```

- **On macOS and Linux:**

```
source venv/bin/activate
```

When activated, the name of the virtual environment (`venv`) will appear in your shell prompt, indicating that all `python` and `pip` commands are now scoped to the `venv`.

**NOTE:** Create a new virtual environment for each project to keep dependencies separate.

## 6.3 Managing Packages with `pip`

With an activated virtual environment, you can use `pip` to install, upgrade, and remove packages. Essentially, `pip` is a package manager which helps you to install the external dependencies from [PyPi](#).

- **Installing Packages:**

```
pip install package_name
```

- **Listing Installed Packages:**

```
pip list
```

- **Uninstalling Packages:**

```
pip uninstall package_name
```

As well, you can run `pip --help` if you want to do anything other with `venv` inside your project.

Let's take a look at the couple of interesting modules in the sections below.

## 7. requests

The `requests` module is a Python HTTP library that simplifies making network requests. It's known for its ease of use and user-friendly interface.

### 7.1 Installation

Before you can use `requests`, you'll need to create `venv`, activate and install `requests` into it.

```
python3 -m venv venv
source venv/bin/activate
pip install requests
```

Let's create an application and split everything into different modules as it was already discussed above.

### 7.2 Weather Fetcher App

**Objective::** Develop an application to display real-time weather information based on user input.

**NOTE:** The application will interact with the `OpenWeatherMap` API to obtain data for the specified city and present it to the user in a clear and concise format.

## Structure

weather\_fetcher/

├─ fetch.py

├─ display.py

├─ main.py

└─ constants.py

constants.py

```
# Store configuration constants
BASE_URL = "http://api.openweathermap.org/data/2.5/weather"
API_KEY = "your_api_key_here" # Replace with your actual OpenWeatherMap API key
```

fetch.py

```
# Contains the logic for fetching weather data from OpenWeatherMap API
import requests
from .constants import BASE_URL, API_KEY # Ensure proper relative import
def get_weather_data(city):
    """Fetch weather data for a given city."""
    params = {
        'q': city,
        'appid': API_KEY,
        'units': 'metric'
    }
    response = requests.get(BASE_URL, params=params)
    if response.status_code == 200:
        return response.json()
    else:
        response.raise_for_status()
```

display.py

```
# Contains the logic for displaying the fetched weather data
```

```
def display_weather(data):
    """Display the weather information."""
    city = data['name']
    temp = data['main']['temp']
    description = data['weather'][0]['description']
    humidity = data['main']['humidity']

    print(f"Weather in {city}:")
    print(f"Temperature: {temp}°C")
    print(f"Condition: {description}")
    print(f"Humidity: {humidity}%")
```

main.py

```
# Entry point for the Weather Fetcher application
from fetch import get_weather_data
from display import display_weather
def main():
    city = input("Enter the name of the city to get the weather for: ")
    try:
        weather_data = get_weather_data(city)
        display_weather(weather_data)
    except requests.exceptions.HTTPError as err:
        print("HTTP error occurred: ", err)
    except requests.exceptions.RequestException as err:
        print("Request error occurred: ", err)
if __name__ == "__main__":
    main()
```

You can find the whole project in Base/assets/weather\_fetcher directory.

## 8. Pillow

Pillow is a fork of the Python Imaging Library (PIL) which adds image processing capabilities to your Python interpreter.

### 8.2 Installation

Install the library in your virtual environment, same as we did with `requests` library:

```
python3 -m venv venv
source venv/bin/activate
pip install Pillow
```

## 8.3 Image Operations App

**Objective:** Develop an application will perform basic operations on an image, such as opening, displaying, rotating, and cropping.

### Structure

```
image_processor/  
├─ processor.py  
└─ main.py
```

processor.py

```
from PIL import Image  
def open_image(path):  
    """Open and return an image."""  
    return Image.open(path)  
def show_image(image):  
    """Display the image."""  
    image.show()  
def rotate_image(image, degrees):  
    """Rotate the image by a certain number of degrees."""  
    return image.rotate(degrees)  
def crop_image(image, box):  
    """Crop the image to a specific box and return the cropped image."""  
    return image.crop(box)
```

main.py

```
from processor import open_image, show_image, rotate_image, crop_image  
def main():  
    image_path = input("Enter the path to the image: ")  
    image = open_image(image_path)  
    show_image(image)  
    rotated = rotate_image(image, 90)  
    show_image(rotated)  
    cropped = crop_image(image, (100, 100, 400, 400))  
    show_image(cropped)  
if __name__ == "__main__":  
    main()
```

You can find the whole project in `Base/assets/image_processor` directory.

Add functionality for saving image and applying filters and effects to it.

## 9. Choose your direction

It's time that you decide your career path and who you want to become in the future. Now you have a general idea about programming and you need to research more information and decide the future direction of your career.

I can recommend a several libraries to look into for Data Science and Web Development.

### Data Science (DS):

- **Pandas:** Data manipulation and analysis.
- **NumPy:** Numerical computing with powerful array objects.
- **Matplotlib & Seaborn:** Data visualization.
- **Scikit-learn:** Machine learning.
- **TensorFlow & PyTorch:** Deep learning.

### Web Development (Backend):

- **Django & Flask:** Web frameworks for building web applications.
- **FastAPI:** Modern, fast web framework for building APIs.
- **SQLAlchemy:** Database toolkit and ORM.
- **Celery:** Asynchronous task queue/job queue.
- **Redis & Memcached:** Caching systems to enhance performance.

Thanks for attention! Happy Restructuring!

## 10. Homework

During this homework I would like you to google. It's an extremely important skill for a programmer, which we use on a daily basis.

**Disclaimer:** Sometimes people can give you tasks which you don't know how to accomplish, there are a couple of unknown words (libs) below. Good luck!

### Task 1: My Gallery

**Objective:** Build a Python application that generates an HTML image gallery from a folder of image files.

#### Requirements:

- Create a virtual environment and install `Pillow` for image processing.
- Create a module to resize images to thumbnails using `Pillow`.
- Ensure images maintain their aspect ratio.
- Write a module to generate an HTML file that displays all thumbnails as clickable links to the original images.
- Implement a command-line interface where the user can specify the folder containing images and the output directory for the HTML gallery.



## Task 2: CLI Task Manager

**Objective:** Create a command-line interface (CLI) task manager application that allows users to add, list, and delete tasks.

### Requirements:

- Create a virtual environment and install `Pillow` for image processing.
- Enable users to add new tasks, mark tasks as completed, list all tasks, and delete tasks through command-line arguments.
- Use the `argparse` or `click` library to handle command-line arguments for different operations (e.g., `--add`, `--complete`, `--list`, `--delete`).

# Lesson 13: Files

"Files are the gateways to the digital world, where data flows between code and storage."

## 1 What is a File?

A file is a collection of connected data or information stored in a computer memory.

Any type of data, such as text, images, software, or other objects, can be stored in files. Files can be accessed by users.

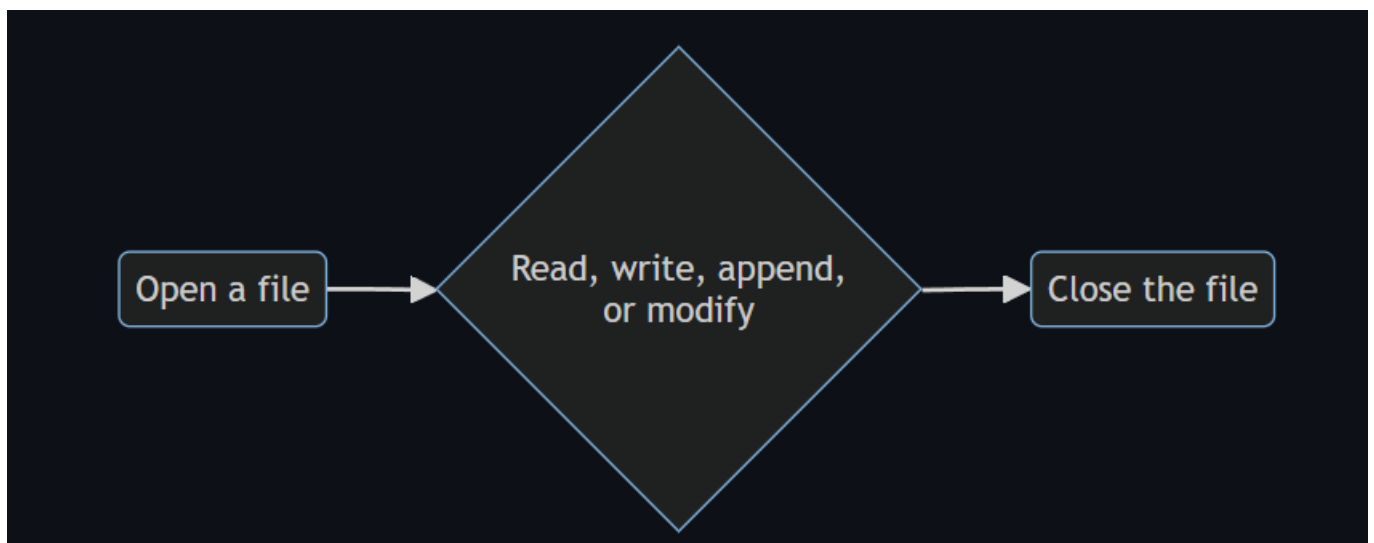
### 1.1 File Types

Text files and binary files are two main types of files. Text files, such as HTML, CSS, and TXT files, are accessible through text editors due to their readable characters. On the other hand, binary files store data in complex formats like images, music, and executable programs.

## 2. Working with Files

### 2.1 Introduction

In order to interact with a file in Python we must **first open** it, **once** the file is **opened** we you can **perform** a desired **operation**, **after** the **editing** of the file is finished file **must be closed**, **otherwise the contents of a file might get corrupted or lost**. Therefore order of actions when working with files is as follows:



### 2.2 Open Files in Python

In order to open file in python use `open()` function, which has two attributes: the file name and the mode in which it will be opened.

## Example

```
file = open("file_name", "mode")
```

The available modes are:

Mode	Description
r	To read a file, open it. (default)
w	To write, open a file. if the file doesn't already exist, creates a new one; otherwise, truncates the existing one.
a	Access a file without truncating it so that you can append data to the end of it. if the file does not already exist, creates it.
t	Switch to text mode. (default)
x	For exclusive creation, open a file. The operation fails if the file already exists.
b	In binary mode, open.
+	To update a file, open it (reading and writing)

You have to close file after you finish working with it to do that, use `close()` method.

## Syntax:

```
file.close()
```

## 2.3 Reading a File

To read contents of a file you can use `read()` method, it returns text of the file as a string.

## Example

```
file = open("Base/assets/lesson_12/menu.py", "r")
file_text = file.read()
print(file_text)
file.close()
```

## Output

```
# menu.py
def display_menu():
```

```
"""Print the main menu options for the application."""
print("Please choose an option:")
print("1. Say Hello")
print("2. Perform a Calculation")
print("3. Exit")
```

In python there is method `readlines()`, it can be used to read contents of a file as well as `read`, but instead of returning one string, it returns a list of small ones (lines of text in the document).

### Example

```
file = open("Base/assets/lesson_12/menu.py", "r")
lines = file.readlines()
for line in lines:
    print(line)
file.close()
```

### Output

```
# menu.py
def display_menu():
    """Print the main menu options for the application."""
    print("Please choose an option:")
    print("1. Say Hello")
    print("2. Perform a Calculation")
    print("3. Exit")
```

## 2.4 Writing to a File

Now you know how to read contents of a file, but if you want to edit it? For this purpose, in python exists method `write()` it writes a string value to a file.

```
output_file = open("output_file", "w")    # if file doesn't exist yet, the
program will simply create it
output_file.write("text")
output_file.close()
```

### Output

Check the content of the file, there should be `text` string written inside.

```
text
```

If we want to write multiple lines to a file, we can use the `writelines()` method.

```
file = open("output_file", "w")
lines = ["line 1\n", "line 2\n", "line 3\n"]
file.writelines(lines)
file.close()
```

### Output(output\_file)

```
line 1
line 2
line 3
```

## 2.5 Renaming and Deleting a File

To rename a file in Python, since there is no built-in way to do that in python, we need 'os' library, particularly the `os.rename()` method.

```
import os
os.rename("old_file_name", "new_file_name")
```

To delete a file in Python use the `os.remove()` method.

```
import os
os.remove("file_name")
```

## 2.6 Real Example

To demonstrate how all of this can be useful in real world programming, consider the following example:

**Objective:** The program that counts every word in a text file and outputs counts into another one:

```
# Open the file(input) and read the data
input_file = open('input.txt', 'r')
file_text = input_file.read().replace(', ', ' ').replace('.', ' ')
input_file.close()
# Count the repetitions of each word in the file
word_count = {}
words = file_text.split(' ')
for word in words:
    if word not in word_count:
        word_count[word] = 1
```

```
    else:
        word_count[word] += 1
# Open the output file and write the word_count dict in it
output_file = open('output.txt', 'w')
output_file.write(str(word_count))
output_file.close()
```

Create a file `input.txt` and paste any random text inside, I used the following one:

## Example

The sun was shining bright on a beautiful day, with birds chirping in the trees and a light breeze blowing through the

air. John walked down the street, whistling a tune and admiring the colorful flowers in the gardens. He stopped at the

corner store to buy a newspaper, then continued on his way. As he walked, he thought about his upcoming vacation to the

beach, and how he couldn't wait to relax in the sun and go swimming in the ocean. Suddenly, he heard a loud crash and

turned to see a car accident on the road ahead. He rushed over to help and called 911 for assistance. The emergency

responders arrived quickly, and John continued on his way, feeling grateful that he was able to make a difference in

someone's time of need.

## Output

Check `output.txt` file it should have been updated.

```
{'The': 2, 'sun': 2, 'was': 2, 'shining': 1, 'bright': 1, 'on': 4, 'a': 7, 'beautiful': 1, 'day': 1, 'with': 1, 'birds': 1, 'chirping': 1, 'in': 5, 'the': 10, 'trees': 1, 'and': 7, 'light': 1, 'breeze': 1, 'blowing': 1, 'through': 1, 'air': 1, 'John': 2, 'walked': 2, 'down': 1, 'street': 1, 'whistling': 1, 'tune': 1, 'admiring': 1, ...}
```

## 2.7 File Methods

Method	Description	Example	Output
<code>read()</code>	Reads the entire contents of the file.	<code>file.read()</code>	"Hello, world!\nThis is a test file."
<code>readline()</code>	Reads a single line from the file.	<code>file.readline()</code>	"Hello, world!\n"
<code>readlines()</code>	Reads all the lines of the file and returns them as a list.	<code>file.readlines()</code>	["Hello, world!\n", "This is a test file."]
<code>write()</code>	Writes a string to the file.	<code>file.write("Hello, Python.")</code>	- (modifies file content)
<code>writelines() )</code>	Writes a list of strings to the file.	<code>file.writelines(["Hello" , "World"])</code>	- (modifies file content)
<code>seek()</code>	Changes the position of the file pointer.	<code>file.seek(0)</code>	- (moves file pointer to the beginning)
<code>tell()</code>	Returns the current position of the file pointer.	<code>pos = file.tell()</code>	10 (depends on the current position)
<code>flush()</code>	Flushes the write buffer of the file.	<code>file.flush()</code>	- (ensures data is written)
<code>truncate()</code>	Truncates the file to a specified size.	<code>file.truncate(100)</code>	- (changes file size to 100 bytes)

## 3. Exception Handling

It's crucial to manage exceptions while working with files in Python.

The `FileNotFoundError`, `PermissionError`, and `IOError` exceptions are the most common when interacting with files. You can use a try-except block to handle file errors.

### Example

```
try:
    file = open("filename.txt", "r")
    contents = file.read()
    file.close()
except FileNotFoundError:
    print("File not found.")
```

### Output

```
File not found.
```

## 4 Full/Relative Paths

When interacting with files, you may define the file's location using a full or absolute path.

The whole route to a file, starting from the root directory, is known as an absolute path.

### Example

```
Base/6.Lists.md --> This is a relative path
```

The path to a file relative to the active working directory is known as a complete path. You may use the `os` module to obtain the current working directory.

If you clone this project, the it will be the following

### Example

```
/home/<username>/review_book/Base
```

**Note:** It may differ depends on which OS you are using.



```
import os
cwd = os.getcwd()
```

Generally, it is preferable to use the `relative` path instead of the `full`, as if you another person runs your project, they will have a different full path, and the application might crash.

## 5. Context Manager `with`

`with open` is a context manager in Python that offers a more effective and secure approach to handle resources like file streams.

It is advised that you use it while opening and working with files in Python.

The `with` statement is employed to surround a block of code's execution with methods specified by a context manager.

The `open()` method serves as the context manager while processing files.

The syntax for using `with open` is as follows:

```
with open("file_name", "mode") as file:
    # code to work with file goes here
```

When you use `with open`, Python automatically closes the file after the block of code included in the `with` statement completes running—even if an exception does.

```
with open("file_name", "r") as file:
    contents = file.read()
    print(contents)
```

This is the best approach to follow working with files. So that it can help to prevent unnecessary errors which may occur.

### Example

As an example for this topic we could rewrite the previous program that was counting words in a text using `with open` statement:

```
# Open the file and read the data
with open('input.txt', 'r') as input_file:
    data = input_file.read().replace(',', ' ').replace('.', ' ')
# Count the occurrences of each word
word_count = {}
words = data.split(' ')
```

```

for word in words:
    if word in word_count:
        word_count[word] += 1
    else:
        word_count[word] = 1
# Open the output file and write the word counts
with open('output.txt', 'w') as output_file:
    output_file.write(str(word_count))

```

## Output:

We will have the same output as we had before. But the approach towards working with files now has changed and became much safer.

```

{'The': 2, 'sun': 2, 'was': 2, 'shining': 1, 'bright': 1, 'on': 4, 'a': 7,
'beautiful': 1, 'day': 1, 'with': 1,
'birds': 1, 'chirping': 1, 'in': 5, 'the': 10, 'trees': 1, 'and': 7, 'light':
1, 'breeze': 1, 'blowing': 1,
'through': 1, 'air': 1, 'John': 2, 'walked': 2, 'down': 1, 'street': 1,
'whistling': 1, 'tune': 1, 'admiring': 1...

```

## 6 Working with Files of Different Formats

Python provides several libraries for working with files of different formats. Here are some of the most commonly used libraries:

- CSV - csv module
- JSON - json module
- XML - xml.etree.ElementTree module
- YAML - pyyaml module
- Pickle - pickle module

### CSV

Here is an example of how to read a CSV file using the csv module:

```

import csv
with open("file.csv", "r") as file:
    reader = csv.reader(file)
    for row in reader:
        print(row)

```

**Input(file.csv)**

```
Country,GDP_per_capita
Luxembourg,110584
Ireland,88588
Switzerland,88224
Cayman Islands,77959
Norway,77544
Singapore,66176
United States,61280
```

## Output

```
['Country', 'GDP_per_capita']
['Luxembourg', '110584']
['Ireland', '88588']
['Switzerland', '88224']
['Cayman Islands', '77959']
['Norway', '77544']
['Singapore', '66176']
['United States', '61280']
```

## JSON

Here is an example of how to read a JSON file using the `json` module:

```
import json
with open("file.json", "r") as file:
    data = json.load(file)
    print(data)
```

### Input(file.json)

```
{
  "name": "John Doe",
  "age": 35,
  "email": "johndoe@example.com",
  "address": {
    "street": "123 Main St",
    "city": "Anytown",
    "state": "CA",
    "zip": "12345"
  }
}
```

## Output

```
{'name': 'John Doe', 'age': 35, 'email': 'johndoe@example.com', 'address':  
{ 'street': '123 Main St', 'city': 'Anytown', 'state': 'CA', 'zip': '12345' }}
```

## XML

Here is an example of how to read an XML file using the `xml.etree.ElementTree` module:

```
import xml.etree.ElementTree as ET  
tree = ET.parse("file.xml")  
root = tree.getroot()  
for child in root:  
    print(child.tag, child.attrib)
```

### Input(file.xml)

```
<root>  
  <person>  
    <name>John Doe</name>  
    <age>35</age>  
    <email>johndoe@example.com</email>  
    <address>  
      <street>123 Main St</street>  
      <city>Anytown</city>  
      <state>CA</state>  
      <zip>12345</zip>  
    </address>  
  </person>  
</root>
```

### Output

```
person {}
```

## YAML

Here is an example of how to read a YAML file using the `pyyaml` module:

```
import yaml  
with open("file.yaml", "r") as file:  
    data = yaml.load(file, Loader=yaml.FullLoader)
```

```
print(data)
```

## Input(file.yaml)

```
- name: John Doe
  age: 35
  email: johndoe@example.com
  address:
    street: 123 Main St
    city: Anytown
    state: CA
    zip: '12345'
```

## Output

```
[{'name': 'John Doe', 'age': 35, 'email': 'johndoe@example.com', 'address':
{'street': '123 Main St', 'city': 'Anytown', 'state': 'CA', 'zip': '12345'}}]
```

## Pickle

Pickle is a specifically designed for Python file type, which allows us to store or save Python objects in memory. It is particularly useful in machine learning as it allows to store and transfer heavy ai models in regular memory instead of operative.

## Example

```
import pickle
# Write a Python object to a file
data = {"key": "value"}
with open("file.pkl", "wb") as file:
    pickle.dump(data, file)
# Read a Python object from a file
with open("file.pkl", "rb") as file:
    data = pickle.load(file)
    print(data)
```

## 7. Quiz

### Question 1:

What does the `open()` function do in Python?

- A) Opens a website URL in Python
- B) Opens a file in a specified mode
- C) Opens a new Python installation window
- D) Creates a new Python object

### Question 2:

What is the output of using the `readlines()` method on a file object?

- A) A single string containing all lines of the file
- B) A list of strings, each representing one line of the file
- C) The first line of the file
- D) The last line of the file

### Question 3:

Which file mode allows you to append to the end of an existing file without truncating it?

- A) 'r+'
- B) 'w'
- C) 'a'
- D) 'x'

### Question 4:

What does the `with` statement provide when working with file operations?

- A) A way to compact the code into a single line
- B) Error checking mechanisms
- C) Automatic closure of the file
- D) Improved file read speeds

### Question 5:

What exception is raised if a file operation is attempted on a non-existent file without the appropriate handling?

- A) `FileNotFoundError`
- B) `IOError`
- C) `OSError`
- D) `ValueError`

## 8. Homework

### Task 1: Exception Safe File Reader

**Objective:** Create a function `safe_file_reader` that reads contents from a given file and handles any possible exceptions.

**Requirements:**

- The function should accept a file path as an argument.
- Use a `try-except` block to handle `FileNotFoundError` and print a friendly message if the file is not found.
- Use the `with` statement to ensure the file is properly closed after reading.
- Print the contents of the file if it is found and successfully opened.

**Task 2: File Content Reverser**

**Objective:** Write a function `reverse_file_content` that reads a file, reverses its content, and writes it back to the same file.

**Requirements:**

- The function should take a file path as an argument.
- Read the original content of the file and reverse the order of lines.
- Write the reversed content back to the same file.
- Ensure all file operations are done within a `with` block to handle the file resource properly.

**Task 3: Batch File Renamer**

**Objective:** Create a script `batch_rename` that renames all files in a directory by appending `"_old"` to their original names.

**Requirements:**

- Use the `os` module to list and rename files.
- Ensure the script checks if the directory exists and handle any exceptions.
- Only rename files (not directories).
- Print the old and new names of the files as they are renamed.

For additional functionality and tasks, try to use new modules and libs from the previous lesson, be creative, as it's high time you start developing your own applications!

# Lesson 14: Functional Programming

"Here should be the same quote with veins as in Lesson 10"

## 1. Revision of Functions

Let's revisit the key concepts covered in [Lesson 10](#) about functions in Python:

1. **Introduction to Functions:** Functions are self-contained blocks of code designed to perform a specific task. They help make your code modular, manageable, and reusable, enhancing readability and simplifying debugging.
2. **Parameters vs Arguments:**
  - **Parameters** are the variables listed in the function's definition.
  - **Arguments** are the actual values passed to the function when it is called.
3. **Positional vs Key Arguments:**
  - **Positional Arguments** must be passed in the order the parameters were defined.
  - **Keyword Arguments** are named when passed, allowing them to be in any order.
4. **Scopes:**
  - **Local Variables** can only be accessed within their function.
  - **Global Variables** can be accessed anywhere in the code.
5. **Return Statements:**
  - The `return` statement exits a function, optionally passing back a value to the caller. If no expression is specified, `None` is returned.
6. **Optional Parameters:**
  - Functions can have optional parameters with default values, providing default behavior and making them flexible in the number of arguments they accept.
7. **Args and Kwargs:**
  - `*args` allows a function to accept any number of positional arguments.
  - `**kwargs` allows a function to accept any number of keyword arguments.

Always try to split your logic into functions and modules!



## 2. Programming Principles(KIS, DRY, YAGNI)

### 2.1 KISS (Keep It Simple, Stupid)

The KISS principle advocates for simplicity in your code. It's about finding the simplest solution to a problem without unnecessary complexity.

In functional programming, this often means choosing straightforward, readable solutions over clever, convoluted ones.

#### Best Practices

- Write clear and concise functions that do one thing and do it well.
- Avoid unnecessary abstraction layers that can make the code harder to understand.
- Use clear and descriptive names for functions and variables to make your code self-documenting.

#### Example

Avoid doing this, it's a bad solution to put everything into one function, also it violates Single Responsibility Principle which you will get familiar with during the lesson about [SOLID](#).

```
def compute_statistics(numbers):
    total = sum(numbers)
    length = len(numbers)
    mean = (total / length) if length > 0 else float('nan')
    variance = sum((x - mean) ** 2 for x in numbers) / length if length > 0
else float('nan')
    return {"total": total, "length": length, "mean": mean, "variance":
variance}
```

#### Example

Do this instead!

```
def compute_mean(numbers):
    return sum(numbers) / len(numbers) if numbers else float('nan')
def compute_variance(numbers, mean):
    return sum((x - mean) ** 2 for x in numbers) / len(numbers) if numbers else
float('nan')
def compute_statistics(numbers):
    total = sum(numbers)
    length = len(numbers)
    mean = compute_mean(numbers)
    variance = compute_variance(numbers, mean)
    return {"total": total, "length": length, "mean": mean, "variance":
variance}
```

## Output:

```
{"total": 15, "length": 5, "mean": 3.0, "variance": 2.0}
```

## Explanation

The bad code example crams too much logic into a single function, making it hard to read and maintain.

The good code example applies the KISS principle by breaking down the complex function into smaller, more manageable pieces which makes it easier to understand, test, and maintain.

## 2.2 DRY (Don't Repeat Yourself)

The DRY principle is about reducing repetition in your code. It encourages you to abstract common patterns into reusable functions or components.

### Best Practices

- Identify patterns and commonalities in your code and abstract them into separate functions.
- Use higher-order functions to create more generalized and reusable code components.

### Example

Bad approach, you repeat the same operation several times!

```
def print_user_details(user):  
    print(f"Name: {user['name']}")  
    print(f"Age: {user['age']}")  
    print(f"Email: {user['email']}")  
    print(f"Name: {user['name']}")  
    print(f"Membership: {user['membership']}")
```

### Example

Use for loop, simple as it is!

```
def print_user_details(user):  
    for key in user:  
        print(f"{key}: {user[key]}")  
user_details = {  
    "Name": "John Doe",  
    "Age": "30",  
    "Email": "johndoe@example.com",  
    "Membership": "Premium"  
}  
print_user_details(user_details)
```

```
Name: John Doe
Age: 30
Email: johndoe@example.com
Membership: Premium
```

## Explanation

The bad code example unnecessarily repeats the logic for printing user details. The good code example eliminates repetition by using a loop, adhering to the DRY principle.

Moreover, we have made our code more dynamical, in case new keys are added.

## 2.3 YAGNI (You Aren't Gonna Need It)

YAGNI is a reminder to avoid adding unnecessary complexity or functionality to your code.

It suggests that you should not implement something until it is necessary. In practice, this means focusing on what you need right now, not what you might need in the future.

## Best Practices

- Focus on the requirements at hand and implement only the functionalities that are immediately needed.
- Keep your functions and modules focused and lean. The fewer responsibilities they have, the easier they are to manage, test, and reuse.

## Example

Will we have a `future_promotion_plan`?

```
def calculate_discount(price, customer_age, customer_membership,
future_promotion_plan):
    discount = 0
    if customer_age > 65:
        discount += 0.10
    if customer_membership == "Premium":
        discount += 0.15
    if future_promotion_plan:
        discount += get_potential_discount()
    return price * (1 - discount)
```

## Output:

```
85.0
```

## Example

Now we have much cleaner solution comparing to what we have seen before.

```
def calculate_discount(price, customer_age, customer_membership):
    discount = 0
    if customer_age > 65:
        discount += 0.10
    if customer_membership == "Premium":
        discount += 0.15
    return price * (1 - discount)
```

## Output:

85.0

## Explanation

The bad code example includes functionality (`future_promotion_plan`) that isn't required for the current requirements, violating the YAGNI principle.

The good code example removes this unnecessary complexity, focusing on the current needs and keeping the implementation simple and straightforward. This results in cleaner, more maintainable code.

Let's put it altogether!

## Example

```
# Applying KISS, DRY, and YAGNI principles in a restaurant order system
def calculate_item_total(price, quantity):
    return price * quantity
def apply_discount(total, discount_rate):
    return total * (1 - discount_rate) if discount_rate else total
def print_receipt(order_items, discount_rate=None):
    grand_total = 0
    print("Receipt:")
    for item, details in order_items.items():
        item_total = calculate_item_total(details['price'],
        details['quantity'])
        grand_total += item_total
        print(f"{item}:      ${item_total:.2f}      ({details['quantity']} @
        ${details['price']:.2f})")
    if discount_rate:
        print(f"Subtotal: ${grand_total:.2f}")
        grand_total = apply_discount(grand_total, discount_rate)
        print(f"Discount: {discount_rate * 100}%")
    print(f"Grand Total: ${grand_total:.2f}")
def main():
    order_items = {
        'Burger': {'price': 8.50, 'quantity': 2},
        'Fries': {'price': 3.00, 'quantity': 1},
```

```
'Soda': {'price': 1.50, 'quantity': 2}
}
discount_rate = 0.1 # 10% discount
print_receipt(order_items, discount_rate)
main()
```

## Output

```
Receipt:
Burger: $17.00 (2 @ $8.50)
Fries: $3.00 (1 @ $3.00)
Soda: $3.00 (2 @ $1.50)
Subtotal: $23.00
Discount: 10%
Grand Total: $20.70
```

1. **KISS (Keep It Simple, Stupid):** Functions are simple and focused. `calculate_item_total` calculates the total for a single item, and `apply_discount` applies a discount to a total amount.
2. **DRY (Don't Repeat Yourself):** The `print_receipt` function iterates through the items to compute and display the totals, avoiding repetition. The calculations are delegated to specific functions (`calculate_item_total` and `apply_discount`), ensuring that each calculation is defined only once.
3. **YAGNI (You Aren't Gonna Need It):** The system is straightforward, providing only what is necessary for the task at hand. There's no over-engineering or anticipation of future, speculative requirements.

Try modifying some functions you have already created in previous applications following these principles, and you will see, how easier it is to maintain your code!

## 3. Function pointers

Pointer is a piece of code that references or points to a memory location that stores a value or an object.

In Python, unlike in low level programming languages, such as C there are no pointers for regular objects and values. However, there is such functionality with regards to functions. Which is often extremely helpful for your code.

### 3.1 Syntax

In order to create function's pointer all you have to do is assign your function to another variable without parentheses ( ) at the end. You will store a reference (A.K.A pointer) inside this variable.

## Example

```
def addition(a, b):  
    return a + b  
print(addition)  
print(type(addition))  
func = addition  
print(func(1, 2))
```

## Output

```
<function addition at 0x7fdec2d8d800>  
<class 'function'>  
140594728458240  
3
```

## Explanation

If you check the type of `addition` using `type(addition)`, Python tells you that `addition` is of type `function`, which confirms that `addition` is indeed a function object.

And as you see, we can call the `addition` function through `func` variable!

Function pointers have many useful applications, which allow you to improve your code's quality.

## Example

For instance, you could assign function pointers to values of a dict, thereby creating a kind of reusable `if` statement, this kind of dictionary is called handler.

```
def addition(a, b):  
    return a + b  
def subtraction(a, b):  
    return a - b  
func_handler = {  
    "add": addition,  
    "subtract": subtraction  
}  
def main():  
    num1 = int(input("1st number:"))  
    num2 = int(input("2nd number:"))  
    choice = input("choose arithmetical operation:")  
    print("output:", func_handler[choice](a=num1, b=num2))  
main()  
main()
```

## Output

```
1st number:4
2nd number:3
choose arithmetical operation:add
output:7
1st number:4
2nd number:3
choose arithmetical operation:subtract
output:1
```

The technique presented in the example above might seem useless from the first glance, because we can simply write this, right?

```
if choice == "add":
    print(add(num1, num2))
elif choice == "subtract":
    print(subtract(num1, num2))
```

Of course, however it is much more convenient to use handlers in this case, because with many inputs your if statement would grow exponentially. And you'll end up having endless if statements.

Handlers is one of the most favourite patterns I have ever used. Think about where it can be applied within your applications.

## 4. Higher Order Functions

Higher-order functions are functions that accept other functions as their parameters or return functions as their results.

**Note:** Pay attention to parentheses ( ) when using higher-order functions, as they distinguish between referencing the function and calling it.

### 4.1 `map()`

**`map()` Function:** This is a built-in function that applies a specified function to each item of an iterable (such as a list) and returns a map object (an iterable). The beauty of `map()` is that it allows for function application without explicitly writing a loop.

#### Example

```
def square(x):
    """Returns the square of a number."""
    return x * x
numbers = [1, 2, 3, 4, 5]
squared = map(square, numbers)
print(list(squared))
```

## Output

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

## 4.2 filter()

Similar to `map()`, `filter()` takes a function and an iterable, but it constructs an iterator from those elements of the iterable for which the function returns true. In other words, `filter()` forms a list of elements for which a function returns true.

### Example

```
def is_even(x):  
    """Returns True if x is even, False otherwise."""  
    return x % 2 == 0  
numbers = [1, 2, 3, 4, 5]  
even_numbers = filter(is_even, numbers)  
print(list(even_numbers))
```

## Output

```
[2, 4]
```

**NOTE:** In both cases we converted outputs of `map()`/`filter()` to `list`, as they return custom objects, we need to convert results into iterables.

I wouldn't recommend you to refuse using `for` loops, but definitely it worth updating some parts of your applications with `map()` and `filter()`. Again, try it yourself!

## 5. Closures

Closures provide a way for a function to capture and "remember" the environment in which it was created, even when it's called outside that environment.

**IMPORTANT:** Please, read attentively and try to understand how they work. We will need closures to create custom decorators for your functions later on.

### 5.1 How it works?

A closure occurs when a function has access to a local variable from an enclosing scope that has finished its execution.

When functions in Python refer to variables in their enclosing scope, the values of those variables are captured and retained throughout the lifetime of the function object.



Even if the outer function has returned, the inner function still has access to those captured variables. This behavior forms a closure.

## Example

There are lots of practical use cases where closures may be useful, but I want to highlight the point, that it is possible to retain state with them (variables from the outer function) without using global variables or object properties.

```
def outer_function(msg):  
    message = msg # A local variable within the outer_function  
    def inner_function():  
        # The inner_function has access to the 'message' variable of the  
outer_function.  
        print(message)  
    return inner_function # The outer_function returns the inner_function  
# Create a closure  
hi_func = outer_function('Hi')  
bye_func = outer_function('Bye')  
# Call the closures  
hi_func()  
bye_func()
```

## Output

```
Hi  
Bye
```

## Explanation

1. When we call `outer_function('Hi')`, the local variable `message` is set to `'Hi'`. The `outer_function` then returns the `inner_function`.
2. The returned `inner_function` retains access to the `message` variable of `outer_function`. This retained access to the `message` variable is a closure.
3. Even though `outer_function` has finished executing, the `message` variable is not garbage collected. This is because the `inner_function` closure retains a reference to it. When `hi_func()` is called, it still has access to the `message` variable of the `outer_function`, allowing it to print `'Hi'`.
4. Each call to `outer_function` creates a new closure. So, `hi_func` and `bye_func` are independent of each other. Each closure retains its own unique environment. `hi_func` retains the environment where `message` was `'Hi'`, and `bye_func` retains the environment where `message` was `'Bye'`.

For now, the best will be to stop at this point, think about closures and try to experiment with them, before moving to decorators. Try to realise the concept of closures and create one with your hands.

## 6. Decorators

Decorators provide the ability to alter the functionality of a function or method.

When you decorate a function with a decorator, you're essentially replacing that function with a new function that typically calls the original function and does something extra.

Decorators use the @ symbol and are placed on the line before the function definition.

### Example

```
def my_decorator(func):
    def wrapper(*args, **kwargs):
        print("Do something before")
        result = func(*args, **kwargs)
        print("Do something after")
        return result
    return wrapper

@my_decorator
def say_hello(name, greeting="Hello"):
    print(f"{greeting}, {name}!")

say_hello("Alice", greeting="Hi")
```

### Explanation

1. **The Decorator Function:** This is a function that takes another function as an argument. This function will usually define an inner function.
2. **The Inner Function:** This function is defined inside the decorator function and is where you put the code that you want to execute before and/or after the original function runs. This function will call the original function at some point, and return the result of that call.
3. **Returning the Inner Function:** The decorator function will return the inner function. This way, when you use the decorator, you're replacing the original function with the inner function.

### Output

```
Do something before
Hello, Alice!
Do something after
```

As you can see in the example above in decorator functions we can even modify the arguments or return values which allows for extreme flexibility of the code.

Generally we use decorators in order to enable code reusability (which can be applied to ANY function or method, without direct altering of behaviour, without changing the code of these functions).

As well, you can add an additional validation or measure the time of execution of your functions.

### Example

```

def superuser(func):
    allowed_users = ["Yehor", "Sasha"]
    def wrapper(*args, **kwargs):
        if kwargs["name"] not in allowed_users:
            raise KeyError("Oops, you are not a superuser")
        result = func(*args, **kwargs)
        return result
    return wrapper

def validate_strings(func):
    def wrapper(*args, **kwargs):
        for value in kwargs.values():
            if not isinstance(value, str):
                raise TypeError("All arguments must be strings")
        return func(*args, **kwargs)
    return wrapper

# Yes! You can apply several decorators to the function
# The resolution of nested decorators will be from the top to the bottom
@validate_strings
@superuser
def say_hello(name, greeting="Hello"):
    print(f"{greeting}, {name}!")

say_hello(name="Sasha")
say_hello(name="Dima")

```

Note, that you can decorate a function indirectly, not using @ as a syntax sugar.

```

decorator = superuser(say_hello)
decorator(name="Yehor", greeting="Morning")

```

## Output

```

Hello, Sasha!
KeyError: 'Oops, you are not a superuser'

```

I LOVE DECORATORS! Hopefully you too now!

## 7. Lambda Functions

Lambda function in Python is a small anonymus function, meaning that it doesn't have a name. It can take any number of arguments but it can only have one expression.

### 7.1 Syntax

To define a lambda function, use the `lambda` keyword followed by a comma-separated list of arguments, a colon `:`, and then the expression.

#### Example

```
# Lambda function that multiplies its input by 5
l1 = lambda a: a * 5
print(l1(2)) # Output: 10
# Lambda function that sums three arguments
l2 = lambda a, b, c: sum([a, b, c])
print(l2(2, 3, 4)) # Output: 9
```

The simplicity of lambda functions is demonstrated here, showing how they directly return the result of a single expression.

Generally, lambdas ideal for encapsulating small bits of functionality that you do not need to reuse elsewhere. As well, the best usage will be passing them into `map()` or `filter()` functions.

### Example

```
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
squared_numbers = map(lambda x: x * x, numbers)
print(list(squared_numbers))
even_numbers = filter(lambda x: x % 2 == 0, numbers)
print(list(even_numbers))
```

### Output

```
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
[2, 4, 6, 8, 10]
```

Create --> Execute --> Say Goodbye!

## 7.2 Lambda vs Def

I am not a big fan of lambdas, though, should admit that I have nothing against them! I guess, the best in this situation would be referring to this table while making a decision what should your code do.

Feature	<b>lambda</b>	<b>def</b>
Syntax	<code>lambda args: expression</code>	<code>def name(args): body</code>
Expressions Allowed	Single	Multiple
Reusability	Single-use, typically	Designed for reuse
Use Case	Small, one-liner	More complex logic
Named	No	Yes

## 8. Function Composition

Function composition is the process of combining two or more functions to produce a new function. Composing functions together is like snapping together a series of pipes for our data to flow through.

### Example

```
def f(x):  
    return x + 4  
def g(x):  
    return 2 * x ** 2 + 3  
print(f(g(5)))
```

### Output

57

### Explanation

`g()` is being executed -> returns a value and passes it to the `f()` function.

The best approach to compose functions in a more functional programming style is to use higher-order functions along with lambda functions.

### Example

```
def compose(f, g):  
    return lambda x: f(g(x))  
def add_five(x):  
    return x + 5  
def multiply_three(x):  
    return x * 3  
composed_function = compose(add_five, multiply_three)  
print(composed_function(5))
```

### Output

20

### Explanation

In the example provided, the function `compose` is a higher-order function because it takes two functions, `f` and `g`, as its arguments. It returns a new function that represents the composition of `f` and `g`. In other words, it creates a new function where `g` is applied first to any input, and then `f` is applied to the result of `g`.

The final output is 20, which is the result of first multiplying 5 by 3 (getting 15), and then adding 5 to the result (getting 20).

## 9. Currying

Function currying is a functional programming concept where a function, instead of taking multiple arguments at once, takes the first argument and returns a new function until all arguments have been provided.

Then, the final result is returned. It's a way of translating a function that takes multiple arguments into a sequence of functions that each take a single argument.

Let's look at an example to understand how currying works:

### Example

Default approach of calling the function

```
def add(a, b, c):  
    return a + b + c  
result = add(1, 2, 3) # Call the function with all arguments at once  
print(result) # Output: 6
```

### Example

Using currying

```
def add(a):  
    return lambda b: (  
        lambda c: a + b + c  
    )  
add_one = add(1) # Returns a function that takes the second argument  
add_two = add_one(2) # Returns a function that takes the third argument  
result = add_two(3) # Finally computes the result  
print(result) # Output: 6
```

Curried functions allows you to delay computation. You can pass around intermediate functions (with some arguments fixed) and only complete the computation later, when all arguments are available.

If you are familiar with databases, consider it as a sort of transaction in DB. It might be useful in some cases when you need to make several operations at one time.

## 10. Recursion

Recursion is a programming method where a function calls itself to complete a job or solve an issue.

### Example

```
def fibonacci(n):  
    if n <= 1:  
        return n  
    else:  
        return fibonacci(n - 1) + fibonacci(n - 2)  
for i in range(10):  
    print(fibonacci(i), end=' ')
```

### Output

```
0 1 1 2 3 5 8 13 21 34
```

### Explanation

In this example `fibonacci()` is calling itself `n` times. That is useful when you need to run repetitive task which implies the same algorithm several times to get the final result eventually.

You have to understand that such operations are extremely resource consuming and try to avoid recursion in your apps. But in case it's impossible or unavoidable, you may refer to such mechanism to solve your issue.

## 11. Function Memorization

Memorization is a technique for storing the outcomes of earlier function calls to expedite subsequent computations.

Repeated function calls with the same inputs allow us to save the prior values rather than doing pointless calculations again.

Calculations are significantly accelerated as a consequence. One example of how you might want to implement memorization in your code is decorators.

### Example

Let's begin by creating the functions themselves. For the better demonstration of effectiveness of memorization technique we can use recursive Fibonacci sequence calculator function which usually requires a lot of computational power.

```

def memoize(func):
    cache = {}
    def wrapper(*args):
        if args in cache:
            return cache[args]
        else:
            result = func(*args)
            cache[args] = result
            return result
    return wrapper

@memoize
def fibonacci(n):
    if n <= 1:
        return n
    else:
        return fibonacci(n - 1) + fibonacci(n - 2)

```

Basically, we cache the result in memory which eventually will lead to better performance once we decide to refer to the same function again.

## Example

```

from datetime import datetime
# non memoized function time
start_time = datetime.now()
print(fibonacci(499))
non_memoized_function_time = datetime.now() - start_time
print(f"execution of non memoized function took {non_memoized_function_time}")
# memoized function time
start_time = datetime.now()
print(fibonacci(499))
memoized_function_time = datetime.now() - start_time
print(f"execution of memoized function took {memoized_function_time}")
# Difference
print(f"non      memoized      function      took      {non_memoized_function_time      -
memoized_function_time} more time")

```

**IMPORTANT:** Pass a small value to fibonacci function, I have created a random output as calculation for 499 Fibonacci numbers will take ages and why this is happening will be explained in a lesson about algorithms.

## Output

```

8616829160023845073278831216566478809594106832606088332452990347014905611582359
2713458328176574447204501
execution of non memoized function took 0:00:00.000573
8616829160023845073278831216566478809594106832606088332452990347014905611582359
2713458328176574447204501
execution of memoized function took 0:00:00.000015

```



As you can see in the output above, there is a huge time difference between two times thanks to memoization.

**Note:** Do not get confused by the fact that we called same function twice - the first time we called the function its result hasn't been recorded yet, therefore it is the same as calling unmemoized function.

## 12. Quiz

### Question 1:

What is a higher-order function in Python?

- A) A function that operates at a higher security level.
  - B) A function that accepts other functions as arguments or returns a function as a result.
  - C) A function that can only be executed as an administrator.
  - D) A function that performs higher mathematical calculations.
- 

### Question 2:

What does the `map()` function do?

- A) Maps a function to a specific module.
  - B) Applies a given function to each item of an iterable and returns a list of results.
  - C) Creates a visual map of function calls.
  - D) Translates a function from one programming language to another.
- 

### Question 3:

What is the purpose of function currying?

- A) To add flavor to the function.
  - B) To secure a function against external modifications.
  - C) To transform a function with multiple arguments into a sequence of functions each taking a single argument.
  - D) To optimize a function for faster execution.
- 

### Question 4:

Which of the following is a correct example of a lambda function?

- A) `lambda x, y: x + y`
  - B) `def lambda(x, y): return x + y`
  - C) `lambda(x, y): x + y()`
  - D) `lambda = (x, y): x + y`
- 

### Question 5:

How does memoization enhance function performance?

- A) By converting the function to machine code directly.
  - B) By storing the results of expensive function calls and returning the cached result when the same inputs occur again.
  - C) By distributing function execution across multiple processors.
  - D) By rewriting the function in a more efficient programming language.
- 

### Question 6:

What is a closure in Python?

- A) A syntax for closing a file after it's been opened.
  - B) A technique to terminate a loop early.
  - C) An inner function that remembers and has access to variables in the local scope in which it was created, even after the outer function has finished executing.
  - D) A special type of error that occurs when a function finishes execution.
- 

### Question 7:

In Python, what does the decorator syntax typically involve?

- A) The # symbol.
  - B) The @ symbol before a function definition.
  - C) The & symbol.
  - D) The ! symbol.
- 

### Question 8:

What is the main advantage of using recursion in programming?

- A) It simplifies the code for functions where the solution involves solving smaller instances of the same problem.
- B) It always speeds up the execution time.

- C) It uses less memory compared to iterative solutions.
- D) It can be used with any function without restrictions.

### Question 9:

What will be the output of the following Python code snippet using the `filter()` function?

```
nums = [1, 2, 3, 4, 5, 6]
filtered_nums = filter(lambda x: x % 2 == 0, nums)
print(list(filtered_nums))
```

- A) [1, 3, 5]
  - B) [2, 4, 6]
  - C) [1, 2, 3, 4, 5, 6]
  - D) []
- 

### Question 10:

Consider the following function that uses recursion to calculate the factorial of a number. What value is returned when calling `factorial(5)`?

```
# Ha-ha, it's a question from math :)
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n-1)
print(factorial(5))
```

- A) 120
  - B) 24
  - C) 100
  - D) None
- 

### Question 11:

What is demonstrated by the following code snippet using function composition?

```
def double(x):
    return x * 2
def increment(x):
    return x + 1
def compose(f, g):
    return lambda x: f(g(x))
```

```
f = compose(double, increment)
print(f(3))
```

- A) The code doubles the input and then increments it.
  - B) The code increments the input and then doubles it.
  - C) The code triples the input.
  - D) The code increments the input twice.
- 

### Question 12:

What does the following curried function calculate, and what will be the output when executed as shown?

```
def multiply(a):
    return lambda b: a * b
multiply_by_3 = multiply(3)
result = multiply_by_3(5)
print(result)
```

- A) Adds 3 to 5; output is 8
  - B) Multiplies 3 by 5; output is 15
  - C) Divides 5 by 3; output is approximately 1.67
  - D) Subtracts 3 from 5; output is 2
- 

## 13. Homework

In this homework I will provide some snippets, which you will have to modify to consolidate what you've learnt. Additionally, create a decorator which measures the execution time and apply to each function!

### Task 1: Inventory Manager

**Objective:** Implement a function that handles adding and updating an inventory for a fantasy game character using higher-order functions.

#### Requirements:

- Write a `manage_inventory()` function that can take commands such as "add" or "update" along with item details and modifies the inventory accordingly.

- Use closures to encapsulate the inventory state within the manager.

```
def inventory_manager():  
    inventory = {}  
    def manage(command, item, quantity):  
        pass  
    return manage
```

## Task 2: Message Encoder

**Objective:** Create a simple text encoder using function composition that applies multiple transformations to text.

### Requirements:

- Define multiple small lambda functions for different text transformations (e.g., reverse, encode characters, etc.).
- Compose these functions to create a more complex text encoding function.

```
def compose(*functions):  
    def composed_func(input):  
        pass  
    return composed_func  
encoder = compose(reverse, encode)  
# Encode a message  
encoded_message = encoder("hello world")
```

## Task 3: Music Composer

**Objective:** Implement a system that lets users compose their own music by adding notes and applying effects using higher-order functions.

### Requirements:

- Define functions for adding musical notes and applying various effects like echo or speed change.
- Use function composition to apply multiple effects to a sequence of notes.

```
def add_notes(*notes):  
    pass  
def echo_effect(notes):  
    # Be creative!  
    pass  
def speed_change(notes, factor):  
    # Be creative!  
    pass  
# Create a new composition and apply effects
```

```
echoed = compose_music(echo_effect)
print(f"Echoed Composition: {echoed}")
# Apply speed change
faster = speed_change(echoed, 2)
print(f"Faster Composition: {faster}")
```

You can even create more decorators based on your taste and try to apply them!

Congratulations with mastering the functional programming! Looking forward to see your solutions!

# Lesson 15: OOP

## (Object oriented programming)

"Object-oriented programming: the blueprint for building digital worlds, where data and behavior unite in elegant real world objects"

### 1. Intro to OOP

#### 1.1 What is it?

Object-Oriented Programming (OOP) is a programming paradigm based on the concept of objects, which can contain data and code.

There are two main classifiers which object could have attributes and methods. Let's take a closer look on them!

#### 1.2 Attributes vs Methods

Classifier	Attributes	Methods
Definition	Attributes represent the <i>state or qualities of the object</i> , often called <i>fields</i> or <i>properties</i> .	Methods are <i>functions defined inside an object</i> . They represent the <i>behavior</i> or <i>actions</i> that <i>an object can perform</i> .
Usage	Used to <i>store information about the object</i> , like <i>size, color, or other properties</i> .	Used to define actions that can be performed by the object, like calculations, operations, or any other functions.

Data is represented and structured in the following format. Below you can see the attributes and methods defined for the dict class.

#### Example

```
print(dir(dict))
```

Output

```
['clear', 'copy', 'fromkeys', 'get', 'items', 'keys', 'pop', 'popitem',  
'setdefault', 'update', 'values']  
# clear -> method (because it's an operation on the object)  
# 'copy', -> method  
# 'fromkeys' -> method  
# 'get', -> method
```

```
# ...
# 'values'          -> attribute (field of the object) -> returns all values of
the dictionary
# 'items',         -> attribute
# 'keys',          -> attribute
```

## 1.2 Real world examples

There are lots of examples can be found just around the world. Let's take a look at car.

What color is it? What is the maximum speed? These are all - **properties** of the object.

The car can drive, break, stall and rev. These are - **methods** of the object.

Let's take a look at the table below with more examples:

Real-World Object	Attributes	Methods
<i>Car</i>	color, brand, horsepower, fuel_level	drive(), brake(), stall(), rev()
<i>Bank Account</i>	account_number, balance, account_holder	deposit(amount), withdraw(amount), check_balance()
<i>Smart phone</i>	model, operating_system, battery_percentage, screen_size	call(number), send_message(content), take_photo()
<i>Book</i>	title, author, number_of_pages, genre	read_page(page_number), bookmark_page(page_number), close()

Same it can be represented in the world of programming. You simply create the `class`, define its attributes and its methods. It comes in handy, when you can have a custom object for specific needs.

## 1.3 Syntax



In Python, classes are defined using the `class` keyword, followed by the class's name and a colon. Inside the class, methods (functions) are *defined to implement the behaviors of the objects*.

**Note:** Take a look at indentation, in case it's wrong the Python interpreter will consider method as a function.

## Example

```
class Car:
    def __init__(self, color, brand):          # params
        self.color = color                    # attributes
        self.brand = brand
    def drive(self):                          # method of the class
        print("This car is now driving.")
    def brake(self):                         # method of the class
        print("The car has stopped.")
car_1 = Car('brown', 'bugatti')              # create a specific instance of the
class (Calling __init__() under the hood)
car_2 = Car('blue', 'volvo')
# Attributes can be accessed with a ``.(dot)`` notation, same as we've seen
before with dictionary
print(f"Andrew Tate owns a {car_1.color} {car_1.brand}")
print(f"Average person owns a {car_1.color} {car_1.brand}")
# Methods as well
car_1.drive()
```

## Output

```
Andrew Tate owns a brown bugatti
Average person owns a brown bugatti
This car is now driving. # Buggati is driving, while Volvo is stationary for
now.
```

## Explanation

- We created a class `Car` with the following attributes `color` and `brand`, 2 instances of this class - (`car_1`, `car_2`) and called method `drive()` for the first instance.
- `self` - **specific instance** of the class. By using `self`, we can access the *attributes* and *methods* of the class in Python. It binds the *attributes* with the given *arguments*.
- `__init__(*args, **kwargs)` - is a **special method** that's *automatically called* when a new instance (object) of a class is created. It is also known as constructor method.

Consider `self` as reference to specific object, for example we are all people, but each person (`self`) is unique.

## Example

```
print(id(car_1))
print(id(car_2))
```

## Output

```
140094732495696
140094732495760
```

**Note:** Calling `__init__()` initialising the newly created object's attributes with specific values, which we pass while creating an object.

**Note:** `car_1` and `car_2` are completely different objects, although both of them are instances of the same class, this can be proved by using `id()` function.

Now you can see that it is very convenient way to store some data with Python classes, as you don't have to define keys and values like you did it using dictionary.

## Example

```
class Book:
    def __init__(self, title, author, is_borrowed=False):
        self.title = title
        self.author = author
        self.is_borrowed = is_borrowed
    def borrow_book(self):
        if not self.is_borrowed:
            self.is_borrowed = True
            return f"You have borrowed '{self.title}' by {self.author}."
# Yes, you can use this
        else:
            return f"'{self.title}' is already borrowed."
    def return_book(self):
        if self.is_borrowed:
            self.is_borrowed = False
            return f"'{self.title}' has been returned."
        else:
            return f"'{self.title}' was not borrowed."
# Creating instances of ``Book``
book_1 = Book("1984", "George Orwell")
book_2 = Book("To Kill a Mockingbird", "Harper Lee")
# Borrowing the 1st book
print(book_1.borrow_book())
# Attempting to borrow the 1st book again
print(book_1.borrow_book())
# Borrowing the 2nd book
print(book_2.borrow_book())
# Returning the first book
print(book_1.return_book())
# Checking the status of the second book
print(f"Is '{book_2.title}' borrowed? {'Yes' if book_2.is_borrowed else 'No'}")
# BTW, we can define a method for that inside the class, just use your
imagination!
```

## Output

```
You have borrowed '1984' by George Orwell.  
'1984' is already borrowed.  
You have borrowed 'To Kill a Mockingbird' by Harper Lee.  
'1984' has been returned.  
Is 'To Kill a Mockingbird' borrowed? Yes
```

## 1.4 Object-Oriented vs Functional Programming?

Once we have learnt about functional and object oriented programming, there is often sort of confusion exists which to use among both paradigms.

Object-Oriented Programming (OOP) is often chosen for *large, complex systems* where encapsulating data and behavior into objects makes the code more *manageable, readable* and *reusable*.

It is *great for modeling real-world entities* and is commonly used in software development for user interfaces, simulations, and large-scale applications.

Functional Programming is often used in *data science world*. Functions do specific things, but classes - are specific things.

Frankly speaking, in real world programming the majority of applications are written using OOP, as it is a modern approach for high-level languages. But it's a good practice to combine both paradigms and use functions with classes in your application.

## 2. Class VS Instance Attributes and Methods

When working with classes in object-oriented programming, it's crucial to understand the difference between *class attributes* vs *instance attributes*.

### 2.1 Instance Attributes

1. *Instance attributes and methods* are *tied to a specific instance of a class*.
2. *Each instance has its own copy of these attributes and methods*.
3. *Changing an instance attribute only affects that particular instance, not all instances of the class*.

### 2.2 Class Attributes

1. *Use these when the value or behavior should be the same across all instances of the class*.
2. *They are not tied to any particular instance of the class*.
3. *If the class attribute value is changed, the change is reflected across all instances*.

### 2.3 Syntax

You can access the class attributes with several ways: `Class.class_attribute` or `instance.class_attribute`.

## Example

```
class Vehicle:
    total_vehicles = 0 # Class attribute
    def __init__(self, make, model):
        self.make = make # Instance attribute
        self.model = model # Instance attribute
        Vehicle.total_vehicles += 1
# Creating instances of Vehicle
car1 = Vehicle("Toyota", "Corolla")
car2 = Vehicle("Ford", "F-150")
# Accessing class attribute
print("Total vehicles:", Vehicle.total_vehicles)
# Accessing instance attributes
print(car1.make, car1.model)
print(car2.make, car2.model)
"""
# As it was mentioned before, the value and behavior is the same across all
instances of the class.
# Class attributes and methods can be accessed through the instances of that
class.
"""
print(car1.total_vehicles == car2.total_vehicles) # True
```

## Output

```
Total vehicles: 2
Toyota Corolla
Ford F-150
True
```

## Example

```
class User:
    active_users = 0 # Class attribute
    def __init__(self, username):
        self.username = username # Instance attribute
        User.active_users += 1
# Creating instances of ``User`` (calling __init__())
user1 = User("Alice")
user2 = User("Bob")
print(user1.active_users)
print(user2.active_users)
```

## Output

```
2
```

## Explanation

In each example, the `class` attribute is shared among all instances of the class and reflects a property or statistic that is relevant to the class. But outputting them using `.(dot)` notation is not really a comfortable way to manipulate the object. There are some more efficient ways described below in section 3.

## 3. Class vs Instance methods + @staticmethod

In Python methods within a class can be categorized into three types based on their interaction with class or instance attributes / instance methods, class methods, and static methods. Each type serves its unique purpose and is defined differently.

### 3.1 Instance Methods

**Definition:** Functions defined inside a class that operate on instances of the class. They can freely *access* and *modify* instance attributes and other instance methods.

**First Parameter:** `self` - refers to the individual instance of the class.

**Usage:** Used for operations that **require data specific to an individual object**.

### Example

```
class Vehicle:
    def __init__(self, make, model):
        self.make = make
        self.model = model
    # Instance method
    def display_info(self):
        return f"This vehicle is a {self.make} {self.model}."
car = Vehicle("Toyota", "Corolla")
print(car.display_info()) # Outputs: This vehicle is a Toyota Corolla.
```

### 3.2 Class Methods

**Definition:** Class methods are functions defined inside a class that *operate on the class itself*, rather than on instances of the class.

**First Parameter** `cls` - refers to the class itself, not the instance.

**Decorator:** `@classmethod` - indicates that the method is a class method.

**Usage:** Typically used for operations that apply to the class *as a whole*, rather than to individual objects.

```
class Vehicle:
    total_vehicles = 0
    @classmethod
    def increment_total_vehicles(cls):
        cls.total_vehicles += 1
Vehicle.increment_total_vehicles()
print(Vehicle.total_vehicles)  # Outputs: 1
```

### 3.3 Static Methods

**Definition:** Static methods are functions defined inside a class that *don't implicitly access either class or instance attributes*.

**Decorator:** @staticmethod - indicates that the method is a static method.

**Usage:** Typically used for utility functions that perform a task in isolation. They can't modify class or instance state.

```
class Vehicle:
    @staticmethod
    def is_motorcycle(wheels):
        return wheels == 2
print(Vehicle.is_motorcycle(2))  # Outputs: True
print(Vehicle.is_motorcycle(4))  # Outputs: False
```

**Note:** There is no First Parameter passed as an argument to the method.

Understanding *when* and *how* to use each method type is **crucial** for designing effective and logical classes and the whole system design.

#### Example

```
class BankAccount:
    # Class attribute
    total_accounts = 0
    def __init__(self, owner, balance=0):
        self.owner = owner
        self.balance = balance
        # Increment the total accounts (NOTE: we can actually create a method
        # which does this and call it in our constructor)
        BankAccount.total_accounts += 1
    # Instance method (for direct interaction with ``specific object``)
    def deposit(self, amount):
        self.balance += amount
        return f"{self.owner}'s account: Deposited ${amount}. New balance: ${self.balance}."
```

```

# Instance method (for direct interaction with ``specific object``)
def withdraw(self, amount):
    if amount > self.balance:
        return f"{self.owner}'s account: Insufficient funds. Withdrawal
denied."
    self.balance -= amount
    return f"{self.owner}'s account: Withdrew ${amount}. New balance:
${self.balance}."
# Class method to get total accounts
@classmethod
def get_total_accounts(cls):
    return f"Total bank accounts opened: {cls.total_accounts}."
# Static method to check if a withdrawal amount is within a daily limit
@staticmethod
def check_daily_limit(amount, daily_limit=500):
    return amount <= daily_limit
# Creating bank account instances
account1 = BankAccount("John Doe", 1000)
account2 = BankAccount("Jane Doe", 500)
# Calling instance methods
print(account1.deposit(500))
print(account2.withdraw(200))
# Calling class method
print(BankAccount.get_total_accounts())
# Calling static method
print(BankAccount.check_daily_limit(400))
print(BankAccount.check_daily_limit(600))

```

## Output

```

John Doe's account: Deposited $500. New balance: $1500.
Jane Doe's account: Withdrew $200. New balance: $300.
Total bank accounts opened: 2.
True
False

```

You can experiment with all methods described above and create powerful custom classes according to the rules.

## 4. Key Paradigms of OOP

There are several OOP concepts which every developer on the planet Earth **must have to know** .

### 4.1 Encapsulation

Encapsulation is often considered the *first pillar* of Object-Oriented Programming. It refers to the *bundling of data with the methods that operate on that data*.

It is used to *hide the internal representation, or state, of an object from the outside*.

Let's explore Encapsulation through a real-world example: a Coffee Machine.

You (the customer) interact with it through a simple interface:

1. Selecting a type of coffee.
2. Placing a cup.
3. Pressing a button.

However, the internal processes are:

1. Grinding beans .
2. Heating water.
3. Mixing coffee and water with a specific pressure and temperature.

This is encapsulation in action - *exposing only the necessary controls to the user* and *hiding the complex process*.

In programming, encapsulation is implemented through the use of private/protected/public attributes and methods.

In simple words it is called the access levels:

- **Public Access:** By default, *all attributes and methods* in a Python class are public. They *can be easily accessed from outside the class*.
- **Protected Access** (`_`): This is more of a convention than enforced by the language. It signals that these *attributes and methods should not be accessed directly*, even though Python *does not strictly enforce this*.
- **Private Access** (`__`): Python performs name mangling on these names. This means that Python *interpreter changes the name of the variable* in a way that makes it harder to create subclasses that accidentally override the private attributes and methods.

The decent code which can represent Encapsulation as a concept is the following:

## Syntax

```
class Smartphone:
    def __init__(self, brand, model):
        self.brand = brand # Public attribute
        self.model = model # Public attribute
        self.__battery_level = 100 # Private attribute (initially fully
        charged)
        self._installed_apps = [] # Protected attribute (initial app list)
    def install_app(self, app_name):
        """Public method to install a new app."""
        if app_name not in self._installed_apps:
            self._installed_apps.append(app_name)
            print(f"App '{app_name}' installed.")
        else:
            print(f"App '{app_name}' is already installed.")
    def uninstall_app(self, app_name):
        """Public method to uninstall an app."""
```



```

    if app_name in self._installed_apps:
        self._installed_apps.remove(app_name)
        print(f"App '{app_name}' uninstalled.")
    else:
        print(f"App '{app_name}' is not installed.")
def show_installed_apps(self):
    """Public method to display installed apps."""
    print("Installed Apps:", self._installed_apps)
def __check_battery(self, required_amount):
    """Private method to check if enough battery is available."""
    return self.__battery_level >= required_amount
def get_battery_level(self):
    """Public method to check the battery level."""
    return f"Current battery level: {self.__battery_level}%"
# Creating and interacting with a Smartphone object
my_phone = Smartphone('Pixel', 'Pixel 5')
# Installing and uninstalling apps
my_phone.install_app('WhatsApp')
my_phone.install_app('Spotify')
my_phone.uninstall_app('WhatsApp')
my_phone.show_installed_apps()
# Using battery and charging
print(my_phone.get_battery_level())
my_phone.charge_phone(30)

```

## Explanation

Attribute/Method	Type	Access Level	Description
brand	Attribute	Public	Can be accessed both inside and outside the class.
model	Attribute	Public	Can be accessed both inside and outside the class.
__battery_level	Attribute	Private	Can only be accessed and modified within the class.
_installed_apps	Attribute	Protected	Intended for internal use within the class or subclasses, but can technically be accessed from outside (Don't do that!)
install_app(app_name)	Method	Public	Can be accessed outside

Attribute/Method	Type	Access Level	Description
			the class.
<code>uninstall_app(app_name)</code>	Method	Public	Can be accessed outside the class.
<code>show_installed_apps()</code>	Method	Public	Can be accessed outside the class.
<code>charge_phone(amount)</code>	Method	Public	Can be accessed outside the class.
<code>get_battery_level()</code>	Method	Public	Can be accessed outside the class.
<code>__check_battery(amount)</code>	Method	Private	Can only be accessed within the class. Checks if the battery level is sufficient for a specified operation.

## Output

```
App 'WhatsApp' installed.
App 'Spotify' installed.
App 'WhatsApp' uninstalled.
Installed Apps: ['Spotify']
Current battery level: 100%
```

Basically, we created a good and solid interface for the user to interact with, moreover, we have hidden the technical implementation from the user, which doesn't have to fully understand how it works inside, they would just use the Smartphone straightforward.

## Example

```
class CoffeeMachine:
    def __init__(self):
        self.__water_level = 1000
        self.__beans_quantity = 500
        self.__temperature = 90
    def make_coffee(self, coffee_type):
        if not self.__check_resources(coffee_type):
            return "Please refill the machine."
        return f"Enjoy your {coffee_type}!"
    def __check_resources(self, coffee_type):
        # Private method to check if there are enough resources to make the
        coffee
```

```

        if coffee_type == "espresso" and self.__water_level >= 50 and self.__beans_quantity >= 30:
            self.__use_resources(50, 30)
            return True
        # Additional conditions for other coffee types can be added here
        return False
    def __use_resources(self, water_used, beans_used):
        # Private method to use resources
        self.__water_level -= water_used
        self.__beans_quantity -= beans_used
    def refill_water(self, water_quantity):
        self.__water_level += water_quantity
    def refill_beans(self, beans_quantity):
        self.__beans_quantity += beans_quantity
# Interacting with the coffee machine
coffee_machine = CoffeeMachine()
print(coffee_machine.make_coffee("espresso")) # Enjoy your espresso!
coffee_machine.refill_water(500)

```

## Output

```
Enjoy your espresso!
```

The general usage:

1. **Data Hiding** The internal state of the coffee machine is hidden from the outside world. Users interact with a simple interface without worrying about the internal processes.
2. **Access Modifiers**: Users cannot directly access the internal components (like the water heater or the grinder); they can only use the buttons provided.
3. **Simplicity**: The user of the coffee machine doesn't need to know the exact process of how coffee is made. They only select the type of coffee they want and let the machine handle the rest.
4. **Maintenance**: If something goes wrong inside the coffee machine, or if an improvement is made to the internal mechanism (like a more efficient grinder), it doesn't affect the user's interaction with the machine. The interface remains the same.

### 4.1.1 @property

The `@property` decorator in Python allows you to define methods in a class that behave like *attributes*. But behind the scenes, *it can perform complex computations*, such as fetch data, or implement logic with constraints.

**Note:** If you also need to define a setter function, you can use the `@property_name.setter` decorator, where `property_name` is the name of the property.

## Example

```

class CarEngine:
    def __init__(self):
        self._rpm = 0 # Engine RPM, protected attribute

```

```

        self._temperature = 70 # Engine temperature in Fahrenheit, protected
attribute
    @property
    def rpm(self):
        """Get the current RPM of the engine."""
        return self._rpm
    @rpm.setter
    def rpm(self, value):
        """Set the RPM of the engine, ensuring it's within a safe operational
range."""
        if 0 <= value <= 8000:
            self._rpm = value
            print(f"RPM set to {value}.")
        else:
            print("RPM must be between 0 and 8000.")
    @property
    def temperature(self):
        """Get the current temperature of the engine."""
        return self._temperature
    @temperature.setter
    def temperature(self, value):
        """Set the temperature of the engine, ensuring it's within a safe
operational range."""
        if 50 <= value <= 250:
            self._temperature = value
            print(f"Temperature set to {value}°F.")
        else:
            print("Temperature must be between 50°F and 250°F.")
    def start_engine(self):
        """Simulate starting the engine."""
        if self._rpm == 0:
            self.rpm = 1500 # Setting RPM to a typical idle speed.
            print("Engine started.")
        else:
            print("Engine is already running.")
# Using the CarEngine class
my_car_engine = CarEngine()
my_car_engine.start_engine()
print(f"Current RPM: {my_car_engine.rpm}")
# Trying to set the RPM and temperature
my_car_engine.rpm = 9000
my_car_engine.temperature = 300
my_car_engine.rpm = 3000
my_car_engine.temperature = 200

```

## Output

```

RPM set to 1500.
Engine started.
Current RPM: 1500
RPM must be between 0 and 8000.
Temperature must be between 50°F and 250°F.
RPM set to 3000.

```

**Use (Getters/Setters):** If you need to expose internal state, do it through (getters) and (setters) to maintain control over the state. *This allows for validation, logging, or other controls each time an attribute is accessed or modified.*

## 4.2 Inheritance

Inheritance allows us to define a class that inherits all the methods and properties from another class. It's a way to form a relationship between classes, allowing for the creation of a more complex, yet organized system.

### 4.2.1 Why?

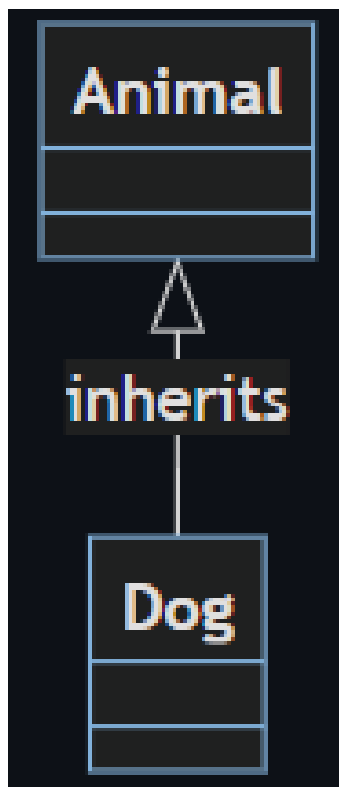
- **Code Reusability:** Inheritance promotes the reuse of code. Once a behavior is defined in a base class, *it can be inherited by other classes, avoiding duplication.*
- **Extensibility:** Modifications and enhancements can be made in the base class and all inheriting classes will automatically incorporate the changes. (Though in some cases it can violate SOLID, so be careful with this)
- **Hierarchy Creation:** It helps in creating a *hierarchical classification of classes which is natural in many real-world scenarios.*

### 4.2.2 Syntax and Types of Inheritance

#### Single Inheritance

A child class inherits from one parent class. This is the simplest form of inheritance.

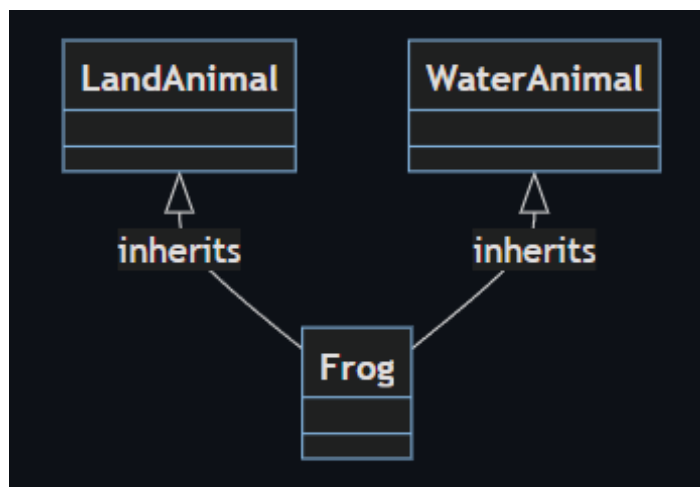
```
class Animal:
    # Parent class code
class Dog(Animal):
    # Child class Dog inherits from Animal
```



## Multiple Inheritance

A child class inherits from more than one parent class.

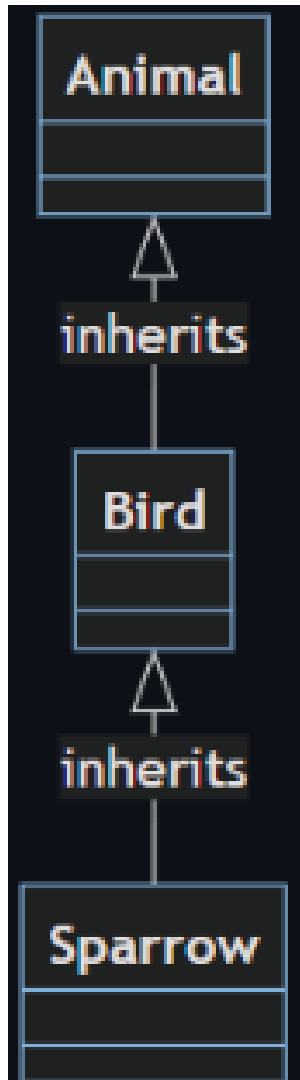
```
class LandAnimal:  
    # Code for land animals  
class WaterAnimal:  
    # Code for water animals  
class Frog(LandAnimal, WaterAnimal):  
    # Frog inherits from both LandAnimal and WaterAnimal
```



## Multilevel Inheritance

A class inherits from a child class, making it a grandchild class.

```
class Animal:
    # Base class code
class Bird(Animal):
    # Bird inherits from Animal
class Sparrow(Bird):
    # Sparrow inherits from Bird, which is a child of Animal
```



**IMPORTANT:** In order to build a good quality codebase with inheritance, you, as a programmer should analyse what classes have in common and identify **the highest level of Abstraction**.

For example Animal and Dog have the following in common. They both can walk(), sound(), eat(). In this case, the Animal is the highest level of Abstraction.

#### 4.2.3 Method `super()`

`super()` function is a crucial part of the **inheritance mechanism**. It allows you to call methods from a parent or sibling class within a child class.

```

class Book:
    def __init__(self, title, author, ISBN):
        self.title = title
        self.author = author
        self.ISBN = ISBN
        print(f"Book '{self.title}' by {self.author} created. ISBN: {self.ISBN}")
    def display_info(self):
        print(f"Title: {self.title}, Author: {self.author}, ISBN: {self.ISBN}")
# Subclass ``EBook``
class EBook(Book):
    def __init__(self, title, author, ISBN, file_format):
        super().__init__(title, author, ISBN) # Initialize the base class
        attributes
        self.file_format = file_format
        print(f"EBook format: {self.file_format}")
    def display_info_for_ebook(self):
        super().display_info() # Display info from the base class method
        print(f"File Format: {self.file_format}")
# Subclass ``PrintedBook``
class PrintedBook(Book):
    def __init__(self, title, author, ISBN, weight):
        super().__init__(title, author, ISBN) # Initialize the base class
        attributes
        self.weight = weight
        print(f"Printed Book weight: {self.weight} kg")
    def display_info_for_printed_book(self):
        super().display_info() # Display info from the base class method
        print(f"Weight: {self.weight} kg")
ebook = EBook("The Python Journey", "Jane Doe", "123456789", "PDF")
ebook.display_info_for_ebook()
printed_book = PrintedBook("The Python Journey", "Jane Doe", "123456789", 1.5)
printed_book.display_info_for_printed_book()

```

## Output

```

Book 'The Python Journey' by Jane Doe created. ISBN: 123456789
EBook format: PDF
Title: The Python Journey, Author: Jane Doe, ISBN: 123456789
File Format: PDF
Book 'The Python Journey' by Jane Doe created. ISBN: 123456789
Printed Book weight: 1.5 kg
Title: The Python Journey, Author: Jane Doe, ISBN: 123456789
Weight: 1.5 kg

```

## Explanation

In this example, the EBook and PrintedBook classes inherit from the Book class. They use `super()` to initialize the attributes from the Book class and to extend the `display_info` method.



This structure allows for shared behaviors and properties to be defined once in the `Book` class while enabling each subclass to have its own specialized attributes and methods, maintaining a clear and organized hierarchy.

Always try to follow these rules working with inheritance:

- **Ensure a Logical Hierarchy:** The *inheritance structure should reflect a logical and real-world hierarchy*.
- **Avoid Deep Inheritance Trees:** Deeply nested inheritance can lead to complexity and fragility in your code. *Prefer composition over deep inheritance where possible*.

Think in advance about the structure of inheritance and you will never have any problems with code, except of MRO, heh, no spoilers!

## 4.3 Polymorphism

Polymorphism, is translated from the Greek words 'poly' (many) and 'morph' (form), is a cornerstone concept in Object-Oriented Programming (OOP).

*It refers to the ability of different objects to respond to the same message—or in programming terms, the ability of different classes to respond to the same method call—in their own unique ways.*

We have encountered some functions in Python which use the same approach already:

### Example

```
print(len("Hello")) # Works on a string
print(len([1, 2, 3])) # Works on a list
```

### Explanation

**Polymorphic Behavior:** The `len()` function returns the length of an object. It can be applied to various data types including strings, lists, tuples, and dictionaries.

**How It Works:** Internally, `len()` calls the object's `__len__` method. The implementation of `__len__` can differ depending on the object's class, but from the user's perspective, `len()` consistently returns the size or length of the object.

So technically speaking it is the mechanism which allows the same function or operator to work with different types of objects, but the behavior remains unchanged.

Same is applied to `min()`, `max()`, `+`, `*` operators and functions.

### Key Points:

- **Code Reusability:** Developers can write a function or method once and use it with objects of multiple classes.

- **Flexibility in Code:** New components are easy to integrate into the established system.
- **Ease of Maintenance:** As changes or enhancements are needed, developers can implement without the need to modify every implementation.

### 4.3.1 Method overriding

In method overriding, a method in a child class *has the same name, return type, and parameters* as a method in its parent class.

The version of the method that *gets executed depends on the type of the object invoking it.*

#### Example

```
class Animal:
    def speak(self):
        print("This animal speaks")
class Dog(Animal):
    def speak(self):          # Overrides the parental method (same name, same
return type, same params)
        print("Dog barks")
animal = Animal()
animal.speak()
dog = Dog()
dog.speak()
```

#### Output

This animal speaks

Dog barks

#### Example

```
class Vehicle:
    def fuel_efficiency(self):
        pass
class Car(Vehicle):
    def fuel_efficiency(self):
        return "Car: Approximately 30 miles per gallon."
class Truck(Vehicle):
    def fuel_efficiency(self):
        return "Truck: Approximately 15 miles per gallon."
# List of different types of vehicles
vehicles = [Car(), Truck()]
for vehicle in vehicles:
    print(vehicle.fuel_efficiency())
```

#### Output

Approximately 30 miles per gallon.

Approximately 15 miles per gallon.

## Explanation

Depending on the type of vehicle, the overridden method *provides specific fuel efficiency details*.

There is one more type of polymorphism called Operator Overloading, but this will be explained during dunder(magic) methods section, though we have seen and worked with it already, it's always worth knowing what's under the hood.

## 4.4 Abstraction

Abstraction in Object-Oriented Programming (OOP) is a *conceptual mechanism that focuses on identifying the essential aspects of an entity* while ignoring its detailed background and explanations.

Sounds hard, right? Let's take an example from the real world (like a smart TV)

When you use a smart TV, *you don't need to know all the technical details about how TVs work internally*.

Instead, you *just use the remote control to switch it on, change channels, adjust the volume, and more*. That means that you are using the highest level of abstraction and don't dive into the complex details.

## Example

```
"""
This can be a great example of how abstraction works, everything is hidden from
the user and they still can use the player without having a look how it works
inside
"""
class MusicPlayer:
    def __init__(self, song):
        self.song = song
        # Assume there are complex operations to load and prepare the song
        print(f>Loading the song: {self.song}")
    def play(self):
        # Complex operations to play music are hidden from the user
        print(f>Playing the song: {self.song}")
    def pause(self):
        # Complex operations to pause the music are hidden
        print(f>Pausing the song: {self.song}")
    def stop(self):
        # Complex operations to stop the music are hidden
        print(>Stopping the playback.")
```

```
# Using the MusicPlayer
my_music_player = MusicPlayer("Beethoven - Symphony No.9")
my_music_player.play()
my_music_player.pause()
my_music_player.stop()
```

It's all about exposing only the necessary parts of the entity/class, simplifying what the user needs to interact with.

#### 4.4.1 Implementing Abstraction in Python

Python, being a *dynamically-typed language*, does not enforce abstraction as strictly as statically-typed languages such as Java or C++.

Abstract classes *serve as blueprints for other classes*. They allow you to **define methods that *must be created (overridden)*** within any child classes built from the abstract class.

##### *How to Implement:*

- `abc` Module: Python provides the `abc` (Abstract Base Classes) module to define abstract *base classes*.
- `@abstractmethod` Decorator: You can use the `@abstractmethod` decorator to define abstract methods within an abstract class.

```
from abc import ABC, abstractmethod
class Shape(ABC): # Inherits from ABC, making Shape an abstract class.
    """
    This class represent the highest level of abstraction.
    You simply have to find what is common between it and its subsequent
    classes
    This can be: area, perimeter, volume etc..
    Same you can create the following hierarchy
    ``Animal`` is the highest level of abstraction, all animals have in common
    the following things such as: `speak()`, `eat()`, `drink()`, etc..
    """
    @abstractmethod
    def area(self):
        pass # No implementation here. Subclasses MUST provide an
        implementation.

    @abstractmethod
    def perimeter(self):
        pass # No implementation here. Subclasses MUST provide an
        implementation.
class Rectangle(Shape):

    def __init__(self, length, width):
```

```

        self.length = length
        self.width = width

    def area(self):
        return self.length * self.width

    def perimeter(self):
        return 2 * (self.length + self.width)

# rect = Shape() # This will raise an error, as you can't instantiate an
# abstract class.
rect = Rectangle(10, 20)
print(rect.area())
print(rect.perimeter())

```

## Output

200

60

**Very important:** An abstract method is a method that has a declaration but does not have an implementation. But the specific implementation can be added for the methods in child classes based on the objects` behaviour.

### 4.4.2 Best Practices

Abstraction is a powerful tool in software development, but like any tool, it must be used judiciously.

You have to use each principle of the Object Oriented Programming wisely and consider everything in advance before taking an action.

The starting point is to understand the needs of your application and the users to define abstractions that *accurately reflect real-world entities and operations*.

```

"""
Imagine you're building an online store.
Initially, you might have a simple ``Product`` class. As the store grows, you
recognize the need to categorize products.
!!! Instead of creating separate classes for each category with duplicated
code, you abstract the common features into the `Product` class and use
subclasses for specific behaviors. !!!
"""

class Product:
    def __init__(self, name, price):
        self.name = name
        self.price = price

```

```

class Book(Product):
    def __init__(self, name, price, author):
        super().__init__(name, price)
        self.author = author
class Electronics(Product):
    def __init__(self, name, price, warranty_period):
        super().__init__(name, price)
        self.warranty_period = warranty_period
# Usage
book = Book("The Pragmatic Programmer", 42.15, "Andy Hunt")
laptop = Electronics("SuperLaptop", 999.99, "1 year")

```

- **Understand the Problem Domain:** The *level of abstraction should match the complexity of the problem you're solving.*
- **Avoid Over-Abstraction:** Over-abstraction occurs when you create too many layers or overly generic(natural) structures, making the code hard to understand and maintain. *If a simple function or class will do, there's no need to abstract further.*
- **Prevent Under-Abstraction:** Under-abstraction is when the code is too concrete, with repeated logic and lack of reusable components. *Identify common patterns and behaviors in your code and abstract them into functions or classes.*

Don't forget that you can refactor everything in the code no matter when and how it was written, but try your best to design a stable application during the planning stage!

## 5. Examples of good OOP designs

Let's design a simple library management system and a zoo management system applying all the knowledge we have obtained during this lesson:

### Example

```

from abc import ABC, abstractmethod
from datetime import datetime, timedelta
# Abstract class LibraryItem (Abstraction and Inheritance)
class LibraryItem(ABC):
    def __init__(self, title, author, isbn):
        self._title = title # Encapsulation: title is protected
        self._author = author # Encapsulation: author is protected
        self._isbn = isbn # Encapsulation: ISBN is protected
        self._checkout_date = None

    @abstractmethod
    def checkout(self, days=14): # Abstract method (Abstraction)
        self._checkout_date = datetime.now()
        self._return_date = self._checkout_date + timedelta(days=days)

    @abstractmethod
    def get_return_date(self): # Abstract method (Abstraction)
        if self._checkout_date:
            return self._return_date
        return "Item not checked out"

    def get_details(self): # Encapsulation: Accessing protected attributes

```

```

        return f"Title: {self._title}, Author: {self._author}, ISBN:
{self._isbn}"
# Concrete class Book inheriting from LibraryItem (Inheritance)
class Book(LibraryItem):
    def checkout(self, days=14): # Polymorphism: Method overriding
        super().checkout(days) # Call to parent method
        print(f"Book '{self._title}' checked out for {days} days.")
    def get_return_date(self): # Polymorphism: Method overriding
        return super().get_return_date()
# Concrete class DVD inheriting from LibraryItem (Inheritance)
class DVD(LibraryItem):
    def checkout(self, days=7): # Polymorphism: Method overriding with
different default days
        super().checkout(days) # Call to parent method
        print(f"DVD '{self._title}' checked out for {days} days.")
    def get_return_date(self): # Polymorphism: Method overriding
        return super().get_return_date()
# Usage (we use the list of objects here)
library_items = [
    Book("The Pragmatic Programmer", "Andy Hunt", "123456789"),
    DVD("The Matrix", "Lana Wachowski, Lilly Wachowski", "987654321")
]
for item in library_items:
    print(item.get_details())
    item.checkout()
    print(f"Return Date: {item.get_return_date()}\n")

```

## Output

```

Title: The Pragmatic Programmer, Author: Andy Hunt, ISBN: 123456789
Book 'The Pragmatic Programmer' checked out for 14 days.
Return Date: [Calculated Date]
Title: The Matrix, Author: Lana Wachowski, Lilly Wachowski, ISBN: 987654321
DVD 'The Matrix' checked out for 7 days.
Return Date: [Calculated Date]

```

## Example

```

from abc import ABC, abstractmethod
# Abstract class Animal (Abstraction and Inheritance)
class Animal(ABC):
    def __init__(self, name):
        self._name = name # Encapsulation: name is protected
    @abstractmethod
    def speak(self): # Abstract method (Abstraction)
        pass
    @abstractmethod
    def preferred_food(self): # Abstract method (Abstraction)
        pass
    def get_name(self): # Encapsulation: Accessing protected attribute
        return self._name
# Concrete class Lion inheriting from Animal (Inheritance)
class Lion(Animal):

```

```

def speak(self): # Polymorphism: Method overriding
    return f"{self.get_name()} the Lion: Roar!"
def preferred_food(self): # Polymorphism: Method overriding
    return "meat"
# Concrete class Elephant inheriting from Animal (Inheritance)
class Elephant(Animal):
    def speak(self): # Polymorphism: Method overriding
        return f"{self.get_name()} the Elephant: Trumpet!"
    def preferred_food(self): # Polymorphism: Method overriding
        return "vegetation"
# Usage
zoo_animals = [
    Lion("Leo"),
    Elephant("Ella")
]
for animal in zoo_animals:
    print(f"{animal.get_name()} speaks: {animal.speak()} and prefers {animal.preferred_food()}")

```

## Output

```
Leo speaks: Leo the Lion: Roar! and prefers meat
```

```
Ella speaks: Ella the Elephant: Trumpet! and prefers vegetation
```

Be creative with programming, try applying everything you have learnt with your custom classes and objects from the real world.

Remember that each application must have scalability, interactivity, readability and usability. And here it is, now you are ahead of 90% of programmers!

## 6. Quiz

### Question 1:

What is the output of the following code?

```

class Dog:
    def __init__(self, name):
        self.name = name
    def speak(self):
        return self.name + " says Woof!"
class Cat(Dog):
    def speak(self):
        return self.name + " says Meow!"
pet = Cat("Paws")
print(pet.speak())

```



- A) Paws says Woof!
  - B) Paws says Meow!
  - C) An error occurs
  - D) None of the above
- 

**Question 2:**

Which of the following is a principle of Object-Oriented Programming?

- A) Recursion
  - B) Polymorphism
  - C) Synchronization
  - D) Multithreading
- 

**Question 3:**

In Python, what does the `__init__` method do?

- A) Initializes a new thread
  - B) Initializes a newly created object
  - C) Acts as a destructor for an object
  - D) None of the above
- 

**Question 4:**

What does encapsulation mean in OOP?

- A) Running multiple threads concurrently
  - B) Bundling of data with methods that operate on that data
  - C) Inheriting properties from a base class
  - D) Overriding methods in the child class
- 

**Question 5:**

Which keyword is used to create a class in Python?

- A) `class`
- B) `object`
- C) `struct`
- D) `def`

---

**Question 6:**

How do you define a private attribute in a Python class?

- A) By prefixing the attribute with an underscore \_
  - B) By prefixing the attribute with two underscores \_\_
  - C) By using the `private` keyword
  - D) By defining the attribute outside of the class
- 

**Question 7:**

What is inheritance in OOP?

- A) The process by which an object acquires the properties of another object
  - B) The process by which a class acquires the properties and methods of another class
  - C) A function that a class performs
  - D) A type of function in mathematical programming
- 

**Question 8:**

What does the term "polymorphism" refer to in OOP?

- A) The ability of a function to perform different tasks based on the object
  - B) The ability of different classes to respond to the same function
  - C) The method of packing the data and functions together
  - D) The capability of a class to derive properties and characteristics from another class
- 

**Question 9:**

Which of the following best describes an abstract class in OOP?

- A) A class that cannot be instantiated and is designed to be subclassed
  - B) A class that provides a simple interface to a complex system
  - C) A template for creating objects
  - D) A class designed for efficient memory allocation
- 

**Question 10:**

In Python, how do you define an abstract method?

- A) By using the `@abstract` decorator
  - B) By prefixing the method name with `__`
  - C) By using the `@abstractmethod` decorator from the `abc` module
  - D) By declaring the method in an interface
- 

### Question 11:

Which OOP principle is illustrated by defining a method in a child class that has the same name as a method in the parent class?

- A) Encapsulation
  - B) Abstraction
  - C) Inheritance
  - D) Method Overriding
- 

### Question 12:

What is the output of the following code?

```
class A:
    def speak(self):
        return "Class A Speaks"

class B(A):
    def speak(self):
        return super().speak() + " and Class B Speaks"

obj = B()
print(obj.speak())
```

- A) Class B Speaks
  - B) Class A Speaks and Class B Speaks
  - C) Class A Speaks
  - D) An error occurs
- 

### Question 13:

Which of the following statements about static methods in Python is true?

- A) They can modify the state of an instance
- B) They cannot access or modify the class state
- C) They are declared using the `@staticmethod` decorator
- D) They must have at least one parameter

---

### Question 14:

What does the `super()` function do in Python?

- A) It returns the superclass of a class
- B) It calls a method from the parent class
- C) It initializes a superclass object

## 7. Homework

### Task 1: Pet Simulator

**Objective:** Create a virtual pet simulator where users can adopt a pet, feed it, play with it, and monitor its health and happiness.

**Requirements:**

- Implement a `Pet` class with attributes for name, hunger, happiness, and health.
- Include methods to `feed`, `play`, and `check_status` of the pet. Feeding decreases hunger, playing increases happiness, and both actions affect health.
- Create a simple user interface in the console to interact with the pet. (Try even using classes for `Menu` in the console itself)

### Task 2: Story Game

**Objective:** Develop an interactive story game where choices made by the player lead to different outcomes. Use your imagination and create the whole story storing all nodes into a list afterall.

**Requirements:**

- Design a `StoryNode` class representing each point in the story, with a narrative part and choices leading to other `StoryNodes`.
- Implement a method to display the story at the current node and let the user make a choice.
- The game progresses based on the player's decisions until reaching an ending.

### Task 3: Art Gallery

**Objective:** Implement a system to manage an art gallery, including artwork registration, artist information, and artwork sales.

**Requirements:**

- Create `Artist` and `Artwork` classes, where each `Artwork` is associated with an `Artist`.
- The `Gallery` class should manage a collection of artworks, with methods to add artwork, sell artwork, and display available artworks.
- Implement features to track the total sales and current inventory of the gallery.

- Use polymorphism, inheritance, abstraction and encapsulation to organise your code.

## Task 4: Recipe System

**Objective:** Design a system to create, store, and display recipes, including ingredients, quantities, and cooking instructions.

### Requirements

- Define `Ingredient` and `Recipe` classes, where a `Recipe` can contain multiple `Ingredients` and their quantities.
- The `Cookbook` class should aggregate multiple `Recipes`, with methods to add new recipes, find recipes by ingredient, and display all recipes.
- Enhance user interaction with features like searching for recipes based on available ingredients.

## Task 5: Fitness Tracker

**Objective:** Build a simple fitness tracker that allows users to log exercises, track workout routines, and monitor progress.

### Requirements

- Implement a `Workout` class with attributes like `date`, `exercise`, `duration`, and `intensity`.
- The `User` class should contain user details and a log of workouts, with methods to add a workout, summarize total activity, and set fitness goals.
- Provide insights to the user based on their activity logs, such as total hours worked out in a month or progress towards goals.
- In the end, it would be cool to add several methods which calculate the calories burnt during workout sessions.

# Lesson 16: Intermediate OOP

Yes, same quote again. Sorry, I have spent creativity out already :)

In object-oriented programming (OOP), composition and aggregation are two design techniques that allow you to build complex classes from simpler ones.

Both techniques enable you to model a "has-a" relationship between objects, but they differ in terms of the lifecycle and ownership of the composed objects.

## 1. Composition and Aggregation

### 1.1 Composition

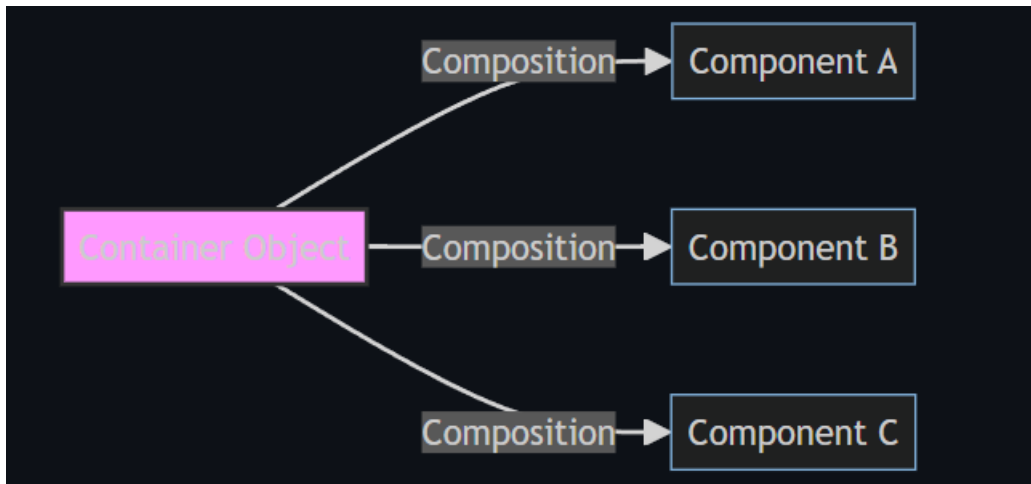
Composition is a **strong form** of association where the **composed object cannot exist independently** of the owner.

**Note:** If the owner object is destroyed, the related objects are destroyed as well.

**Example:**

Let's consider a Car class. A car is composed of an engine, tires, and a steering wheel. **These components are integral parts of the car. They don't make sense without the car.**

The basic scheme for Composition will be the following:



```
class Engine:
    def start(self):
        print("Engine started")
    def stop(self):
        print("Engine stopped")
class Tire:
    def inflate(self, pressure):
        print(f"Tire inflated to {pressure} psi")
class Car:
```

```

def __init__(self):
    """
    Note that here we are passing the instances of the ``Engine`` and
    ``Tire`` classes. It means that we can access their ``attributes/methods``
    within the ``Car`` class/
    """
    self.engine = Engine() # Composition
    self.tires = [Tire(), Tire(), Tire(), Tire()] # Composition
    print("Car created with its engine and tires")
def start(self):
    self.engine.start() # Calling (Delegating) method to the `Engine`
    print("Car started")

car = Car()
car.start()

```

## Explanation

In this example, the Car class is composed of Engine and Tire objects. The Engine and Tire objects are part of the Car's lifecycle and do not exist independently of the Car.

## Output

```

Car created with its engine and tires
Engine started
Car started

```

## Example

Let's take a look on the Computer class. A computer is composed of a CPU, a Memory, and a HardDrive.

These *components are fundamental parts of a computer and are essential for its operation.*

```

class CPU:
    def compute(self):
        print("Computing...")
class Memory:
    def load(self, data):
        print(f"Loading data: {data}")
class HardDrive:
    def read(self, address):
        print(f"Reading data from address: {address}")
class Computer:
    def __init__(self):
        self.cpu = CPU() # Composition
        self.memory = Memory() # Composition
        self.hard_drive = HardDrive() # Composition
        print("Computer assembled with CPU, Memory, and Hard Drive")
    def boot(self):
        self.memory.load("Operating System")

```

```

        self.cpu.compute()
        self.hard_drive.read("0x00")
        print("Computer booted successfully")
computer = Computer()
computer.boot()

```

## Explanation

It is a great example of **the strong association form between objects**. Once Computer is destroyed the CPU, Memory, and HardDrive become useless. As we can't use it anymore at all, they literally can't exist without each other.

## Output

```

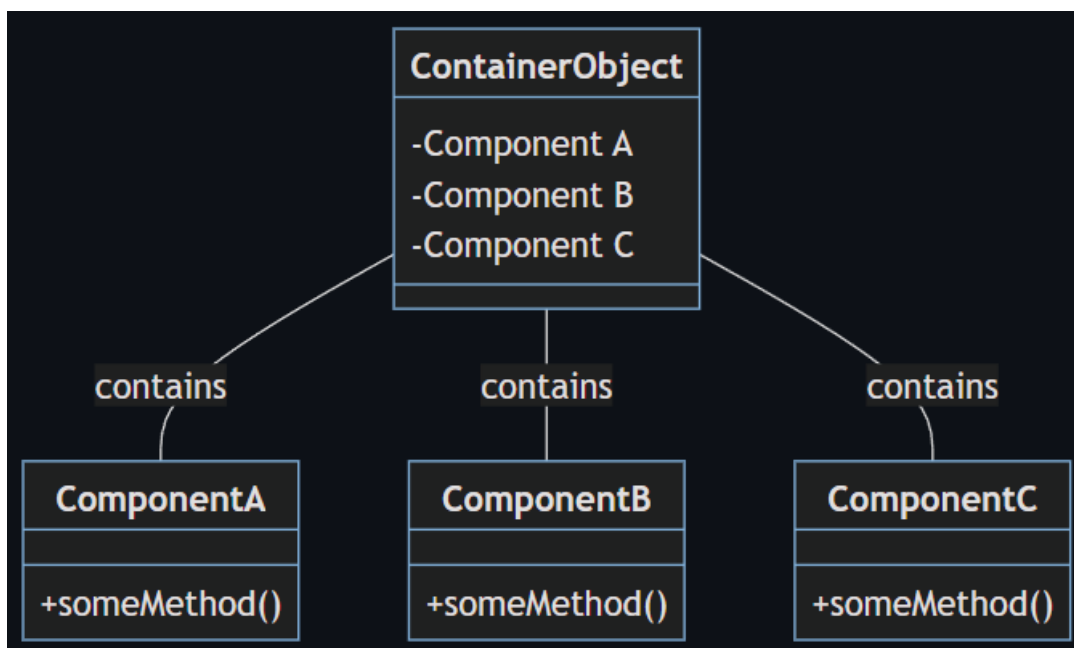
Computer assembled with CPU, Memory, and Hard Drive
Loading data: Operating System
Computing...
Reading data from address: 0x00
Computer booted successfully

```

## 1.2 Aggregation

Aggregation is a **weaker form** of association than composition. If you delete the container object, the content(objects inside the class) can live without container object.

**Note:** The same scheme applies to aggregation, but the key difference between composition and aggregations is how strong objects are associated to each other.





## Example

```
class Book:
    def __init__(self, title):
        self.title = title
class Library:
    def __init__(self):
        self.books = [] # Aggregation
    def add_book(self, book: Book):
        self.books.append(book) # book is an instance of class ``Book``
    def list_books(self):
        for book in self.books:
            print(book.title)
book1 = Book("1984")
book2 = Book("Brave New World")
library = Library()
library.add_book(book1)
library.add_book(book2)
library.list_books()
```

## Explanation

A Library contains books, but *they can exist independently of the library*.

Books can be added to or removed from the library *without affecting the existence of the books themselves*.

## Output

1984

Brave New World

## Example

```
class Professor:
    def __init__(self, name):
        self.name = name
    def teach(self):
        print(f"Professor {self.name} is teaching")
class University:
    def __init__(self):
        self.professors = [] # Aggregation
    def add_professor(self, professor):
        self.professors.append(professor)
    def start_classes(self):
        for professor in self.professors:
            professor.teach()
professor1 = Professor("John Doe")
professor2 = Professor("Jane Smith")
```

```
university = University()
university.add_professor(professor1)
university.add_professor(professor2)
university.start_classes()
```

Here, University and Professor have an aggregation relationship. Professors can exist independently of a university and can be part of multiple universities.

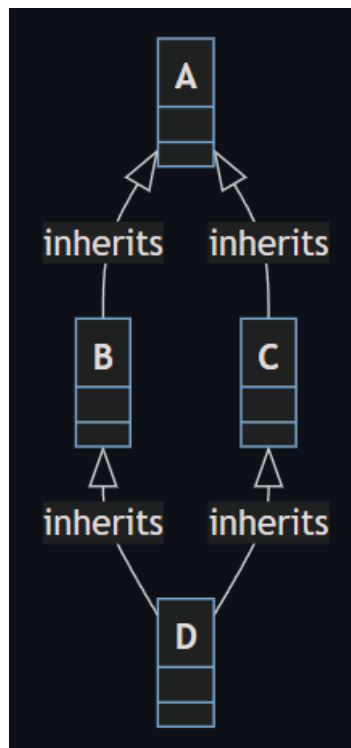
## 2. Advanced Inheritance

### 2.1 MRO (Method Resolution Order)

In simple terms, *Method Resolution Order* (MRO) **it's the sequence in which classes are searched for a method or attribute.**

The MRO **ensures that the right method is executed in the presence of inheritance especially when multiple inheritance is involved.**

Suppose we have the following hierarchy of classes:



#### Example

```
class A:
    def process(self):
        print("Process in A - The original recipe")
class B(A):
    def process(self):
        super().process()
```

```

        print("Process in B - Mom's twist to the recipe")
class C(A):
    def process(self):
        super().process()
        print("Process in C - Dad's twist to the recipe")
class D(B, C):
    def process(self):
        super().process()
        print("Process in D - Your own twist to the recipe")
d = D()
d.process()

```

The most common problem in Multiple inheritance is a Diamond Problem.

It occurs when a class inherits from two classes that both inherit from the same superclass. It can lead to ambiguity in the method resolution path.

## Output

```

Process in A - The original recipe
Process in C - Dad's twist to the recipe
Process in B - Mom's twist to the recipe
Process in D - Your own twist to the recipe

```

## Explanation

When you try to cook (`process()`) the recipe as D, Python looks at the MRO to decide how to combine these different versions.

The MRO makes sure that:

- **Step1:** Your version comes first (D): It's the most specific, just like you'd prefer your own twist to the recipe.
- **Step2:** Then it checks B and C: These are like your mom's and dad's versions. It respects the order you mention in class D (B first, then C).
- **Step3:** Finally, it checks A: The original recipe, to make sure nothing is left out.

## 2.2 How to view Method Resolution Order?

In order to view the MRO of a class, you can use the **mro** attribute or the `mro()` method.

### Example

```

print(D.__mro__)
# or
print(D.mro())

```

## Output

```
(<class '__main__.D'>, <class '__main__.B'>, <class '__main__.C'>, <class '__main__.A'>, <class 'object'>)
[<class '__main__.D'>, <class '__main__.B'>, <class '__main__.C'>, <class '__main__.A'>, <class 'object'>]
```

This way, the MRO makes sure that Python uses the methods in a sensible, predictable order, especially when your classes (like these recipe versions) are related through multiple inheritances.

It's Python's way of ensuring that the most "specific" version of your or method is used!

**IMPORTANT:** Bear in mind, that it is always the best to use composition over inheritance!

## 3. Dunder (Magic) Methods in Python

In Python, dunder methods are methods that allow instances of a class to interact with the built-in functions and operators.

To be honest it's my the most favourite feature in Python. As it is not only simplifies the interaction between instances, but give you a lot in terms of object manipulating.

Typically, they are not invoked directly, making it look like they are called by magic.

We could group dunder methods but some categories:

### 3.1 Initialization, Construction and Destruction

- `__init__(self, [...])`: Initializes a new instance of a class.
- `__new__(cls, [...])`: Creates a new instance of a class. It's a class method (called before `__init__`) that returns a new instance of the class.
- `__del__(self, [...])`: Defines behavior for object destruction. It's called when an instance's reference count reaches zero, and Python is about to destroy the object.

#### 3.1.1 `__init__()`

We have worked with `__init__()` already, it is the constructor method of a class.

It's called when an instance of the class is created. And basically we don't call it directly using `.` notation, it is done by the magic under the hood.

#### Example

```
class House:
    def __init__(self, style, num_rooms):
        self.style = style
        self.num_rooms = num_rooms
        print(f"Building a {self.style} house with {self.num_rooms} rooms")
my_house = House("Victorian", 4)
```

## Output

Building a Victorian house with 4 rooms

### 3.1.2 `__new__()`

`__new__()` is the method called *before* `__init__()`. It's responsible for returning a new instance of your class. In most cases, you don't need to override `__new__()`, but it can be useful in certain advanced scenarios, like controlling the creation of a new instance (singleton patterns) or extending immutable types.

Don't worry if you don't fully understand when and how to use `__new__()`, we will come back to it during OOP design Patterns section

### Example

Consider `__new__()` as the manufacturing process of a car. It puts together the raw materials to build a new car and then passes the newly created car to the assembly line (the `__init__()` method) for further customization like painting and installing additional parts.

```
class Car:
    _instances = []

    def __new__(cls, model):
        if model not in cls._instances:
            instance = super(Car, cls).__new__(cls)
            cls._instances.append(model)
            return instance
        else:
            print(f"{model} already exists.")
            return None

    def __init__(self, model):
        self.model = model
        print(f"Initializing {self.model}")

car1 = Car("Toyota")
car2 = Car("Honda")
car3 = Car("Toyota")  # This won't create a new instance
```

### Output:

```
Initializing Toyota

Initializing Honda

Toyota already exists.
```

### 3.3 \_\_del\_\_()

It's not meant to be an object destructor like in other programming languages, but rather a method to clean up resources if necessary.

When you need to close a file, release a lock, or close a network connection when an object is about to be destroyed, `__del__` can be quite handy.

#### Example

```
class DatabaseConnection:
    def __init__(self):
        self.connection = self.connect_to_database()
    def connect_to_database(self):
        print("Connecting to the database.")
        # Code to connect to database goes here
        return "DatabaseConnectionObject"
    def close_connection(self):
        print("Closing the connection to the database.")
        # Code to close database connection goes here
    def __del__(self):
        self.close_connection()
db_connection = DatabaseConnection()
del db_connection
```

#### Output

```
Closing the connection to the database.
```

#### Explanation

In most cases Python garbage collector does this job for us automatically, but overriding the `__del__()` can help programmer to clean up the resources, which are not needed, we don't want to have a hanging connection to the DB, unless we don't use it meanwhile, do we?

## 3.2 Representation

### 3.2.1 \_\_str\_\_()

`__str__()` should return a user-friendly string representation of your object, making it more readable and useful for print statements or any situation where the object is presented to end-users.

## Example

Consider `__str__()` as the way a user profile is displayed on a social media platform. You want it to be readable and informative.

```
class UserProfile:
    def __init__(self, username, email):
        self.username = username
        self.email = email
    def __str__(self):
        return f"UserProfile(username='{self.username}', email='{self.email}')"
user = UserProfile("john_doe", "john@example.com")
print(user)
```

## Output:

```
UserProfile(username='john_doe', email='john@example.com')
```

### 3.2.2 `__repr__()`

`__repr__()` is used to return a string that would be a valid Python expression to recreate the object. It's aimed at developers and should be as unambiguous as possible.

## Example

Consider `__repr__()` as a precise recipe for a dish. It includes exact amounts and types of ingredients, ensuring that someone can recreate the dish perfectly.

```
class Ingredient:
    def __init__(self, name, quantity):
        self.name = name
        self.quantity = quantity
    def __repr__(self):
        return f"Ingredient(name={self.name!r}, quantity={self.quantity!r})"
ingredient = Ingredient("Sugar", "100g")
print(ingredient)
```

## Output:

```
Ingredient(name='Sugar', quantity='100g')
```

As well, this methods can help you to debug your projects better, output objects in the way it is comfortable for you and improve code quality.

### 3.3 Collection/Container Emulation

Method	Syntax	Description	Example
<code>__len__(self)</code>	<code>len(obj)</code>	Returns the length of the container. Called by <code>len()</code> .	<code>len(my_collection)</code>
<code>__getitem__(self, key)</code>	<code>obj[key]</code>	Allows access to elements using the <code>self[key]</code> syntax.	<code>value = my_collection[key]</code>
<code>__setitem__(self, key, value)</code>	<code>obj[key] = value</code>	Allows setting elements using the <code>self[key] = value</code> syntax.	<code>my_collection[key] = value</code>
<code>__delitem__(self, key)</code>	<code>del obj[key]</code>	Allows deleting elements using the <code>del self[key]</code> syntax.	<code>del my_collection[key]</code>
<code>__contains__(self, item)</code>	<code>item in obj</code>	Checks if the container contains <code>item</code> . Called by <code>in</code> .	<code>if item in my_collection:</code>

The class `Menu` demonstrates the sort of a storage for items, and as we know already `dict` methods are perfect for storage managing and handling.

#### Example

```
class Menu:
    def __init__(self):
        self.items = {}
    def __setitem__(self, item, price):
        self.items[item] = price
    def __getitem__(self, item):
        return self.items.get(item, f"{item} not available")
    def __delitem__(self, item):
        if item in self.items:
            del self.items[item]
        else:
            print(f"{item} not found in the menu")
    def __len__(self):
        return len(self.items)
    def __contains__(self, item):
```



```

        return item in self.items
    def __str__(self):
        return '\n'.join(f"{item}:    ${price}"    for    item,    price    in
self.items.items())
    def __repr__(self):
        return f"Menu({self.items})"
# Usage
my_menu = Menu()
my_menu["Coffee"] = 2.99    # calling ``__setitem__``
my_menu["Tea"] = 1.99
print(my_menu.items)
print(my_menu["Coffee"])    # calling ``__getitem__``
print(my_menu["Espresso"])
print(len(my_menu))    # calling ``__len__``
print("Coffee" in my_menu)    # calling ``__contains__``
print("Espresso" in my_menu)
del my_menu["Tea"]    # calling ``__delitem__``
print(my_menu.items)
print(str(my_menu))    # calling ``__str__``
print(repr(my_menu))    # calling ``__repr__``

```

## Output

```

{'Coffee': 2.99, 'Tea': 1.99}
2.99
Espresso not available
2
True
False
{'Coffee': 2.99}
Coffee: $2.99
Menu({'Coffee': 2.99})

```

## 4. Comparison and Arithmetic Operators

### 4.1 Comparison

Method	Syntax	Description	Example
<code>__eq__(self, other)</code>	<code>obj1 == obj2</code>	Defines behavior for the == operator	<code>if obj1 == obj2:</code>
<code>__ne__(self, other)</code>	<code>obj1 != obj2</code>	Defines behavior for the != operator	<code>if obj1 != obj2:</code>
<code>__lt__(self, other)</code>	<code>obj1 &lt; obj2</code>	Defines behavior for the < operator	<code>if obj1 &lt; obj2:</code>
<code>__le__(self, other)</code>	<code>obj1 &lt;= obj2</code>	Defines behavior for the <= operator	<code>if obj1 &lt;= obj2:</code>

Method	Syntax	Description	Example
<code>__gt__(self, other)</code>	<code>obj1 &gt; obj2</code>	Defines behavior for the > operator	<code>if obj1 &gt; obj2:</code>
<code>__ge__(self, other)</code>	<code>obj1 &gt;= obj2</code>	Defines behavior for the >= operator	<code>if obj1 &gt;= obj2:</code>

Imagine you're running an apple store.

Apples can differ in weight, type, and ripeness. It would be helpful to compare apples based on these attributes, add their weights, or even get a combined ripeness score.

This is comparison dunder methods come into play, allowing you (an owner) to define how apples should be compared or combined.

### Example

```
class Apple:
    def __init__(self, weight, type, ripeness):
        self.weight = weight
        self.type = type
        self.ripeness = ripeness # 1 to 10 scale
    # Use `str` for easy debugging
    def __str__(self):
        return f"{self.type} apple, Weight: {self.weight}g, Ripeness: {self.ripeness}/10"

    # (==) Two apples are considered equal if they have the same type and ripeness
    def __eq__(self, other):
        return self.type == other.type and self.ripeness == other.ripeness
    # (!=) Defined by the type or ripeness
    def __ne__(self, other):
        return not (self == other)
    # (<): An apple is considered less than another if it's less ripe or, lighter
    def __lt__(self, other):
        if self.ripeness == other.ripeness:
            return self.weight < other.weight
        return self.ripeness < other.ripeness
    # (<=) Combination of less than and equality
    def __le__(self, other):
        return self < other or self == other
    # (>) Opposite of less than
    def __gt__(self, other):
        return not (self <= other)
    # (>=) Combination of greater than and equality
    def __ge__(self, other):
        return not (self < other)

apple1 = Apple(150, "Golden", 7)
apple2 = Apple(200, "Golden", 8)
print(apple1 < apple2)
```

```

print(apple2 > apple1)
print(apple2 >= apple1)
print(apple2 <= apple1)
# And so on..

```

## Output

```

True
True
True
False

```

## 4.2 Arithmetic

Method	Syntax	Description	Example
<code>__add__(self, other)</code>	<code>obj1 + obj2</code>	Defines behavior for the + operator.	<code>result = obj1 + obj2</code>
<code>__sub__(self, other)</code>	<code>obj1 - obj2</code>	Defines behavior for the - operator.	<code>result = obj1 - obj2</code>
<code>__mul__(self, other)</code>	<code>obj1 * obj2</code>	Defines behavior for the * operator.	<code>result = obj1 * obj2</code>
<code>__truediv__(self, other)</code>	<code>obj1 / obj2</code>	Defines behavior for the / operator.	<code>result = obj1 / obj2</code>

Imagine you started living with your partner, and agreed on the common family budget. Sasha earns less than Yehor but she wants his money in the wallet. She has a wallet with some money as well, how do we combine those wallets?

Let's take a look on the specific class `Wallet` with some amount of money in it.

## Example

```

class Wallet:
    def __init__(self, amount):
        self.amount = amount
    def __str__(self):
        return f"${self.amount}"
    # Add money to the wallet
    def __add__(self, other):
        if isinstance(other, Wallet):
            return Wallet(self.amount + other.amount)
        else:
            return Wallet(self.amount + other)
    # Subtract money from the wallet
    def __sub__(self, other):
        if isinstance(other, Wallet):

```

```

        return Wallet(self.amount - other.amount)
    else:
        return Wallet(self.amount - other)
# Equality - Check if the amount in two wallets is the same
    def __eq__(self, other):
        return self.amount == other.amount
# Examples of using the class
wallet1 = Wallet(100)
wallet2 = Wallet(50)
# Add two wallets together
new_wallet = wallet1 + wallet2
print(new_wallet)
# Add a fixed amount to a wallet
updated_wallet = wallet1 + 20
print(updated_wallet)
remaining_balance = wallet1 - wallet2
print(remaining_balance)
# Check if two wallets have the same amount
print(wallet1 == wallet2)

```

## Output

```

$150
$120
$50
False

```

After dunder methods are implemented, we can manage finances in a very intuitive way (I would even say in a more understandable way for human kind).

## 5. Attribute Access

Attribute access methods provide a way to manage how attributes are accessed, assigned, or deleted in your class. They offer a level of control over your class's attributes, allowing for validation, logging, or other custom behaviors.

Method	Syntax	Description	Example
<code>__getattr__(self, name)</code>	<code>obj.name</code>	Defines behavior when an attribute is accessed, and	Accessing <code>obj.undefined_attr</code> calls <code>__getattr__('undefined_attr')</code>

Method	Syntax	Description	Example
		it's not found in the instance's dictionary.	
<code>__setattr__(self, name, value)</code>	<code>obj.name = value</code>	Called when an attribute assignment is attempted.	<code>obj.attr = value</code> triggers <code>__setattr__('attr', value)</code>
<code>__delattr__(self, name)</code>	<code>del obj.name</code>	Called when an attribute deletion is attempted.	<code>del obj.attr</code> calls <code>__delattr__('attr')</code>

#### 4.1 `__getattr__(self, name)`

`__getattr__` is called when an attribute is not found in an object's instance dictionary. It's useful for implementing a default behavior for missing attributes or for creating proxy objects.

#### Example: Default Attributes in a Configuration

##### Example

```
class Configuration:
    def __init__(self):
        self.settings = {"theme": "Light", "language": "English"}
    def __getattr__(self, name):
        # Return a default value if the setting is not found
        return f"Setting '{name}' not found. Using default value."

config = Configuration()
print(config.theme)           # calling ``__getattr__()`` under the hood
print(config.language)
```

```
print(config.font_size)
```

## Output

```
Setting 'theme' not found. Using default value.  
Setting 'language' not found. Using default value.  
Setting 'font_size' not found. Using default value
```

### 4.2 `__setattr__(self, name, value)`

`__setattr__` is invoked when an attribute assignment is made. It can be used to enforce constraints, perform validations, or automatically update related attributes.

#### Example: Validating and Logging Attribute Changes

```
class Product:
    def __init__(self, name, price):
        self.name = name
        self.price = price
    def __setattr__(self, name, value):
        if name == "price" and value < 0:
            raise ValueError("Price cannot be negative.")
        print(f"Setting {name} to {value}")
        super().__setattr__(name, value)
product = Product("Coffee", 5)      # calling ``__setattr__()`` method
product.price = 6
product.quantity = 10
```

## Output

```
Setting name to Coffee
Setting price to 5
Setting price to 6
Setting quantity to 10
```

### 4.3 `__delattr__(self, name)`

`__delattr__` is called when an attribute deletion is attempted. It allows you to define behavior when attributes are deleted, such as preventing the deletion of certain attributes or performing cleanup operations.

#### Example: Preventing Deletion of Critical Attributes

```
class ProtectedAttributes:
    def __init__(self):
        self._critical_data = "Sensitive Info"
    def __delattr__(self, name):
        if name == "_critical_data":
            raise AttributeError("Deletion of _critical_data is not allowed.")
```

```

        super().__delattr__(name)
protected_obj = ProtectedAttributes()
del protected_obj._critical_data           # calling ``__delattr__()`` method

```

## Output

```

Traceback (most recent call last):
  File "<string>", line 11, in <module>
  File "<string>", line 7, in __delattr__
AttributeError: Deletion of _critical_data is not allowed.

```

## Explanation

**Note:** You can notice that it is very similar to `@property`, in fact `property` calls these methods under the hood and simplifies E2E interaction with a code for a programmer.

**Note:** You must be extremely careful implementing these attribute access methods, you can provide a more controlled and secure way to manage how attributes are accessed, assigned, or deleted in your classes. This can lead to bugs hard to debug in some cases, but if it's used wisely, you get an extremely powerful tool to manage the class attributes directly.

There are some more useful dunder methods such as `__iter__()` `__call__()` or `__enter__()` and `__exit__()`, but it will be described within design patterns and context managers sections, so keep going to master Python skills!

# 4. Enums

Enums is another way to structure your code and organize sets of related constants. It's a great instrument to use when you have constants which remain the same within your application structured within one class.

## 4.1 Syntax

Enums are defined by *subclassing the Enum class*. Each member of the enum has a name and a value.

## Example

```

from enum import Enum
class Color(Enum):
    RED = 1
    GREEN = 2
    BLUE = 3
# You can access them either by name or by value
color = Color.RED           # by name
print(color)
color = Color(1)            # by value
print(color)

```

## Output

```
Color.RED
```

```
Color.RED
```

Each member of an `enum` has two main properties:

- `name`: The name of the member (e.g., `'RED'`).
- `value`: The value associated with the member (1).

### Example

```
# Exten the code above with the following:  
color = Color.RED  
print(color.name)    # Output: RED  
print(color.value)   # Output: 1
```

### Output

```
RED
```

```
1
```

As well , if we take a look inside `enums` and call `dir()` function we can see that it has some methods and attributes which will come in handy.

### Example

```
print(dir(Color))
```

### Output

```
['BLUE', 'GREEN', 'RED', '__class__', '__contains__', '__doc__', '__getitem__',  
 '__init_subclass__', '__iter__', '__len__', '__members__', '__module__',  
 '__name__', '__qualname__']
```

Try and play around `enums`, you will find them much usefull than dictionaries we used before to store contents into. Especially with `isinstance()` and `type()` functions.

## 5. Quiz

### Question 1:

In OOP, what distinguishes Composition from Aggregation?



- A) Composition allows for dynamic changes to the relationship, whereas Aggregation is static.
- B) Aggregation implies ownership and the lifecycle of the contained objects is tied to the container object.
- C) Composition implies ownership and the lifecycle of the contained objects is tied to the container object.
- D) Aggregation is a "use-a" relationship, while Composition is a "has-a" relationship.

### Question 2:

What does Method Resolution Order (MRO) in Python solve?

- A) It determines the order in which base classes are traversed when executing a method.
- B) It optimizes the code for faster execution of methods.
- C) It ensures that the correct constructor is called for creating an object.
- D) It provides a mechanism for multiple inheritance of attributes.

### Question 3:

How does Python's `__new__` method differ from `__init__`?

- A) `__new__` method is used to instantiate a new object, whereas `__init__` is used to initialize the object.
- B) `__init__` creates a new instance and `__new__` initializes the created instance.
- C) `__new__` method can return an instance of another class, while `__init__` cannot.
- D) There is no difference; `__new__` and `__init__` can be used interchangeably.

### Question 4:

Which method in Python is called when an object is about to be destroyed?

- A) `__delete__`
- B) `__remove__`
- C) `__destroy__`
- D) `__del__`

### Question 5:

What is the purpose of `__repr__` in Python?

- A) To return a machine-readable representation of an object.
- B) To output a string that allows `eval()` to recreate the object.
- C) To represent the memory address of the object.
- D) To provide a pretty-print functionality for debugging purposes.

### Question 6:

In Python, which of the following is a use-case for the `__setattr__` method?

- A) To intercept attribute access.
- B) To prevent the modification of attributes.
- C) To log attribute changes.
- D) All of the above.

### Question 7:

What is the primary benefit of using Enums in Python?

- A) They provide a namespace for a set of identifiers.
- B) They allow the creation of unique constants that can be compared by identity.
- C) They serve as a drop-in replacement for dictionaries.
- D) They enable arithmetic operations on defined constants.

### Question 8:

Which problem does the C3 linearization algorithm in Python's MRO address?

- A) The "Superclass Dilemma" problem.
- B) The "Diamond Inheritance" problem.
- C) The "Circular Dependency" problem.
- D) The "Multiple Constructors" problem.

### Question 9:

What is the purpose of `__getitem__` in Python?

- A) It allows an object to behave like a function.
- B) It allows an object to be indexed like a list or dictionary.
- C) It allows the dynamic creation of new attributes.
- D) It allows an object to be called as an iterator.

### Question 10:

Which statement accurately describes the `__str__` and `__repr__` methods in Python?

- A) `__str__` is used for an informal string representation of an object, while `__repr__` is formal.
- B) `__str__` can only be used for numeric data, while `__repr__` is for textual data.
- C) `__repr__` is used for debugging and development, while `__str__` is for end-user display purposes.
- D) `__repr__` returns a string that is readable by the interpreter, while `__str__` does not.

## 6. Homework

### Task 1: Car Assembly

**Objective:** Simulate a car assembly process using composition where various parts come together to form a complete car.

## Requirements:

- Define classes `Engine`, `Wheel`, `Body`, and any other components you find necessary.
- The `Car` class should be composed of these objects, and it can't exist without them.
- Methods within the `Car` class should delegate responsibilities to its components, like `start_engine` or `inflate_tires`.
- The program should allow creating a `Car` object with all its components.

## Task 2: Order System

**Objective:** Build an e-commerce order system where orders are composed of items, but the lifecycle of items is independent of the order.

## Requirements:

- Define an `Item` class with properties like `name`, `price`, and `quantity`.
- The `Order` class should be composed of `Item` objects and include methods to add items, calculate the total, and finalize the purchase.
- Items should be able to exist without being part of an order, indicating aggregation.
- Users should be able to create an order, add items, and see the order total.
- Users can also create standalone items that are not part of any order.

# Lesson 17: SOLID

"SOLID principles are the compass that navigates developers through the complexities of software architecture"

## 0. Definition

SOLID principles are a set of design principles in software engineering that, when followed properly, make the software more understandable, flexible, and maintainable. The acronym SOLID stands for five design principles:

1. **S** - Single Responsibility Principle (SRP)
2. **O** - Open/Closed Principle (OCP)
3. **L** - Liskov Substitution Principle (LSP)
4. **I** - Interface Segregation Principle (ISP)
5. **D** - Dependency Inversion Principle (DIP)

## 1. Single Responsibility Principle (SRP)

### 1.1 Definition

A class should have one, and only one, reason to change. **This means a class should only *have one job or responsibility*.**

#### Example

Consider a Report class that generates a report and then prints it. According to SRP, these two actions should be separated into different classes.

```
# Violating SRP
class Report:
    def generate_report(self, data):
        # Code to generate the report
        pass

    def print_report(self, report):
        # Code to print the report
        pass

# Adhering to SRP
class ReportGenerator:
    def generate_report(self, data):
        # Code to generate the report
        pass

class ReportPrinter:
    def print_report(self, report):
        # Code to print the report
        pass
```

Again, each class should have its own purpose, so the best to write down functionality/architecture of the project on the paper and stick to that in advance.

## Example

Consider a `UserManagement` class that handles user-related operations such as

- Adding a user
- Sending a welcome email
- Save (logging) the activity of the user

## Incorrect Example

```
class UserManagement:
    def add_user(self, user):
        # Code to add the user to the system
        pass
    def send_welcome_email(self, user):
        # Code to send a welcome email to the user
        pass
    def log_activity(self, activity):
        # Code to log user activity
        pass
```

Think about how we can refactor this, does this class serve its purpose? Is it correct that it handles lots of different operations? Can we refactor it to improve maintainability/readability and scalability?

The answer is that each method is not really related to the `UserManagement` activity, it has 3 different purposes which could be separated into other classes:

## Correct Example

```
class UserRegistry:
    def add_user(self, user):
        # Code to add the user to the system
        pass
class EmailService:
    def send_welcome_email(self, user):
        # Code to send a welcome email to the user
        pass
class ActivityLogger:
    def log_activity(self, activity):
        # Code to log user activity
        pass
```

The code above ensures that *each class has a single reason to change*. Which means that we applied SRP successfully.

## 2. Open/Closed Principle (OCP)

Software entities (like classes, modules, functions) **should be open for extension** but **closed for modification**.

This means that the behavior of a module/class/function can be extended without modifying its source code.

### Example

Let's consider an example of a `TaxCalculator` class that calculates tax based on the type of product. Initially, the class is not following OCP, because *every time a new tax type is introduced, the class has to be modified*.

### Correct Example

```
class TaxCalculator:
    def calculate_tax(self, product_type, price):
        if product_type == "book":
            return price * 0.05 # 5% tax for books
        elif product_type == "food":
            return price * 0.10 # 10% tax for food
        # More conditions for other product types
```

### Incorrect Example

To follow the OCP, we can define a generic `TaxCalculator` class and then extend it for each specific tax type.

```
class TaxCalculator:
    def calculate_tax(self, price):
        pass
class BookTaxCalculator(TaxCalculator):
    def calculate_tax(self, price):
        return price * 0.05
class FoodTaxCalculator(TaxCalculator):
    def calculate_tax(self, price):
        return price * 0.10
# More classes for other product types
...
```

### Example

Let's consider a reporting system where reports can be generated in different formats (e.g., PDF, CSV). Initially, the system is not following OCP because *adding a new report format requires modifying existing code*.

## Incorrect Example

```
class ReportGenerator:
    def generate_report(self, data, format_type):
        if format_type == "PDF":
            # Generate PDF report
            pass
        elif format_type == "CSV":
            # Generate CSV report
            pass
        # More conditions for other formats
```

## Correct Example

Here, we define a generic `ReportGenerator` class and then extend it for each specific report format.

```
class ReportGenerator:
    def generate_report(self, data):
        pass

class PDFReportGenerator(ReportGenerator):
    def generate_report(self, data):
        # Generate PDF report logic
        pass

class CSVReportGenerator(ReportGenerator):
    def generate_report(self, data):
        # Generate CSV report logic
        pass

# More classes for other formats
```

Sometimes people tend to over-engineer the solution by introducing too many layers of abstraction or making the system too generic.

Yes, SOLID principles tend to be a great foundation but please, don't overthink the design of your application, the majority of code can easily be refactored and updated throughout your journey, JUST CODE!

## 3. Liskov Substitution Principle (LSP)

Objects of a superclass should be replaceable with objects of its subclasses *without affecting the correctness of the program*.

### Example

Consider a system with different types of shapes where each shape can calculate its area.

A violation of LSP would occur if a subclass of Shape does not correctly implement the `area()` method.

### Incorrect

```
class Shape:
    def area(self):
        pass
class Rectangle(Shape):
    def __init__(self, width, height):
        self.width = width
        self.height = height
    def area(self):
        return self.width * self.height
class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius
    def area(self):
        # Incorrect implementation or not implemented
        pass
```

It can lead to incorrect behavior when a Circle is used in place of a Shape.

### Correct

```
class Shape:
    def area(self):
        pass
class Rectangle(Shape):
    def __init__(self, width, height):
        self.width = width
        self.height = height
    def area(self):
        return self.width * self.height
class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius
    def area(self):
        return 3.14 * self.radius * self.radius
```

Using this principle can be a challenge in the begging of your application design, but if you use it within a correct approach it can be very predictable and robust for further development

### Example

Let's consider a system with different types of birds. Initially, each bird has a `fly()` method.

A violation of LSP would occur if a subclass of Bird (like Penguin) **cannot correctly implement the `fly()` method.**



## Incorrect

```
class Bird:
    def fly(self):
        # Default fly behavior
        pass
class Sparrow(Bird):
    def fly(self):
        # Implementation for flying
        pass
class Penguin(Bird):
    def fly(self):
        # Penguins can't fly!
        raise Exception("Can't fly")
```

In complex systems `fly()` would lead to incorrect behavior or runtime errors which potentially could be hard to debug.

This can be refactored by creating *separate interfaces* for `FlyingBird` and `NonFlyingBird` to ensure consistency throughout the code and correct implementation of overridden methods.

## Correct

```
class Bird:
    # Common bird behavior (if any)
    pass
class FlyingBird(Bird):
    def fly(self):
        # Implementation for flying
        pass
class NonFlyingBird(Bird):
    # Other behaviors specific to non-flying birds
    pass
class Sparrow(FlyingBird):
    def fly(self):
        # Sparrow-specific flying behavior
        pass
class Penguin(NonFlyingBird):
    # Penguin-specific behaviors
    pass
```

Penguin objects *are not expected to fly*, and thus, the system does not assume that capability, adhering to LSP.

There are lots of rabbit holes for LSP but generally some static linters as `mypy` handle all of SOLID this for developers, but don't forget to turn on the brain while designing your application ;)

## 4. Interface Segregation Principle (ISP)

The Interface Segregation Principle (ISP) states that *no client should be forced to depend on methods it does not use*.

This principle aims to split larger interfaces into smaller, more specific ones so that clients only need to know about the methods that are interesting to them.

## Example

Let's consider a printing system where the initial design forces the `Printer` class to implement functions that are not essential to all types of printers, such as faxing or scanning.

## Incorrect

```
class AllInOnePrinter:
    def print_document(self, document):
        # Print the document
        pass
    def scan_document(self, document):
        # Scan the document
        pass
    def fax_document(self, document):
        # Fax the document
        pass
```

In this example, even simple printers without scanning or faxing capability have to implement the `scan_document` and `fax_document` methods, which violates ISP.

## Correct

We refactor the example by *creating separate interfaces for each responsibility*.

```
class Printer:
    def print_document(self, document):
        pass
class Scanner:
    def scan_document(self, document):
        pass
class FaxMachine:
    def fax_document(self, document):
        pass
# Yes! That's where multiple inheritance comes in use
class AllInOneMachine(Printer, Scanner, FaxMachine):
    def print_document(self, document):
        # Print the document
        pass
    def scan_document(self, document):
        # Scan the document
        pass
    def fax_document(self, document):
        # Fax the document
        pass
```

```

# Create instances
simple_printer = Printer()
scanner = Scanner()
fax_machine = FaxMachine()
# Using individual component for each class
document = "This is a document."
simple_printer.print_document(document)      # ``Printer``: Printing document
scanner.scan_document(document)             # ``Scanner``: Scanning document
fax_machine.fax_document(document)          # ``FaxMachine``: Faxing document
# All in one can handle three different operations in case we need to use all
# of them
all_in_one = AllInOneMachine()
all_in_one.print_document(document)
all_in_one.scan_document(document)
all_in_one.fax_document(document)

```

It might be hard to understand but the simple explanation will be the following:

- `Printer`, `Scanner`, and `FaxMachine` are interfaces (or abstract classes) that define **specific functionalities**. (That is a great example of SRP as well)
- `AllInOneMachine` implements all these interfaces, providing the functionality for printing, scanning, and faxing.

**Note:** Clients that only need a printer can depend on the `Printer` interface without being forced to know about scanning or faxing and same applies to `Scanner` and `FaxMachine`.

If you need to print, you use `Printer`, to fax `FaxMachine`, and to scan `Scanner` is the way to go!

## Example

Consider a vehicle control system where the initial design forces all vehicle types to implement functionalities that are not essential for all of them, such as `launchMissiles` for military vehicles or `playMusic` for civilian vehicles.

We don't want to have the ability to launch missiles in our car we use day to day for commuting to work, do we?

## Incorrect

```

class VehicleControl:
    def steerLeft(self):
        # Steer the vehicle left
        pass
    def steerRight(self):
        # Steer the vehicle right
        pass
    def launchMissiles(self):
        # Launch missiles (mainly for military vehicles)
        pass
    def playMusic(self):

```

```
# Play music (mainly for civilian vehicles)
pass
```

## Correct

We refactor the example above by *creating separate interfaces for each category of functionalities*.

```
# Parental Inerfaces (Base classes)
class BasicVehicleOperations:
    def steer(self):
        # Steer the vehicle
        pass
class MilitaryOperations:
    def launchMissiles(self):
        # Launch missiles
        pass
class EntertainmentOperations:
    def playMusic(self):
        # Play music
        pass
class CivilianVehicle(BasicVehicleOperations, EntertainmentOperations):
    def steer(self):
        # Implementation for steering
        pass
    def playMusic(self):
        # Implementation to play music
        pass
class MilitaryVehicle(BasicVehicleOperations, MilitaryOperations):
    def steer(self):
        # Implementation for steering
        pass
    def launchMissiles(self):
        # Implementation to launch missiles
        pass
```

Sometimes applying ISP can might result in a large number of interfaces, each with only a few methods. And this can become a headache for developers, but on the other hand, smaller and more specific interfaces lead to more modular and understandable code.

## 5. Dependency Inversion Principle (DIP)

The Dependency Inversion Principle (DIP) suggests that high-level modules *should not* depend on low-level modules.

*Abstractions should not depend on details, but details should depend on abstractions.*

### Example

Imagine you have a news reporting system where a `NewsReporter` class is responsible for reporting news. Initially, it directly uses a `RadioChannel` class to broadcast news. We want to adhere to DIP to make our `NewsReporter` more flexible and not tightly coupled to the `RadioChannel`.

### Incorrect

```
class RadioChannel:
    def broadcast_news(self, news):
        print(f"Broadcasting news on the radio: {news}")
class NewsReporter:
    def __init__(self):
        self.radio_channel = RadioChannel()
    def report_news(self, news):
        self.radio_channel.broadcast_news(news)
```

In this example, `NewsReporter` is directly dependent on `RadioChannel`, meaning if we want to broadcast news on a different medium, like TV, we'd have to modify the `NewsReporter` class, violating DIP.

### Correct

Let's introduce an abstraction (interface) named `BroadcastMedium` and make `NewsReporter` *dependent on this interface rather than a concrete class*.

```
class BroadcastMedium:
    def broadcast_news(self, news):
        pass
class RadioChannel(BroadcastMedium):
    def broadcast_news(self, news):
        print(f"Broadcasting news on the radio: {news}")
class TVChannel(BroadcastMedium):
    def broadcast_news(self, news):
        print(f"Broadcasting news on TV: {news}")
class NewsReporter:
    def __init__(self, broadcast_medium: BroadcastMedium):
        self.broadcast_medium = broadcast_medium
    def report_news(self, news):
        self.broadcast_medium.broadcast_news(news)
```

Now, `NewsReporter` relies on the `BroadcastMedium` interface. We can easily broadcast news on the radio, TV, or any other medium that implements the `BroadcastMedium` interface without changing the `NewsReporter` class.

Since the main idea is that both high and low level modules should depend on abstractions. It is a great when example we don't care of the implementation details of `broadcast_news` in `BroadcastMedium` class, we just call it.

### Example

Let's say we have a book reading app where a `BookReader` class is responsible for reading books. Initially, it's directly using a `PDFReader` class. We'll apply DIP *to make `BookReader` flexible and not dependent on the `PDFReader`.*

## Incorrect

```
class PDFReader:
    def read_book(self, book):
        print(f"Reading {book} in PDF format.")
class BookReader:
    def __init__(self):
        self.reader = PDFReader()
    def read_book(self, book):
        self.reader.read_book(book)
```

In this example, `BookReader` is directly dependent on `PDFReader`. If we want to read books in a different format, we'd need to change `BookReader`, violating DIP.

## Correct

Introduce an abstraction (interface) named `BookFormatReader` and *make `BookReader` dependent on this interface.*

```
class BookFormatReader:
    def read_book(self, book):
        pass
class PDFReader(BookFormatReader):
    def read_book(self, book):
        print(f"Reading {book} in PDF format.")
class EpubReader(BookFormatReader):
    def read_book(self, book):
        print(f"Reading {book} in EPUB format.")
class BookReader:
    def __init__(self, format_reader: BookFormatReader):
        self.format_reader = format_reader
    def read_book(self, book):
        self.format_reader.read_book(book)           # Use the specific format of
the reader
```

Now, `BookReader` depends on the `BookFormatReader` interface. We can easily read books in PDF, EPUB, or any other format that implements the `BookFormatReader` interface *without changing the `BookReader` class.*

## 6. Summarise

I want to share a table with you to which you could refer to during designing your application. It might be not extremely readable, but helpful anyway to bear in mind summarising the key principles of SOLID and all rabbit holes you can encounter.

Principle	Purpose	Advantages	Potential Disadvantages
<b>Single Responsibility Principle (SRP)</b>	Ensure a class has only one reason to change.	<ul style="list-style-type: none"> <li>- Easier maintenance</li> <li>- Increased modularity</li> <li>- Improved readability</li> </ul>	<ul style="list-style-type: none"> <li>- May lead to many small, tightly coupled classes</li> </ul>
<b>Open/Closed Principle (OCP)</b>	Allow entities to be open for extension but closed for modification.	<ul style="list-style-type: none"> <li>- Flexibility in extension</li> <li>- Protection against changes</li> <li>- Reduced risk of bugs</li> </ul>	<ul style="list-style-type: none"> <li>- May introduce abstract layers</li> <li>- Can lead to over-engineering</li> </ul>
<b>Liskov Substitution Principle (LSP)</b>	Subtypes must be substitutable for their base types.	<ul style="list-style-type: none"> <li>- Enhanced reliability</li> <li>- Promotes consistency</li> <li>- Better code reusability</li> </ul>	<ul style="list-style-type: none"> <li>- Restricts how inheritance is used</li> <li>- Can make hierarchy design complex</li> </ul>
<b>Interface Segregation Principle (ISP)</b>	No client should be forced to depend on methods it doesn't use.	<ul style="list-style-type: none"> <li>- Decoupled system</li> <li>- Increased cohesion</li> <li>- Easier to understand interfaces</li> </ul>	<ul style="list-style-type: none"> <li>- May increase the number of interfaces</li> <li>- Potential for duplicate methods</li> </ul>
<b>Dependency Inversion Principle (DIP)</b>	High-level modules should not depend on low-level modules.	<ul style="list-style-type: none"> <li>- Decoupled architecture</li> <li>- Easier to refactor and test</li> <li>- Promotes flexible system</li> </ul>	<ul style="list-style-type: none"> <li>- Increased complexity</li> <li>- Indirect relations between components</li> </ul>

## 7. Let's Refactor!

Suppose we want to create an app that manages and displays messages in various formats.

## Example

```
class MessageManager:
    def __init__(self, content):
        self.content = content
    def format_message(self, format_type):
        if format_type == "JSON":
            return f'{{"message": "{self.content}"}}'
        elif format_type == "XML":
            return f"<message>{self.content}</message>"
    def display_message(self, format_type):
        formatted_message = self.format_message(format_type)
        print(formatted_message)

# Usage
message_manager = MessageManager("Hello, SOLID!")
message_manager.display_message("JSON")
```

In order to refactor our application we should use the following steps:

**Step 1:** Go throughout the code and identify which principle is violated?

```
class MessageManager:
    def __init__(self, content):
        self.content = content
    # SRP Violation: This class handles multiple responsibilities
    def format_message(self, format_type):
        if format_type == "JSON":
            return f'{{"message": "{self.content}"}}'
        elif format_type == "XML":
            return f"<message>{self.content}</message>"
    # OCP Violation: Modifying the class to add a new format
    # LSP Violation: Subclasses would find it hard to support all format types
    # ISP Violation: Clients that need only message formatting are forced to
    # depend on display functionality too
    def display_message(self, format_type):
        formatted_message = self.format_message(format_type)
        print(formatted_message)

message_manager = MessageManager("Hello, SOLID!")
message_manager.display_message("JSON")
```

## Explanation

1. **Single Responsibility Principle (SRP) Violation:** The MessageManager class handles multiple responsibilities, including formatting and displaying messages.
2. **Open/Closed Principle (OCP) Violation:** To support a new message format, the format\_message method needs to be modified, violating the principle of open for extension, closed for modification.
3. **Interface Segregation Principle (ISP) Violation:** Clients that only want message formatting are forced to depend on the display functionality.



4. **Dependency Inversion Principle (DIP) Violation:** The `MessageManager` class has a concrete dependency on message formatting and display, making it inflexible.

**Step 2:** Address each issue separately

A) Separate Responsibilities (SRP)

```
class Message:
    def __init__(self, content):
        self.content = content
```

### Explanation

- SRP: Single responsibility for managing message content.

B) Create Formatter Interface and Implementations (OCP, LSP, ISP)

```
from abc import ABC, abstractmethod
class MessageFormatter(ABC):
    @abstractmethod
    def format(self, message):
        pass
class JSONFormatter(MessageFormatter):
    def format(self, message):
        return f'{{"message": "{message.content}"}}'
class XMLFormatter(MessageFormatter):
    def format(self, message):
        return f"<message>{message.content}</message>"
```

### Explanation

- OCP: Open for extension, closed for modification
- LSP: Subtypes can be substituted without altering the correctness of the program
- ISP: Clients can choose specific formatters they need without depending on unused methods

C) Implement Display Functionality

```
class MessageDisplayer:
    def __init__(self, formatter: MessageFormatter):
        self.formatter = formatter
    def display(self, message: Message):
        formatted_message = self.formatter.format(message)
        print(formatted_message)
```

### Explanation

DIP: High-level `MessageDisplayer` depends on abstraction `MessageFormatter`, not on concrete implementations!

### Step 3: Put the code altogether

#### Refactored

```
from abc import ABC, abstractmethod

class Message:
    def __init__(self, content):
        self.content = content

class MessageFormatter(ABC):
    @abstractmethod
    def format(self, message):
        pass

class JSONFormatter(MessageFormatter):
    def format(self, message):
        return f'{{ "message": "{message.content}" }}'

class XMLFormatter(MessageFormatter):
    def format(self, message):
        return f"<message>{message.content}</message>"

class MessageDisplay:
    def __init__(self, formatter: MessageFormatter):
        self.formatter = formatter
    def display(self, message: Message):
        formatted_message = self.formatter.format(message)
        print(formatted_message)

message = Message("Hello, SOLID!")
json_formatter = JSONFormatter()
message_display = MessageDisplay(json_formatter)
message_display.display(message)
xml_formatter = XMLFormatter()
message_display = MessageDisplay(xml_formatter)
message_display.display(message)
```

### Step 4: Pray that your code works

#### Output

```
{'message': 'Hello, SOLID!'}
```

```
<message>Hello, SOLID!</message>
```

#### Explanation

1. **Single Responsibility Principle (SRP):** Separated the concerns into different classes: Message for managing message content, MessageFormatter for formatting messages, and MessageDisplay for displaying messages.
2. **Open/Closed Principle (OCP):** Introduced the MessageFormatter interface. New formatters can be added without modifying existing code, adhering to OCP.

3. **Liskov Substitution Principle (LSP):** Clients can use instances of derived classes (JSONFormatter, XMLFormatter) through the MessageFormatter interface without affecting the correctness of the program.
4. **Interface Segregation Principle (ISP):** Clients that only want to format messages can depend on MessageFormatter without being forced to depend on the display functionality.
5. **Dependency Inversion Principle (DIP):** MessageDisplayer depends on the MessageFormatter abstraction, not the concrete implementations. It inverts the traditional dependency from high-level modules to low-level modules.

This refactoring makes the application more maintainable, extensible, and robust by adhering to the SOLID principles.

## Example

Suppose we are developing a system for a bookstore that handles different types of book transactions such as selling, renting, and exchanging books.

### Initial

```
class BookstoreManager:
    def __init__(self, books):
        self.books = books
    def process_transaction(self, book_id, transaction_type):
        if transaction_type == "SELL":
            # process selling transaction
            print(f"Selling book with ID: {book_id}")
        elif transaction_type == "RENT":
            # process renting transaction
            print(f"Renting book with ID: {book_id}")
        elif transaction_type == "EXCHANGE":
            # process exchange transaction
            print(f"Exchanging book with ID: {book_id}")

# Usage
books = {"001": "Book 1", "002": "Book 2"}
bookstore_manager = BookstoreManager(books)
bookstore_manager.process_transaction("001", "SELL")
```

The main skill a software engineer should have is thinking and ability to solve real-world problems using programming. Try to reproduce steps described above and understand which concepts were used in order to refactor our bookstore system.

### Refactored

```
from abc import ABC, abstractmethod
class Book:
    def __init__(self, book_id, title):
        self.book_id = book_id
        self.title = title
class Transaction(ABC):
    @abstractmethod
```

```

    def execute(self, book: Book):
        pass
class SellTransaction(Transaction):
    def execute(self, book: Book):
        print(f"Selling book with ID: {book.book_id}")
class RentTransaction(Transaction):
    def execute(self, book: Book):
        print(f"Renting book with ID: {book.book_id}")
class ExchangeTransaction(Transaction):
    def execute(self, book: Book):
        print(f"Exchanging book with ID: {book.book_id}")
class BookstoreManager:
    def __init__(self, books):
        self.books = books
    def process_transaction(self, book_id, transaction: Transaction):
        if book_id in self.books:
            book = self.books[book_id]
            transaction.execute(book)
        else:
            print(f"Book with ID: {book_id} not found.")
# Usage
books = {"001": Book("001", "Book 1"), "002": Book("002", "Book 2")}
bookstore_manager = BookstoreManager(books)
sell_transaction = SellTransaction()
bookstore_manager.process_transaction("001", sell_transaction)

```

## 8. Quiz

### Question 1:

Which SOLID principle is violated in the Report class?

```

class Report:
    def generate_pdf_report(self, data):
        pass
    def generate_csv_report(self, data):
        pass
    def print_report(self, report):
        pass

```

- A) Single Responsibility Principle (SRP)
- B) Open/Closed Principle (OCP)
- C) Liskov Substitution Principle (LSP)
- D) Interface Segregation Principle (ISP)

### Question 2:

Which SOLID principle is violated in the vehicle class?

```

class Vehicle:
    def start_engine(self):

```

```

    pass
def stop_engine(self):
    pass
def fly(self):
    # Hint: Logic for flying (applicable only for certain vehicles)
    pass

```

- A) Single Responsibility Principle (SRP)
- B) Open/Closed Principle (OCP)
- C) Liskov Substitution Principle (LSP)
- D) Interface Segregation Principle (ISP)

### Question 3:

Which SOLID principle is violated in the NewsPublisher class?

```

class NewsPublisher:
    def publish_news(self, news):
        if self.platform == "Facebook":
            # Publish news to Facebook
            pass
        elif self.platform == "Twitter":
            # Publish news to Twitter
            pass

```

- A) Single Responsibility Principle (SRP)
- B) Open/Closed Principle (OCP)
- C) Liskov Substitution Principle (LSP)
- D) Dependency Inversion Principle (DIP)

### Question 4:

Which SOLID principle is violated in the code above?

```

class Rectangle:
    def set_dimensions(self, width, height):
        self.width = width
        self.height = height
class Square(Rectangle):
    def set_dimensions(self, side):
        self.width = side
        self.height = side

```

- A) Single Responsibility Principle (SRP)
- B) Open/Closed Principle (OCP)
- C) Liskov Substitution Principle (LSP)
- D) Dependency Inversion Principle (DIP)

### Question 5:

Which SOLID principle is most likely to be violated if the `EmailSender` class is used directly in high-level modules?

```
class EmailSender:
    def send_email(self, content, smtp_server):
        # Logic to send email using ``SMTP`` server
        pass
```

- A) Single Responsibility Principle (SRP)
- B) Open/Closed Principle (OCP)
- C) Interface Segregation Principle (ISP)
- D) Dependency Inversion Principle (DIP)

#### Question 6:

Which SOLID principle is violated in the `UserInterface` class?

```
class UserInterface:
    def display_text(self, text):
        pass
    def play_audio(self, audio):
        # Play audio (not needed for text-based interfaces)
        pass
```

- A) Single Responsibility Principle (SRP)
- B) Liskov Substitution Principle (LSP)
- C) Interface Segregation Principle (ISP)
- D) Dependency Inversion Principle (DIP)

#### Question 7:

Which SOLID principle is violated in the `MediaPlayer` class?

```
class MediaPlayer:
    def play_media(self, file):
        if file.type == "audio":
            # Play audio
            pass
        elif file.type == "video":
            # Play video
            pass
```

- A) Single Responsibility Principle (SRP)
- B) Open/Closed Principle (OCP)
- C) Liskov Substitution Principle (LSP)
- D) Dependency Inversion Principle (DIP)

## 9. Homework

### Task 1: SOLID Recipe Organizer:

**Objective:** Update an application below to manage a variety of recipes that allow users to add new recipes, store them, and display them in an organized manner. The application should be adaptable to different types of recipes and dietary requirements.

#### Requirements:

- Users should be able to create new recipes and specify if they are for a special diet.
- The application should be able to save recipes to a file or database.
- Users should be able to retrieve a list of all recipes or search for recipes by various criteria.
- Users should be able to update and delete recipes.

```
class RecipeOrganizer:
    def __init__(self):
        self.recipes = []
    def add_recipe(self, title, ingredients, instructions):
        self.recipes.append({
            'title': title,
            'ingredients': ingredients,
            'instructions': instructions
        })
    def display_recipes(self):
        for recipe in self.recipes:
            print(recipe['title'])
            print('Ingredients:', recipe['ingredients'])
            print('Instructions:', recipe['instructions'])
    def save_to_file(self):
        with open('recipes.txt', 'w') as file:
            for recipe in self.recipes:
                file.write(f"{recipe['title']}\n")
                file.write(f"{recipe['ingredients']}\n")
                file.write(f"{recipe['instructions']}\n\n")

# Usage
organizer = RecipeOrganizer()
organizer.add_recipe('Pasta', ['Pasta', 'Tomato'], 'Boil pasta, add tomato sauce')
organizer.display_recipes()
organizer.save_to_file()
```

Try to identify all violated SOLID principles and re-write this application.

### Task 2: Re-write your apps

Rewrite your apps you have created already to match SOLID. Don't forget to split logic into different modules as well.

# Lesson 18: Logging

"Logging is the silent guardian of software, capturing the whispers of code for analysis and insight. It's a real spy!"

## 1. What is logging?

Logging provides visibility into an application's behavior and state. It is essential for:

- **Debugging:** Understanding the flow of the application and diagnosing issues.
- **Monitoring:** Keeping track of the health and performance of the application.
- **Auditing:** Recording actions for compliance and security analysis.

Properly implemented logging can provide insights into what's happening in your application, which is invaluable for maintenance and troubleshooting.

## 2. Python's Built-in Logging Module

Python's built-in logging module allows you to log messages with different levels and direct them to several destinations.

### 2.1 Choosing the correct Log Level

There are different levels at which logging can be done:

- **DEBUG:** Detailed information, typically of interest only when diagnosing problems.
- **INFO:** Confirmation that things are working as expected.
- **WARNING:** An indication that something unexpected happened, or indicative of some problem in the near future.
- **ERROR:** Due to a more serious problem, the software has not been able to perform some function.
- **CRITICAL:** A serious error, indicating that the program itself may be unable to continue running.

You have to understand which level should be used based on representation above.

### 2.2 Syntax

We need to import logging module on the top of the file.

#### Example

```
import logging
# Configure basic logging
logging.basicConfig(level=logging.INFO)
# Log messages
logging.debug('This is a debug message')
logging.info('This is an info message')
logging.warning('This is a warning message')
logging.error('This is an error message')
```



```
logging.critical('This is a critical message')
```

## Output

```
INFO:root:This is an info message
WARNING:root:This is a warning message
ERROR:root:This is an error message
CRITICAL:root:This is a critical message
```

## 2.3 Practice!

Let's create a small application which will represent the importance of logging

### Example

```
import logging
logging.basicConfig(level=logging.INFO, format='%(asctime)s - %(levelname)s - %(message)s')
class TaskManager:
    def __init__(self):
        self.tasks = {}
        logging.info("Task Manager is initialized")
    def add_task(self, task_id, task_description):
        if task_id in self.tasks:
            logging.warning(f"Task with id {task_id} already exists. Overwriting.")
            self.tasks[task_id] = task_description
            logging.info(f"Task added: {task_id} - {task_description}")
        def complete_task(self, task_id):
            if task_id not in self.tasks:
                logging.error(f"Cannot complete task: {task_id}. Task does not exist.")
            return
            task_description = self.tasks.pop(task_id)
            logging.info(f"Task completed: {task_id} - {task_description}")
        def display_tasks(self):
            if not self.tasks:
                logging.warning("No tasks available to display.")
            return
            logging.debug("Current tasks:")
            for task_id, task_description in self.tasks.items():
                logging.debug(f"{task_id}: {task_description}")
    def main():
        task_manager = TaskManager()
        # Add some tasks
        task_manager.add_task(1, "Write blog post")
        task_manager.add_task(2, "Prepare dinner")
        task_manager.add_task(1, "Update blog post") # This will trigger a warning
        log and info log eventually
        task_manager.display_tasks()
        task_manager.complete_task(2)
        task_manager.complete_task(3) # This will trigger an error log
```

```
task_manager.display_tasks()
main()
```

## Output

```
2024-02-04 15:14:27,174 - INFO - Task Manager is initialized
2024-02-04 15:14:27,174 - INFO - Task added: 1 - Write blog post
2024-02-04 15:14:27,174 - INFO - Task added: 2 - Prepare dinner
2024-02-04 15:14:27,174 - WARNING - Task with id 1 already exists. Overwriting.
2024-02-04 15:14:27,174 - INFO - Task added: 1 - Update blog post
2024-02-04 15:14:27,174 - INFO - Task completed: 2 - Prepare dinner
2024-02-04 15:14:27,174 - ERROR - Cannot complete task: 3. Task does not exist.
```

Now, it is easy to track how our application is being used by the customer and we can trace each action made. This ensures, that it is possible to have visibility of the interaction and can prevent us from unexpected occurrences.

## 3. Configuring Logging: Handlers, Formatters, and Config Files

In order to create an effective and what is more important *useful* logging system we would want to have the following criterias under control:

- Log level.
- Message format.
- Well-structured configuration files.

### 3.1 Handlers

Handlers send the log messages to designated destinations.

*Each logger can have multiple handlers and each handler can process log messages differently.*

Common types of handlers include:

1. **StreamHandler**: Sends log messages to streams like `sys.stdout` or `sys.stderr`.
2. **FileHandler**: Writes log messages to a disk file. It's useful for keeping a persistent log.
3. **SMTPHandler**: Emails log messages to a specified email address. It's useful for critical error reporting.

And many more such as **HTTPHandler** and **SocketHandler**.

**Note**: Those two above are particularly useful once you build a microservice applications and want to have all logs within one reachable place.

Here's how you can set up multiple handlers with different log levels:

```
import logging
```

```

logger = logging.getLogger(__name__)
logger.setLevel(logging.DEBUG) # Set level for logger
console_handler = logging.StreamHandler()
console_handler.setLevel(logging.ERROR) # Only log ERROR and above to console
file_handler = logging.FileHandler('app.log')
file_handler.setLevel(logging.INFO) # Log INFO and above to file
logger.addHandler(console_handler)
logger.addHandler(file_handler)

```

In this setup, the logger captures all messages of level `DEBUG` and above, **but each handler filters the messages differently.**

## 3.2 Formatters

**Formatters specify the layout of log messages.** You can include information like `time`, `log level`, and the message.

### Example

```

formatter = logging.Formatter('%(asctime)s - %(levelname)s - %(message)s')
file_handler.setFormatter(formatter)

```

Commonly used format attributes include:

- `%(name)s`: Name of the logger.
- `%(levelno)s`: Numeric logging level.
- `%(levelname)s`: Text logging level.
- `%(pathname)s`: Full pathname of the source file where the logging call was issued.
- `%(filename)s`: Filename portion of pathname.
- `%(module)s`: Module name.
- `%(funcName)s`: Name of function containing the logging call.
- `%(lineno)d`: Source line number where the logging call was issued.
- `%(asctime)s`: Human-readable time when the `LogRecord` was created.
- `%(message)s`: The logged message.

## 3.3 Config Files

In order to have a convenient way of storing the configurations for everything described above we have two and only two the best options

### Option #1

**Step 1:** Create `logging_config.ini` file and define expected configuration which is suitable for your application needs.

```
[loggers]
```

```

keys=root,sampleLogger
[handlers]
keys=consoleHandler
[formatters]
keys=sampleFormatter
[logger_root]
level=DEBUG
handlers=consoleHandler
[logger_sampleLogger]
level=DEBUG
handlers=consoleHandler
qualname=sampleLogger
propagate=0
[handler_consoleHandler]
class=StreamHandler
level=DEBUG
formatter=sampleFormatter
args=(sys.stdout,)
[formatter_sampleFormatter]
format=%(asctime)s - %(name)s - %(levelname)s - %(message)s
datefmt=%Y-%m-%d %H:%M:%S

```

**Step 2:** Load the configuration file using `logging.config.fileConfig`

```

import logging.config
logging.config.fileConfig('logging.conf')
logger = logging.getLogger('sampleLogger')
logger.debug('This is a debug message')

```

Now you can change the logging behavior without modifying the application code which is great in terms of scalability.

Let's create a mini-application which will use all functionality of our logging setup. And you will see how the quality and readability of our code was improved.

## Example

```

import logging.config
# Load the logging configuration
logging.config.fileConfig('logging_config.ini')
logger = logging.getLogger('bookstoreLogger')
class Bookstore:
    def __init__(self):
        self.books = {}
        logger.info("Bookstore is initialized")
    def add_book(self, isbn, book_details):
        if isbn in self.books:
            logger.warning(f"Book with ISBN {isbn} already exists. Updating the book.")
        else:
            logger.info(f"Book added: ISBN {isbn}, Details: {book_details}")

```

```

        self.books[isbn] = book_details
    def update_book(self, isbn, book_details):
        if isbn in self.books:
            logger.info(f"Book {isbn} updated. Old: {self.books[isbn]}, New:
{book_details}")
            self.books[isbn] = book_details
        else:
            logger.error(f"Cannot update book: ISBN {isbn}. Book does not
exist.")
    def remove_book(self, isbn):
        if isbn in self.books:
            removed_book = self.books.pop(isbn)
            logger.info(f"Book removed: ISBN {isbn} - {removed_book}")
        else:
            logger.error(f"Cannot remove book: ISBN {isbn}. Book does not
exist.")
    def list_books(self):
        if not self.books:
            logger.warning("No books available in the bookstore.")
        else:
            for isbn, details in self.books.items():
                logger.info(f"ISBN {isbn}: {details}")
# Main function to test the Bookstore class
def main():
    bookstore = Bookstore()
    bookstore.add_book("978-0132350884", {"title": "Clean Code", "author":
"Robert C. Martin"})
    bookstore.add_book("978-0201633610", {"title": "Design Patterns", "author":
"Erich Gamma"})
    bookstore.add_book("978-0132350884", {"title": "Clean Code", "author":
"Robert C. Martin"})
    bookstore.update_book("978-0321125217", {"title": "Domain-Driven Design",
"author": "Eric Evans"})
    bookstore.remove_book("978-0201633610")
    bookstore.list_books()
if __name__ == "__main__":
    main()

```

## Output

```

2024-02-04 16:54:17 - bookstoreLogger - INFO - Bookstore is initialized
2024-02-04 16:54:17 - bookstoreLogger - INFO - Book added: ISBN 978-0132350884,
Details: {'title': 'Clean Code', 'author': 'Robert C. Martin'}
2024-02-04 16:54:17 - bookstoreLogger - INFO - Book added: ISBN 978-0201633610,
Details: {'title': 'Design Patterns', 'author': 'Erich Gamma'}
2024-02-04 16:54:17 - bookstoreLogger - WARNING - Book with ISBN 978-0132350884
already exists. Updating the book.
2024-02-04 16:54:17 - bookstoreLogger - ERROR - Cannot update book: ISBN 978-
0321125217. Book does not exist.
2024-02-04 16:54:17 - bookstoreLogger - INFO - Book removed: ISBN 978-
0201633610 - {'title': 'Design Patterns', 'author': 'Erich Gamma'}
2024-02-04 16:54:17 - bookstoreLogger - INFO - ISBN 978-0132350884: {'title':
'Clean Code', 'author': 'Robert C. Martin'}

```

When we have different modules within our application it is crucial to separate the logging for each module and define its custom logger object to improve

## Option #2

```
import logging
import logging.handlers
import logging.config
import os

# Logging configuration dictionary
LOGGING_CONFIG = {
    'version': 1,
    'disable_existing_loggers': False,
    'formatters': {
        'detailed': {
            'class': 'logging.Formatter',
            'format': '%(asctime)s [%(levelname)s] %(name)s: %(message)s'
        },
        'email': {
            'class': 'logging.Formatter',
            'format': 'Timestamp: %(asctime)s\nModule: %(module)s\nLine: %(lineno)d\nMessage: %(message)s'
        },
    },
    'handlers': {
        'console': {
            'class': 'logging.StreamHandler',
            'level': 'INFO',
            'formatter': 'detailed',
            'stream': 'ext://sys.stdout'
        },
        'file': {
            'class': 'logging.handlers.RotatingFileHandler',
            'level': 'DEBUG',
            'formatter': 'detailed',
            'filename': 'app.log',
            'maxBytes': 1024000,
            'backupCount': 3
        },
        'email': {
            'class': 'logging.handlers.SMTPHandler',
            'level': 'ERROR',
            'formatter': 'email',
            'mailhost': 'localhost',
            'fromaddr': 'yourapp@yourdomain.com',
            'toaddrs': ['admin@yourdomain.com'],
            'subject': 'Critical Error in YourApp!'
        },
    },
    'loggers': {
        '': {
            'handlers': ['console', 'file', 'email'],
            'level': 'DEBUG',
        },
    },
}
```

```
# Applying the logging configuration
logging.config.dictConfig(LOGGING_CONFIG)
# Getting the logger
logger = logging.getLogger(__name__)
# Test messages
logger.debug("This is a debug message")
logger.info("This is an info message")
logger.warning("This is a warning message")
logger.error("This is an error message")
logger.critical("This is a critical message")
```

Be mindful of the log level and the amount of logging in performance-critical parts of the application. It's really important to tackle this in advance.

Keep the logging configuration separate from the application logic. Using configuration files or a dedicated configuration module can achieve this.

Use different logging configurations for different environments (development, testing, production). This can be managed by having separate configuration files for each environment or by using environment variables (.env).

## 4. Logging Best Practices

Effective logging is not just about adding log statements to your code. It involves thoughtful consideration of *what, where, and how to log*.

1. **Be clear and descriptive:** Log messages should provide enough context to be understood on their own.
2. **Use appropriate log levels:** This helps in filtering and analyzing logs.
3. **Avoid logging sensitive information:** Such as passwords or personal user data.
4. **Manage log file size:** Use mechanisms like log rotation to avoid consuming too much disk space.

And that's it! Happy Logging!

## 5. Homework

Add logging to your existing projects, config loggers in the most suitable way for you or use configs provided in this lesson.

# Lesson 19: Testing

"In the world of Python, testing is not just a phase. It's a commitment to excellence in craftsmanship."

## 1. Why do we need testing?

Testing your code is essential for verifying its correctness, ensuring reliability, and maintaining code quality.

The general idea of testing, is ensuring that your application works as expected. Due to testing you will be able to detect bugs early during the development process and fix them immediately.

As well it can represent sort of documentation that helps developers to understand how the application should work and show the behavior of an application.

## 2. Types of Testing

There are several types of testing which are used widely in IT society.

Type of Testing	Description
Unit Testing	Testing individual units or components of a program in isolation to verify that each part functions correctly.
Integration Testing	Testing the integration or interfaces between components to ensure they work together as expected.
Functional Testing	Testing the application against its functional requirements to ensure it behaves as expected from an end-user perspective.

The best approach would be to cover all of these scenarios using testing tools, but during this lesson I want to focus more on unit testing and tell about available options in Python.

## 3. Introduction to `unittest`

The most common way to test Python application is to use built in `unittest` library.

Suppose we have the following logic encapsulated within our app:

### Example

```
def add(a, b):  
    return a + b  
def subtract(a, b):  
    return a - b  
def divide(a, b):  
    return a / b
```



As a developer, several questions should come to mind regarding potential issues:

- What if `add` or `subtract` are passed non-numeric types, like a string and a number?
- What happens when attempting to `divide` by zero?
- Will the application crash, or will it handle these situations gracefully?

Instead of making assumptions, we can write tests to guarantee that our application behaves as intended under various circumstances.

### 3.1 Writing Your First Unit Test

Unit tests are designed to test individual components, or "units", of your application in isolation.

This means you should be testing the smallest part of an application, like a function or a method, to ensure it does exactly what it's supposed to do.

#### Example

```
import unittest
# Our app
def add(a, b):
    return a + b
def subtract(a, b):
    return a - b
def divide(a, b):
    if b == 0:
        raise ValueError("Cannot divide by zero")
    return a / b
# Test cases
class TestArithmeticFunctions(unittest.TestCase):

    def test_add(self):
        """
        - Testing subtraction with correct types and assure it works as
        expected.
        - Testing subtraction with incorrect types.
        """
        self.assertEqual(add(1, 2), 3)
        self.assertEqual(add(-1, 1), 0)
        self.assertRaises(TypeError, add, '1', 2)
    def test_subtract(self):
        """
        - Testing subtraction with correct types and assure it works as
        expected.
        - Testing subtraction with incorrect types.
        """
        self.assertEqual(subtract(10, 5), 5)
        self.assertEqual(subtract(-1, -1), 0)
        self.assertRaises(TypeError, subtract, '10', 5)
    def test_divide(self):
        """
        - Testing division with correct types and assure it works as expected.
        - Testing division by zero.
```

```

- Testing division with incorrect types.
"""
self.assertEqual(divide(10, 2), 5)
self.assertRaises(ValueError, divide, 10, 0)
self.assertRaises(TypeError, divide, '10', 2)

# This ensures that the tests will be run if the script is executed
if __name__ == '__main__':
    unittest.main()

```

The output shows that tests were successful. Try to modify written the tests and see what happens in case the logic we are trying to test, they should fail explaining what happens wrong, so that the developer can analyse and find out what happens wrong in application.

## Output

```

...
-----
Ran 3 tests in 0.000s
OK

```

Testing the OOP applications isn't really different from we have seen already.

Let's create a small BankAccount system and try to write tests for it:

## Example

```

class BankAccount:
    def __init__(self):
        self.balance = 0
    def deposit(self, amount):
        if amount <= 0:
            raise ValueError("Deposit amount must be positive")
        self.balance += amount
    def withdraw(self, amount):
        if amount > self.balance:
            raise ValueError("Insufficient funds")
        self.balance -= amount
    def get_balance(self):
        return self.balance

```

## Example

Here we would need to define the setUp method which creates an instance of BankAccount and it will be easily accessible within TestBankAccount method

```

import unittest
class TestBankAccount(unittest.TestCase):

```

```

def setUp(self):
    """
    The setUp method is run before each test. It's a good place to set up
    a clean environment for each test.
    """
    self.account = BankAccount()
def test_deposit(self):
    """
    Test that depositing money into the account works correctly.
    """
    self.account.deposit(100)
    self.assertEqual(self.account.get_balance(), 100)
def test_withdraw(self):
    """
    Test that withdrawing money from the account works correctly.
    """
    self.account.deposit(200)
    self.account.withdraw(50)
    self.assertEqual(self.account.get_balance(), 150)
def test_withdraw_insufficient_funds(self):
    """
    Test that a ``ValueError`` is raised when trying to withdraw more money
    than the balance of the account.
    """
    self.account.deposit(50)
    with self.assertRaises(ValueError):
        self.account.withdraw(100)
def test_deposit_negative_amount(self):
    """
    Test that a ``ValueError`` is raised when trying to deposit a negative
    amount.
    """
    with self.assertRaises(ValueError):
        self.account.deposit(-20)
def test_initial_balance(self):
    """
    Test that the initial balance of the account is zero.
    """
    self.assertEqual(self.account.get_balance(), 0)
# This ensures that the tests will be run if the script is executed
if __name__ == '__main__':
    unittest.main()

```

## Output

```

.....
-----
Ran 5 tests in 0.000s

```

## 3.2 Structuring Your Tests

As everything in programming MUST have a correct structure, as we have seen it already

Same applies to testing with `unittest`, each test case class **MUST** have the following:

- **Setup:** Prepare the necessary environment or state before the actual tests run. This is often done in a method named `setUp()`.
- **Test Cases:** Individual functions that start with the word `test_` to inform the test runner what to execute.
- **Assertions:** Statements that check if the output of your code matches the expected result.
- **Teardown** (Optional): Clean-up steps that need to be taken after the test cases run, often implemented in a method named `tearDown()`.

Generally, testing should not be crossed with code which will be used in production, a great approach would be to create a `tests/` directory and include all tests and files there.

### Example

```
my_project/

|

├─ my_project/

|   └─ __init__.py

|   └─ bank_account.py

|

└─ tests/

    └─ __init__.py

    └─ test_bank_account.py
```

## 3.3 Running Tests with a Command

To run the tests, navigate to the directory containing your test script and execute the following command:

```
python test_bank_account.py
```

Alternatively, if you want to run all tests across different test files, use:

```
python -m unittest discover
```

## 4. Introduction to `pytest`

There is no really much difference between `pytest` and `unittest` tools for testing in terms of serving their purposes. They are both configurable, they both test your application and work pretty much similar.

But, `pytest` simplifies the writing of small tests yet scales to support complex functional testing. This is why `pytest` is my favored tool and most of production code is tested within this library.

### 4.1 Install `pytest`

To begin using `pytest`, you first need to install it via `pip`:

```
pip install pytest
```

Once installed, writing a test is as straightforward as defining a function prefixed with `test_`, and then using plain `assert` statements:

#### Example

```
# test_example.py
def add(a, b):
    return a + b
def test_add():
    assert add(2, 3) == 5
    assert add('space', 'ship') == 'spaceship'
```

#### Running tests

```
pytest
```

`pytest` will automatically discover and run any files named `test_*.py` or `*_test.py` in the current directory and its subdirectories.

You can create the same tests for your application, and using `pytest` in testing your app.

### 4.2 Running Tests with `pytest`:

There are some useful commands which I use in production day to day, they can narrow down the tests or provide more information:

Command	Description
---------	-------------

Command	Description
<code>pytest test_example.py</code>	Executes all the tests defined in <code>test_example.py</code> .
<code>pytest tests/</code>	Runs all the tests found in the <code>tests</code> directory.
<code>pytest -v</code>	Provides a verbose output, detailing all the tests run along with their individual outcomes.
<code>pytest test_example.py::test_add</code>	Executes only the <code>test_add</code> function within the <code>test_example.py</code> file.
<code>pytest -k "expression"</code>	Runs tests that match the given substring expression, a powerful way to run specific tests within a larger suite.
<code>pytest --ignore=tests/</code>	Ignores the specified directory or file during the test run.
<code>pytest --maxfail=num</code>	Exits the test session after <code>num</code> failures, useful for quickly identifying and addressing errors.
<code>pytest --tb=style</code>	Modifies the traceback output format for easier debugging. Valid styles include <code>long</code> , <code>short</code> , <code>line</code> , <code>native</code> , and <code>no</code> .
<code>pytest -x</code>	Stops the test run on the first failure encountered, which can be useful during development.
<code>pytest --lf</code>	Reruns only the tests that failed at the last run, or skips tests that passed.

Don't hesitate to use anything from this command during development, they can save you a decent amount of time, helping with debugging of your application.

## 5. Test-Driven Development (TDD)

Test-Driven Development (TDD) is a modern software development practice where tests are written before the code that will make the tests pass. It follows a simple iterative cycle known as "Red-Green-Refactor":

1. **Red:** Write a test that defines a function or improvements of a function, which should fail because the function isn't implemented yet.
2. **Green:** Implement the function in the simplest way possible to make the test pass.
3. **Refactor:** Clean up the code, while ensuring that tests still pass.

TDD encourages developers to think through their design before writing the code.

### 5.1 Benefits of TDD

- **Documentation:** The tests serve as live documentation for the application.
- **Design:** Helps in building a better design as it requires writing testable code.

- **Confidence:** Each change is made with confidence that the existing features are not broken.

The disadvantage is that sometimes it is really hard to follow this approach, either it is time-consuming, the project is enormously big and it is hard for developer to put it alltogether.

I won't lie, if I have a deadline for the task, sometimes I might neglect using TDD, though no one should do this :)

Let's create an application using TDD approach step by step to see how it should look like:

### Step 1: Writing a Failing Test (Red Phase)

Suppose we're building a simple calculator application. Our first feature is an addition function. We start by writing a test for this functionality before the function itself exists.

Create a test file `test_calculator.py`:

```
# test_calculator.py
def test_addition():
    assert add(2, 3) == 5
```

Running this test (`pytest test_calculator.py`) will result in a failure because the `add` function does not exist yet. This is the Red phase of TDD.

### Step 2: Making the Test Pass (Green Phase)

Now, we write the minimal amount of code needed to pass the test. Create a file `calculator.py` and implement the `add` function:

```
# calculator.py
def add(a, b):
    return a + b
```

And modify `test_calculator.py` to import the `add` function:

```
# test_calculator.py
from calculator import add
def test_addition():
    assert add(2, 3) == 5
```

Running the tests again with `pytest`, we see that the test now passes. This is the Green phase of TDD.

### Step 3: Refactoring (Refactor Phase)

With the test passing, we can now refactor our code with confidence. This might involve the following:

- Renaming functions for clarity.
- Optimizing the algorithm.
- Restructuring the code for better readability.

#### Step 4. Run tests again

After refactoring, run the tests again to ensure nothing has broken. This continuous cycle enhances the code quality over time.

#### Output

```
.  
-----  
Ran 1 tests in 0.000s  
OK
```

TDD requires strong discipline and may initially slow down development, especially for teams new to this approach. However, the long-term benefits are much more significant, consider adopting this practice into your applications.

## 5. Mocking and Patching

### 5.1 Mocking

Mocking objects simulate the behavior of real objects within your system.

Mock objects can be programmed with predefined responses, making them highly flexible for testing a wide range of scenarios.

- Test components in isolation from the rest of the system.
- Simulate various states of external systems or resources that are difficult or time-consuming to replicate in a test environment.
- Avoid side effects that can interfere with test outcomes.
- Control the test environment by specifying expected inputs and outputs.

### 5.2 Patching

Patching (often used in conjunction with mocking) involves temporarily replacing the actual implementation of a class, method, or function with a mock during test execution.

- Functions and methods, to control their outputs or side effects.
- System-level operations, such as file I/O, to prevent tests from altering the system's state.
- Libraries and frameworks, to test your code's interaction with them without requiring the actual implementation to be invoked.



## 5.3 Practice!

pytest can integrate the `unittest.mock` module from the Python standard library, enabling the use of mocks and patches in your tests.

With this integration you can simulate complex behaviors and the assertion of interactions with mock objects.

### Example

Imagine we have a simple application that sends email notifications to users about important events. For simplicity, let's focus on the function that triggers sending the email.

```
# notifications.py
def send_email_notification(email_address, message):
    if not email_address or not message:
        raise ValueError("Email address and message are required")
    print(f"Sending email to {email_address}: {message}")
    return True
```

### Testing

```
# test_notifications.py
import pytest
from unittest.mock import patch
from notifications import send_email_notification
@patch('notifications.print') # Mocking the print function to simulate email sending
def test_send_email_notification(mock_print):
    email_address = "user@example.com"
    message = "Your application has been approved."
    result = send_email_notification(email_address, message)
    # Assert the mock (print function here) was called with the expected arguments
    mock_print.assert_called_with(f"Sending email to {email_address}: {message}")
    assert result == True
def test_send_email_notification_missing_arguments():
    # Testing missing arguments to ensure our error handling works
    with pytest.raises(ValueError):
        send_email_notification("", "")
```

### Explanation

- **Mocking:** In this example, we mock the `print` function to simulate the action of sending an email. The `@patch` decorator from `unittest.mock` is used to replace `print` with a mock object only for the duration of the test.
- **Assertion:** We assert that the mock object was called with the expected arguments, simulating the check that the email notification was triggered with the correct content.

Instead of sending emails every time we run the test, we "simulated" the logic of email sending and have tested that everything works as expected.

## 6. Advanced Techniques

### 6.1 Parameterized Testing

Parameterized testing allows you to run the same test function with different inputs, reducing code duplication and making it easier to cover a wide range of scenarios.

`pytest.mark.parametrize`

pytest offers a simple way to parameterize tests using the `@pytest.mark.parametrize` decorator.

```
import pytest
@pytest.mark.parametrize("test_input,expected", [
    (5, 25),
    (9, 81),
    (2, 4)
])
def test_square(test_input, expected):
    assert test_input ** 2 == expected
```

This test will run three times, each with the different `test_input` and `expected` values provided.

(So there will be 3 tests with different values inside one, instead of creating tons of duplicate code)

### 6.2 Fixture Management

Fixtures in pytest are functions run by pytest before (and sometimes after) the actual test functions to which they're applied. Fixtures are a powerful feature for setting up and tearing down test *environments or contexts*.

#### Example

```
import pytest
@pytest.fixture
def sample_data():
    return [1, 2, 3, 4, 5]
# After creating a fixture we can use an object to test different conditions
def test_sum(sample_data):
    assert sum(sample_data) == 15
def test_length(sample_data):
```

```
assert len(sample_data) == 5
```

The `sample_data` fixture is automatically injected into the `test_sum` and `test_length` function by `pytest`, demonstrating fixture management for reusable test data setup.

## 6.3 Error Handling in Tests

Testing how your application handles errors is as crucial as testing its success paths. `pytest` simplifies the process of asserting exceptions.

### Testing for Expected Errors

```
import pytest
def raise_custom_error():
    raise ValueError("An error occurred")
def test_raise_custom_error():
    with pytest.raises(ValueError) as e:
        raise_custom_error()
    assert str(e.value) == "An error occurred"
```

This test checks that `raise_custom_error` raises a `ValueError` with the expected message.

All tools can be easily integrated in order to enhance the quality of your tests and after having such knowledge you as developer will be able to write well-structured easy to use tests.

## 7. Coverage Analysis

Code coverage is a measure used to describe the degree to which the source code of a program is executed when a particular test suite runs.

A program with high code coverage has had more of its source code tested, which can lead to fewer bugs.

### 7.1 Install `coverage.py`

`coverage.py` is a tool for measuring code coverage of Python programs. It monitors your program, noting which parts of the code have been executed.

```
pip install coverage
```

### 7.2 Use `coverage.py` with `pytest`

1. Run your tests under coverage run:

```
coverage run -m pytest
```

2. Then, generate a report:

```
coverage report
```

Or for an HTML version:

```
coverage html
```

This report will show which lines of code were not executed by your tests, helping identify areas needing additional testing, which can be added.

## 8. Applying Testing

Let's create a new application, but now, we will use everything we have learnt during this section

### Step 1: Defining Our Application's Requirements

For our Task Manager application, we have the following requirements:

- Ability to add tasks with a unique identifier.
- Ability to mark tasks as complete.
- Send a notification when a task is marked as complete.

### Step 2: Writing Tests First (TDD Approach)

Following the TDD approach, we start by writing tests for our yet-to-be-implemented features. We focus on testing our application's core functionality and how it handles edge cases.

**test\_task\_manager.py**

```
import pytest
from task_manager import TaskManager
@pytest.fixture
def task_manager():
    return TaskManager()           # Instance of our class
def test_add_task(task_manager):
    task_id = task_manager.add_task("Learn pytest")
    assert task_manager.get_task(task_id) == "Learn pytest"
def test_mark_task_as_complete(task_manager):
    task_id = task_manager.add_task("Learn mocking")
    task_manager.mark_task_as_complete(task_id)
    assert task_id not in task_manager.tasks
    assert task_id in task_manager.completed_tasks
def test_mark_nonexistent_task_as_complete(task_manager):
```

```
with pytest.raises(ValueError):
    task_manager.mark_task_as_complete(999)
```

### Step 3: Implementing the Application Logic

After defining our tests, we proceed to implement the application logic to make these tests pass.

`task_manager.py`

```
class TaskManager:
    def __init__(self):
        self.tasks = {}
        self.completed_tasks = {}
    def add_task(self, description):
        task_id = len(self.tasks) + 1
        self.tasks[task_id] = description
        return task_id
    def mark_task_as_complete(self, task_id):
        if task_id not in self.tasks:
            raise ValueError("Task ID does not exist")
        self.completed_tasks[task_id] = self.tasks.pop(task_id)
    def get_task(self, task_id):
        return self.tasks.get(task_id, "Task does not exist")
```

### Step 4: Running the Tests

Use `pytest` to run your tests and ensure they all pass:

```
pytest test_task_manager.py
```

### Step 5: Parameterized Testing

Use parameterized tests to run the same test logic with different inputs effortlessly.

```
@pytest.mark.parametrize("description", ["Task 1", "Task 2", "Task 3"])
def test_add_multiple_tasks(task_manager, description):
    task_id = task_manager.add_task(description)
    assert task_manager.get_task(task_id) == description
```

### Step 6: Coverage Analysis

Finally, assess your test suite's effectiveness using coverage analysis tools to ensure every line of your application logic is tested.

```
pytest --cov=task_manager test_task_manager.py
```

Congratulations, we have learnt how to write Unit Tests!

## 9. Homework

Cover your existing projects with unit tests, aim to get 99% of coverage. Add mocking and patching to the application we have written in Section 8.

# Lesson 20: Iterators and Generators

"Just lazy chaps"

## 1. Iterators

In Python, iterators are fundamental constructs that allow for efficient looping through collections of items, (iterables), such as lists or tuples. They implement two special methods, `__iter__()` and `__next__()`. The `__iter__()` method returns the iterator object itself and is automatically called at the start of loops.

### 1.1 Why do we need them?

**1.Memory Efficiency:** By allowing for item-by-item processing, iterators enable handling large datasets and streams efficiently, without loading everything into memory.

**2.Lazy Evaluation:** This feature enhances performance by delaying the computation of values until the moment they are actually needed. It allows for the handling of potentially infinite data streams.

### 1.2 Syntax

To manually iterate over an iterable object, you can use the `iter()` function to convert it into an iterator and then repeatedly call `next()` to get each item.

#### Example

```
my_list = [1, 2, 3]
my_iter = iter(my_list)
print(next(my_iter))
print(next(my_iter))
print(next(my_iter))
print(type(my_iter))
try:
    print(next(my_iter)) # This will raise StopIteration
except StopIteration:
    print("Reached the end of the iterator")
```

#### Output

```
1
2
3
<class 'list_iterator'>
Reached the end of the iterator
```

As well as we can use iterators in `for` loop. It internally converts the iterable into an iterator, automatically calls `__iter__()` to initiate the iteration, and handles the `StopIteration` exception by terminating the loop when the end of the iterator is reached.

### Example

```
my_list = [4, 7, 0, 3]
# Iterating over the list
for item in my_list:
    print(item)
```

### Output

```
4
7
0
3
```

It's a built in way of working with iterators, but in reality sometimes we want to have more control over them, so that we can define custom iterators.

## 2. Building Your Own Iterators

Creating a custom iterator involves defining a class that implements the `__iter__()` and `__next__()` methods. Let's create a simple class that iterates from 1 up to a given number.

### Example

```
class CountUpTo:
    def __init__(self, max):
        self.max = max
        self.num = 1
    def __iter__(self):
        return self
    def __next__(self):
        if self.num <= self.max:
            result = self.num
            self.num += 1
            return result
        else:
            raise StopIteration

counter = CountUpTo(3)
for num in counter:          # Python calls method ``__next__()`` during each
    print(num)               iteration
```



## Explanation

- The `__iter__()` method must return the iterator object itself. This is used by Python to create an iterator from an iterable object.
- The `__next__()` method must return the next item in the sequence. On reaching the end, and to avoid an infinite loop, it should raise a `StopIteration` exception.

## Output

```
1
2
3
```

A few more examples which can be used for real world applications such as iterator for processing a large file or the iterator representing the tray.

## Example

```
class LargeFileLineIterator:
    def __init__(self, filepath):
        self.filepath = filepath
    def __iter__(self):
        self.file = open(self.filepath, 'r')
        return self
    def __next__(self):
        line = self.file.readline()
        if line:
            return line.strip() # Remove the newline character from the end
        else:
            self.file.close() # Close the file when done
            raise StopIteration

# for will call iter under the hood
filepath = 'path/to/large/file.txt'
for line in LargeFileLineIterator(filepath):
    print(line)
```

Use Python Visualiser to show how exactly iterators are being called and what happens under the hood.

## Example

```
class ListContainer(object):
    def __init__(self, fruits):
        self.fruits = fruits
    def __iter__(self):
        return iter(self.fruits)

# Imagine we have a really big amount of fruits here, in this case we might
# consider using a custom storage instead of default `list` in Python
fruits = ListContainer(["orange", "mango", "banana"])
for fruit in fruits:
    print(fruit)
```

## Output

```
orange
mango
banana
```

## 3. Performance overview

You could ask, why do we need iterator here if we can use the default `with open(filepath)` context manager and read all lines as we have learnt before.

Let's take a closer look on the performance. In the example below we will compare processing a really big file with and without usage of iterators and compare the output.

### Example

```
import time
start_time = time.time()
with open('Intermediate/Assets/m.txt', 'r') as f:
    lines = f.readlines()
    line_count = len(lines)
end_time = time.time()
non_iterator_time = end_time - start_time
print(f"Line count: {line_count}")
print(f"Processing time without iterator: {non_iterator_time} seconds")
start_time = time.time()
line_count = 0
with open('Intermediate/Assets/m.txt', 'r') as f:
    for line in f: # This uses an iterator internally
        line_count += 1
end_time = time.time()
iterator_time = end_time - start_time
print(f"Line count: {line_count}")
print(f"Processing time with iterator: {iterator_time} seconds")
print(f"Time taken without using iterator: {non_iterator_time:.4f} seconds.")
print(f"Time taken using iterator: {iterator_time:.4f} seconds.")
```

**Note::** Time processing may different because of hardware used for calculations

## Output

```
# Time taken without using iterator: 0.0340 seconds.

# Time taken using iterator: 0.0199 seconds.
```

As you can see, the difference is not very significant, but for the bigger files in production environment it can play a key role for performance.

## 4. Generators

Generators are a simple and powerful tool for creating iterators in Python. They allow you to declare a function that behaves like an iterator, i.e., it can be used in a for loop.

### 4.1 Why?

1. **Highly memory-efficient** - they yield one item at a time, only holding one item in memory.
2. **Reduce the complexity of creating iterators**: There's no need to implement the `__iter__()` and `__next__()` methods; the generator function automatically creates these methods in the background.

*But **the best** part of using them is that:*

*Generators can be composed together, allowing for the construction of pipelines that process data in a memory-efficient manner and can be used to filter, transform, or aggregate data efficiently.*

It is an ideal choice for processing streams of data, such as log files, sensor data, or large datasets that cannot fit into memory.

I opened this approach (look at section 4.3.3) recently and didn't know about for a long time, but for now it is being used on the daily basis, let's finally take a look and skip this boring theory.

### 4.2 Syntax

A generator is defined much like a normal function, but it uses the `yield` statement to return data. Each time the generator's function is called, it resumes execution right after the `yield` statement where it left off.

This behavior allows generators to produce a sequence of values over time, pausing after each `yield` and continuing from there on the next call.

#### Example

```
def count_up_to(max):
    count = 1
    while count <= max:
        yield count      # instead of return!
        count += 1
# Using the generator
for number in count_up_to(3):
    print(number)
```

#### Output

```
1
2
3
```

## 4.3 Advanced Generator Examples

Here are a few more examples to illustrate their power in real-world scenarios.

### 4.3.1 Generating Infinite Sequences

One of the fascinating uses of generators is creating infinite sequences. Unlike lists or tuples, generators can produce values indefinitely.

#### Example: Infinite Fibonacci Sequence

```
def infinite_fibonacci():
    a, b = 0, 1
    while True:
        yield a
        a, b = b, a + b
# Using the generator
fib = infinite_fibonacci()
for _ in range(7):
    print(next(fib))
```

#### Output

```
0
1
1
2
3
5
8
```

### 4.3.2 Generator Expressions

Python supports generator expressions, which offer a concise syntax for creating generators. They are similar to list comprehensions but use parentheses instead of square brackets. It's the same as a function with `yield` we have seen in a previous example.

#### Example

```
squares = (x**2 for x in range(10))
for square in squares:
    print(square)
print()
```

#### Output

0  
1  
4  
9  
16  
25  
36  
49  
64  
81

**IMPORTANT:** Do not confuse them with list/set/dict comprehensions!

### 4.3.3 Chaining Generators

Generators can be chained together to create powerful data processing pipelines.

#### Example

Imagine you have a log file where each line contains a timestamp and a message. You want to filter specific messages and then format them for display.

```
def read_logs(file_path):
    with open(file_path, 'r') as file:
        for line in file:
            yield line.strip()
def filter_errors(log_lines):
    for line in log_lines:
        if "ERROR" in line:
            yield line
def format_errors(error_lines):
    for line in error_lines:
        yield f"Error found: {line}"
# Chaining generators
log_path = "Base/assets/application.log"
formatted_errors = format_errors(filter_errors(read_logs(log_path)))
for error in formatted_errors:
    print(error)
```

#### Output

```
Error found: 2024-05-04 08:05:00 ERROR: Database connection failed

Error found: 2024-05-04 08:20:00 ERROR: Server timeout
```

With generators, you can solve a wide range of programming problems such as infinite sequences, large datasets, or building complex data processing pipelines.

It's incredibly powerful mechanism, don't hesitate to use them in your apps!

## 4.4 Memory Usage

Generators are designed to yield one item at a time, only holding that item in memory, which contrasts sharply with lists that store all their elements in memory. Again, this difference becomes especially significant when working with large data volumes.

### Example

Consider calculating the sum of a large range of numbers. Using a list comprehension would require storing all numbers in memory, whereas a generator expression does not.

```
import sys
# Using a list comprehension
large_list = [x for x in range(1000000)]
print("List memory:", sys.getsizeof(large_list), "bytes")
# Using a generator expression
large_gen = (x for x in range(1000000))
print("Generator memory:", sys.getsizeof(large_gen), "bytes")
```

### Output

```
List memory: 8448728 bytes
Generator memory: 104 bytes
```

## 5. Generators VS Lists

**Generators** are ideal for:

- Large datasets that do not fit into memory.
- Stream processing or pipelining tasks where data can be processed sequentially.
- Situations where only a part of the data is needed at any one time.

**Lists** are better suited for:

- Situations requiring random access to elements.
- When the size of the dataset is relatively small, and the overhead of storing it in memory is not a concern.
- Tasks involving list-specific operations like slicing or list comprehensions that benefit from having all data available at once.

Decide what exactly should be used by the needs of your application. My genuine advice would be not to overuse generators as this can lead to some bugs which are hard to track.

## 6. Best practices

**Using Context Managers:** Whenever possible, use context managers (`with` statement) within your generator to ensure that resources are automatically cleaned up when the generator is exhausted or if an exception occurs.

```
def read_file_lines(file_path):  
    with open(file_path, 'r') as file:  
        for line in file:  
            yield line
```

**Try-Finally Blocks:** For more complex scenarios where context managers cannot be used directly within the generator, ensure cleanup code is run through a `try-finally` block.

```
def custom_generator():  
    resource = allocate_resource()  
    try:  
        yield from process_resource(resource)  
    finally:  
        free_resource(resource)
```

**Explicit Closure:** In cases where a generator may not be entirely consumed, ensure that any external resources are explicitly released. This can be done by calling the generator's `close()` method, which triggers any `finally` blocks associated with the generator.

```
gen = custom_generator()  
try:  
    next(gen)  
    # If not consuming the entire generator,  
    # ensure resources are released  
finally:  
    gen.close()
```

Additionally I would recommend to try out `itertools` collections to dive deeper into generators as a part of your further learning.

## 7. Quiz

### Question 1:

What will be the output of the following code snippet?

```
Def simple_gen():  
    yield 'Python'  
    yield 'Rocks'
```

```
gen = simple_gen()
print(next(gen))
print(next(gen))
```

- A) Python Rocks
- B) Python So
- C) StopIteration error

### Question 2:

What is an iterator in Python?

- A) A data type that can store multiple items.
- B) An object that can be iterated upon and returns data, one element at a time when `next()` is called on it.
- C) A syntax for handling exceptions.
- D) A module that provides a way to iterate over data structures.

### Question 3:

Which of the following is true about generator functions?

- A) They return a single value using the `return` statement.
- B) They can yield multiple values, one at a time.
- C) They cannot be used in a `for` loop.
- D) They consume more memory than equivalent list comprehensions.

### Question 4:

What advantage do generators have over list comprehensions when dealing with large datasets?

- A) Generators process elements faster than list comprehensions.
- B) Generators enhance readability and are preferred for simple data processing tasks.
- C) Generators yield one item at a time and are more memory-efficient.
- D) Generators have a more straightforward syntax compared to list comprehensions.

### Question 5:

Which Python module provides a collection of tools for handling iterators?

- A) `collections`
- B) `functools`
- C) `itertools`
- D) `operator`

## 8. Homework



## Task 1: Custom Range Generator

**Objective:** Create a generator function that mimics the behavior of Python's built-in range function.

### Requirements:

- The generator should be able to handle the same arguments as `range()`: start, stop, and step.
- It should yield one number at a time in the specified range.
- Include proper handling for negative steps and reverse iteration.

```
# Starter code
def custom_range(start, stop=None, step=1):
    # Your implementation here
    pass
# Example usage
for num in custom_range(3, 15, 3):
    print(num)
```

## Task 2: Log File Parser

**Objective:** Develop a generator that parses a log file and yields dictionaries of log data.

### Requirements:

- Each yielded dictionary should contain the parts of a log line, such as timestamp, log level, and message.
- The generator should handle large files efficiently.
- Write a function to filter yielded log entries by log level (INFO, DEBUG, ERROR).

```
def log_parser(log_file_path):
    pass
def filter_logs(log_generator, log_level):
    pass
logs = log_parser('path/to/log/file')
error_logs = filter_logs(logs, 'ERROR')
for log in error_logs:
    print(log)
```

## Task 3: Batch Processor

**Objective:** Write a generator function that processes items in batches of a specified size.

### Requirements:

- The generator should accept any iterable as input.
- It should yield lists containing a batch of items.
- If the number of items in the last batch is less than the batch size, it should still be yielded.

```
def batch_processor(iterable, batch_size):  
    pass  
for batch in batch_processor(range(10), 3):  
    print(batch)
```

# Lesson 21: Refactoring and Code Review

"Driving continuous evolution and excellence"

## 1. The Importance of Refactoring and Code Review

### 1.1 Refactoring

Refactoring is the process of restructuring code, while not changing its original functionality.

**Why do we need it?**

- Refactoring improves the internal structure of the software.
- Refactored code is easier to understand and and modify.

### 1.2 Code review

A code review is a peer review of code that helps developers ensure or improve the code quality before they merge and ship it.

It involves systematically examining source code, or overlook of the new feature once Merge Request is opened.

**Why do we need it?**

- Fix mistakes overlooked in the initial development phase.
- Improving both the overall quality of the code.
- Enhancing developers' skills.

## 2. Principles of Good Refactoring

Let's take a look on some examples for each purpose of the code refactoring:

### 2.1 Understandability

**Bad Approach:**

```
def calc(x, y, z):  
    return x + y * z
```

This function is not clear in its purpose based on the parameter names and lacks context.

### Good Approach:

```
def calculate_total_price(quantity, price_per_item, tax_rate):  
    return quantity * price_per_item * (1 + tax_rate)
```

Renaming the function and parameters makes the code self-documenting, clarifying its purpose and how it should be used.

## 2.2 Simplicity

### Bad Approach:

```
def process_data(data):  
    if data is not None:  
        if len(data) > 0:  
            # Process data  
            print("Processing...")  
        else:  
            print("Data is empty.")  
    else:  
        print("No data provided.")
```

This nested conditional structure is unnecessarily complex.

### Good Approach:

```
def process_data(data):  
    if not data:  
        print("No data provided or data is empty.")  
        return  
    print("Processing...")
```

Simplifying the conditionals makes the function easier to read and understand, your code works much faster.

## 2.3 Testability

### Bad Approach:

```
class UserManager:  
    def __init__(self):  
        self.users = []  
    def add_user(self, user):  
        self.users.append(user)  
        # Directly sending an email
```

```
send_email(user)
```

This class is hard to test due to the direct dependency on the email sending function.

### Good Approach:

```
class UserManager:
    def __init__(self, email_sender):
        self.users = []
        self.email_sender = email_sender
    def add_user(self, user):
        self.users.append(user)
        # Using dependency injection for email sending
        self.email_sender.send(user)
```

By using dependency injection, the class becomes easier to test, especially if `email_sender` can be mocked.

## 2.4 Performance

### Bad Approach:

```
def find_duplicates(items):
    duplicates = []
    for i in range(len(items)):
        for j in range(i + 1, len(items)):
            if items[i] == items[j] and items[i] not in duplicates:
                duplicates.append(items[i])
    return duplicates
```

This approach has a high computational complexity, making it inefficient for large lists.

### Good Approach:

```
def find_duplicates(items):
    seen = set()
    duplicates = set()
    for item in items:
        if item in seen:
            duplicates.add(item)
        else:
            seen.add(item)
    return list(duplicates)
```

Using sets to track seen items and duplicates significantly improves the function's performance.

## 2.5 Best Practices

1. **Extract Function:** Breaking down large functions into smaller, more manageable ones.
2. **Rename Variable:** Use more descriptive names.
3. **Remove Duplicate Code:** Identify and eliminate repetitive code patterns.
4. **Simplify Conditional Expressions:** Make conditional logic easier to read and understand.

Don't forget to refer to SOLID, DRY, KISS and YAGNI principles during development and refactoring stages.

## 2.6 Code Smells

Code smells are patterns in the code that indicate potential issues or poor design choices. They're not bugs, code can function correctly, but they suggest areas that might benefit from refactoring.

In order to proceed with successful refactoring towards improving the codebase, the best approach will be to recognise code smells as the first step.

Code Smell	Description
Long Methods	Methods that are too lengthy, making them hard to read, understand, and maintain.
Large Classes	Classes that have too many responsibilities, making them complex and difficult to manage.
Duplicate Code	Repetitive code blocks that appear in multiple places, leading to maintenance challenges.
Magic Numbers	The use of unexplained numbers in the code that convey unclear meaning.
Feature Envy	A method that frequently uses data or methods from another class more than its own, indicating misplaced responsibilities.
Switch Statements	Overuse of switch statements or long if-else chains, often a sign that polymorphism could be better utilized.
Primitive Obsession	Using primitive data types instead of small objects for simple tasks (e.g., currency, ranges, special strings for phone numbers, etc.).
Data Clumps	Groups of variables that are passed around together in various parts of the program, suggesting a need for a new object or structure.

Let's create a complex Python application example that includes these code smells.

### Example

```
# Initial version with code smells
class Employee:
```

```

def __init__(self, name, type_code):
    self.name = name
    self.type_code = type_code
def calculate_pay(self, hours_worked, hourly_rate):
    if self.type_code == 1: # Regular employee
        return hours_worked * hourly_rate
    elif self.type_code == 2: # Manager with fixed bonus
        return hours_worked * hourly_rate + 100
    # Additional conditions may be added here for other types
def send_email(self, message):
    print(f"Sending email to {self.name}: {message}")
employee = Employee("John Doe", 1)
pay = employee.calculate_pay(40, 20)
employee.send_email(f"Your pay this period is: {pay}")

```

## Output

```
Sending email to John Doe: Your pay this period is: 800
```

## Code smells

- **Long Method:** `calculate_pay` method contains logic that could be split out.
- **Switch Statements:** The use of `if-elif` chain for type checking.
- **Primitive Obsession:** Using a primitive type (`type_code`) to represent employee type.
- **Feature Envy:** The `send_email` method might be better suited to a class responsible for communication.
- **Large Class & Duplicate Code:** As more methods like `calculate_pay` are added, `Employee` class will grow and duplicate code for different employee types.

## Refactored

```

from abc import ABC, abstractmethod
# Utilizing Polymorphism and SRP
class Employee(ABC):
    def __init__(self, name):
        self.name = name
    @abstractmethod
    def calculate_pay(self, hours_worked, hourly_rate):
        pass
    def send_email(self, message):
        EmailSender.send(self.name, message)
class RegularEmployee(Employee):
    def calculate_pay(self, hours_worked, hourly_rate):
        return hours_worked * hourly_rate
class Manager(Employee):
    def calculate_pay(self, hours_worked, hourly_rate):
        return hours_worked * hourly_rate + 100
class EmailSender:
    @staticmethod
    def send(name, message):
        print(f"Sending email to {name}: {message}")

```

```
# Usage
employee = Manager("John Doe")
pay = employee.calculate_pay(40, 20)
employee.send_email(f"Your pay this period is: {pay}")
```

## Output

```
Sending email to John Doe: Your pay this period is: 900
```

As you can see the output hasn't changed at all, it means that in most cases refactoring is not about changing the actual logic of an application, but the implementation itself.

## Explanation

- **Polymorphism:** Replaced the if-elif chain with polymorphic classes that extend a common `Employee` interface, adhering to the Open/Closed Principle.
- **Eliminated Primitive Obsession:** Instead of using `type_code`, we now have distinct classes for each employee type.
- **Addressed Feature Envy:** The responsibility of sending emails is moved to a dedicated `EmailSender` class, focusing on communication.
- **Simplified Large Class Problem:** By dividing responsibilities among different classes, we reduce the chance of the `Employee` class becoming unwieldy.
- **Reduced Duplicate Code:** Specific behaviors are encapsulated in their respective classes, reducing code duplication.

The previous version of our application worked fine even without refactoring, but after we updated the codebase not only it became more readable, but the structure of our application is much easier to be scaled during further development.

Let's take a look at one more example, I will try to include all code smells which have been discussed already.

## Example

```
class Task:
    def __init__(self, title, due_date, priority):
        self.title = title
        self.due_date = due_date
        self.priority = priority # 1 for high, 2 for medium, 3 for low
    def display_task(self):
        if self.priority == 1:
            priority_str = "High"
        elif self.priority == 2:
            priority_str = "Medium"
        elif self.priority == 3:
            priority_str = "Low"
        print(f"Task: {self.title}, Due: {self.due_date}, Priority: {priority_str}")
class Project:
```

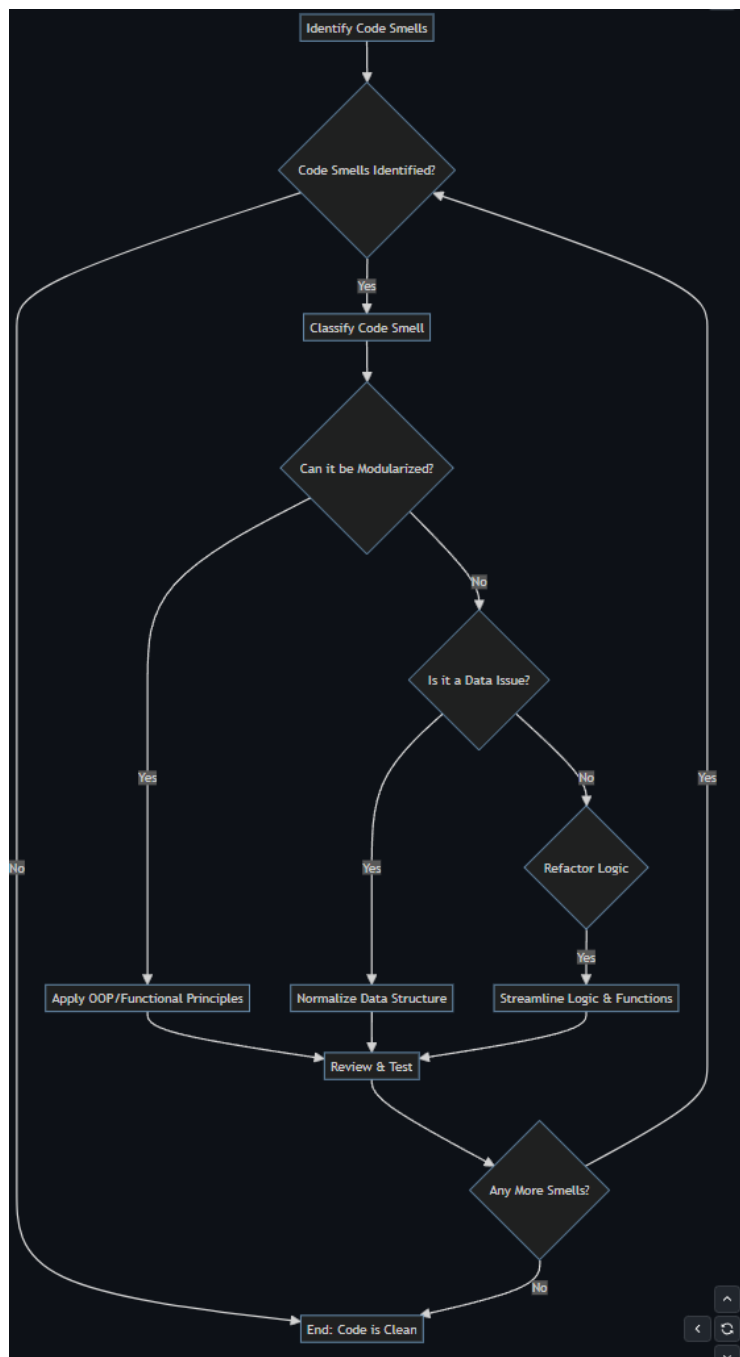


```
def __init__(self, name):
    self.name = name
    self.tasks = []
def add_task(self, title, due_date, priority):
    if not title or not due_date or priority not in [1, 2, 3]:
        print("Invalid task details")
        return
    new_task = Task(title, due_date, priority)
    self.tasks.append(new_task)
def display_project(self):
    print(f"Project: {self.name}")
    for task in self.tasks:
        task.display_task()
```

Let's try to proceed with the following approach of refactoring and identify pain points of our application.

## 2.7 Refactoring workflow

Following the workflow above it took only 30 minutes for me to create a piece of cake from our previous version of application.



## Refactored

```

from enum import Enum, auto
from datetime import datetime

class Priority(Enum):
    HIGH = auto()
    MEDIUM = auto()
    LOW = auto()

class Task:
    def __init__(self, title, due_date, priority):
        self.title = title
        self.due_date = due_date
        self.priority = priority

    def display(self):
        priority_str = self.priority.name.capitalize()

```

```

        print(f"Task: {self.title}, Due: {self.due_date.strftime('%Y-%m-%d')},
Priority: {priority_str}")
class Project:
    def __init__(self, name):
        self.name = name
        self.tasks = []
    def add_task(self, task):
        self.tasks.append(task)
    def display(self):
        print(f"Project: {self.name}")
        for task in self.tasks:
            task.display()

# Usage
project = Project("Website Redesign")
project.add_task(Task("Design Mockups", datetime(2024, 1, 15), Priority.HIGH))
project.add_task(Task("User Testing", datetime(2024, 2, 1), Priority.MEDIUM))
project.display()

```

## Output

```

Project: Website Redesign
Task: Design Mockups, Due: 2024-01-15, Priority: High
Task: User Testing, Due: 2024-02-01, Priority: Medium

```

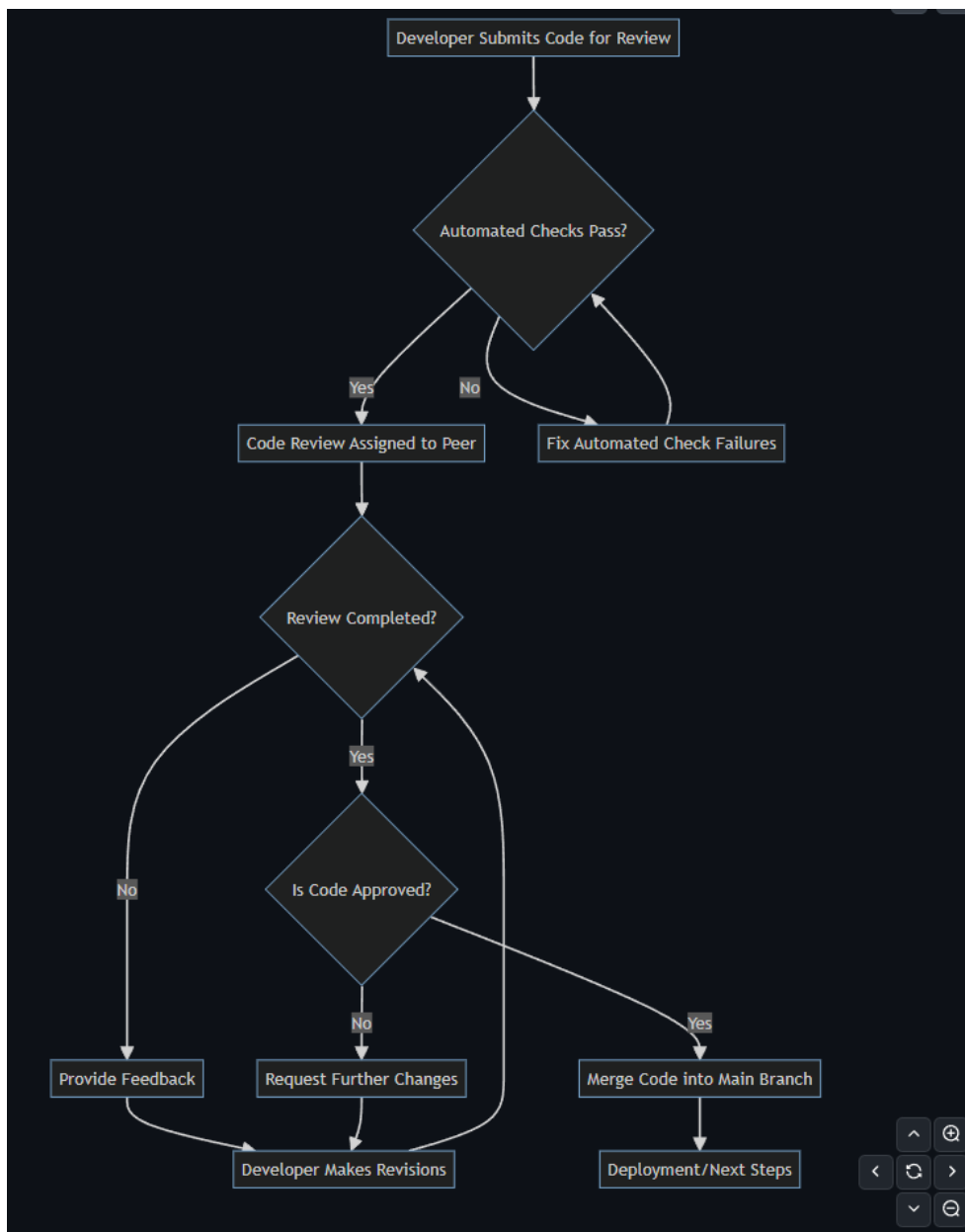
The code becomes cleaner and more maintainable, good job comrades!

## 3. Code reviews

Code review is a systematic examination of source code, intended to find and fix mistakes overlooked in the initial development phase and improve overall code quality.

### 3.1 Code review workflow

The process is carried out by appropriate team members during submitting Pull Requests (PRs) into review according to the company policies.



### 3.2 Best Practices for Conducting Code Reviews

To maximize the effectiveness of code reviews, certain best practices should be followed:

- **Be Constructive:** Feedback should be positive, specific, and focused on the code. Suggest improvements and explain the reasoning behind them.
- **Keep It Small:** Aim for brief, focused code reviews to ensure attention to detail.
- **Automate Where Possible:** Utilize automated tools for code formatting, linting, and basic error detection to save time for reviewers.
- **Embrace Feedback:** Encourage an open, respectful dialogue about suggested changes.

My personal checklist of code review for the majority of the projects will be:

#### 1. Correctness:

- Does the code do what it's supposed to do?
- Are there any logical errors or bugs?

## 2. **Readability:**

- Is the code clear and understandable?
- Are variable and method names descriptive and appropriate?
- Is the code well-commented, providing context where necessary? (Does the code speak for itself?)

## 3. **Architecture and Design:**

- Is the code consistent with the overall project architecture?
- Could the design be simplified or made more efficient?

## 4. **Performance:**

- Are there any obvious performance issues?
- Can any parts of the code be optimized?

## 5. **Security:**

- Does the code introduce any security vulnerabilities?
- Are data and sensitive information handled securely?

## 6. **Testing:**

- Are there sufficient unit and integration tests?
- Do tests cover edge cases as well as typical use cases?
- Are tests clear and meaningful?

## 7. **Documentation:**

- Is the code adequately documented, both in-line and in external documents?
- Does the documentation accurately reflect the current state of the code?

## 8. **Consistency:**

- Is the code consistent with the project's coding standards and conventions?
- Does it follow the established practices for formatting, naming, and structuring?

As well you would want to understand if the changes can be integrated smoothly with the existing codebase and the code for a new feature will be easy to maintain and extend. Though sometimes it is not really obvious and we can't predict what the future holds for the project. Just don't forget that you can use a refactoring techniques at any time described above.

**Note:** You would want do develop your own code review checklist based on common issues relevant to your project

**Important:** Don't obsess on the perfect code and refer to the checklist all the time. Once you understand the key principles that process will be handled automatically by you during each review you conduct.

Use your time wisely and don't forget about deadlines, as programmers say if it works, don't touch it. :)

### 3.3 Practice

Let's take a look on real cases, instead of theory, we'll come into practice straightforward:

#### Case 1: Performance Optimization

**Situation:** A developer submitted a pull request (PR) with a new feature implementation. The code was functional but not optimized for performance.

##### Before Review

```
# Inefficient search for duplicates in a list
def find_duplicates(input_list):
    duplicates = []
    for i in range(len(input_list)):
        for j in range(i + 1, len(input_list)):
            if input_list[i] == input_list[j] and input_list[i] not in
duplicates:
                duplicates.append(input_list[i])
    return duplicates
input_list = [1, 2, 3, 2, 1, 5, 6, 5, 5, 5]
print(find_duplicates(input_list))
```

##### Review Process:

- During the code review, a peer noticed that the new feature relied on a nested loop, leading to  $O(n^2)$  time complexity for a task that could be accomplished with  $O(n)$ .
- The reviewer suggested leaving a comment on the pull request using a hashmap a.k.a set in Python to reduce computational complexity.

##### After Review

```
def find_duplicates(input_list):
    seen = set()
    duplicates = set()
    for item in input_list:
        if item in seen:
            duplicates.add(item)
        else:
            seen.add(item)
    return list(duplicates)
input_list = [1, 2, 3, 2, 1, 5, 6, 5, 5, 5]
print(find_duplicates(input_list))
```

**Outcome:** The developer revised the implementation based on the feedback. This optimization led to a noticeable improvement in the feature's performance, especially with large data sets.

## Case 2: Improving Code Maintainability

**Situation:** A developer's PR involved a complex algorithm that was hard to understand at first glance.

### Before Review

```
def complex_algorithm(data):  
    # Complex operations  
    result = data[0]  
    for i in range(1, len(data)):  
        result = result ^ data[i]  
    return result  
print(complex_algorithm([1, 2, 3, 4]))
```

### Review Process:

- The code review emphasized the lack of comments and documentation for the complex parts of the algorithm.
- The reviewer requested more detailed comments and the inclusion of a reference to the algorithm's source or explanation.

### After Review

```
# Well-documented complex algorithm  
def complex_algorithm(data):  
    """  
    Performs a bitwise XOR operation on all elements of the input list.  
  
    The algorithm iterates through the list, applying a bitwise XOR (^)  
operation  
between the current result and each element.  
Args:  
    data (list of int): A list of integers to process.  
Returns:  
    int: The result of the bitwise XOR operation on all list elements.  
    """  
    # Initialize result with the first list element  
    result = data[0]  
    # Iterate over the list starting from the second element  
    for i in range(1, len(data)):  
        result = result ^ data[i] # Apply bitwise XOR operation  
    return result  
print(complex_algorithm([1, 2, 3, 4]))
```

**Outcome:** The developer added comprehensive comments and documentation, making the code easier for future maintainers to understand and modify if necessary.

## 3.4 Conclusion

The key to success in everything is to have a strong verbal communication between people. Don't hesitate to ask reviewers about anything on the PR, don't be scared to defend your point of view during the code review, of course try reasonably explain it with good arguments. And keep learning continuously as one person can't know anything and only group of enthusiastic people build the future together.

## 4. Application Development Life Cycle

Application development lifecycle is a well-structured set of steps outlining app planning, creation, testing, deploying, and maintenance.

It is crucial for developers to understand the general workflow of app development.

```
graph LR
    A[Planning and Analysis] --> B[Design Phase]
    B --> C[Implementation]
    C --> D[Testing]
    D --> E[Deployment]
    E --> F[Maintenance]
    F --> G{Feedback Loop}
    G -->|Incorporate Feedback| A
    G -->|New Requirements| A
    G -->|Continuous Monitoring| E
    G -->|Security Assessments| D
```

Let's make a quick overview of each stage developing a Contact Management System

### 4.1 Planning and Analysis

During this stage we gather requirements and define the project scope.

- **Objective:** Develop a scalable Contact Management System to store, retrieve, update, and delete contact information.

#### Requirements



- Store contacts with details such as name, email, phone number, and address.
- Support CSV and JSON formats for data import and export.
- Provide search functionality to find contacts by name.
- **Risk Assessment:** Consider data integrity, scalability, and user privacy.

## 4.2 Design Phase

Based on the requirements, the system architecture and user interfaces are designed. This phase includes both high-level architectural planning and detailed design of the application components.

**Note:** In complex applications you would want to create a separate documentation for different modules, preferably with visualisation in diagrams and graphs.

- **Architecture:** Use a modular approach with separate classes for `Contact` and `ContactManager`.
- **ProjectStructure:** Split the application within submodules.
- **Data Structure:** Use a `list` to store contacts and `dictionaries` for individual contact details.
- **Data Handling:** Implement modules for CSV and JSON data import/export.

### Project Structure

```
/contact_management_system

    /src

        __init__.py

        contact.py

        contact_manager.py

        data_handler.py

    /tests

        __init__.py

        test_contact_manager.py

    requirements.txt
```

## 4.3 Implementation

This is where developers write the application, adhering to coding standards and best practices. Have fun :3

**contact.py** - Defines a Contact class.

```
class Contact:
    def __init__(self, name, email, phone, address):
        self.name = name
        self.email = email
        self.phone = phone
        self.address = address
```

**contact\_manager.py** - Manages a list of contacts, including add, update, delete, search, and data import/export functionalities.

```
import csv, json
from contact import Contact
class ContactManager:
    def __init__(self):
        self.contacts = []
    def add_contact(self, contact):
        self.contacts.append(contact)
    def find_contact(self, name):
        return [contact for contact in self.contacts if contact.name == name]
    # Heh, re-write it to match SOLID
    # Potentially at some point the team of developers decide to refactor an
    # application, this will be parallel stage with maintainance
    def export_to_csv(self, filepath):
        with open(filepath, 'w', newline='') as file:
            writer = csv.writer(file)
            writer.writerow(["Name", "Email", "Phone", "Address"])
            for contact in self.contacts:
                writer.writerow([contact.name, contact.email, contact.phone,
contact.address])
    def import_from_csv(self, filepath):
        with open(filepath, 'r') as file:
            reader = csv.DictReader(file)
            for row in reader:
                self.add_contact(Contact(row['Name'], row['Email'],
row['Phone'], row['Address']))
```

## 4.4 Testing

- **Unit Testing:** Write tests for each functionality in ContactManager — adding, updating, finding, and deleting contacts, as well as importing and exporting data.
- **Integration Testing:** Test the system as a whole to ensure different parts work together seamlessly.

**Note:** Unless we are doing TDD which is highly recommended, we would have 2 stages of testing. The first one is before implementation and the second is after.

**test\_contact\_manager.py**

```
import pytest
```

```

from src.contact import Contact
from src.contact_manager import ContactManager
def test_contact_creation():
    """Test the creation of a Contact object with valid attributes."""
    contact = Contact(name="John Doe", email="johndoe@example.com",
phone="1234567890", address="123 Main St")
    assert contact.name == "John Doe"
    assert contact.email == "johndoe@example.com"
    assert contact.phone == "1234567890"
    assert contact.address == "123 Main St"
def test_invalid_email_contact():
    """Test Contact creation with an invalid email address."""
    with pytest.raises(ValueError):
        Contact(name="Invalid Email", email="invalid", phone="1234567890",
address="123 Main St")
def test_add_contact():
    """Test adding a contact to the ContactManager."""
    manager = ContactManager()
    contact = Contact(name="Jane Doe", email="janedoe@example.com",
phone="0987654321", address="321 Main St")
    manager.add_contact(contact)
    assert len(manager.contacts) == 1
    assert manager.contacts[0].name == "Jane Doe"
def test_find_contact():
    """Test finding a contact by name."""
    manager = ContactManager()
    contact = Contact(name="Find Me", email="findme@example.com",
phone="1231231234", address="100 Main St")
    manager.add_contact(contact)
    found = manager.find_contact("Find Me")
    assert len(found) == 1
    assert found[0].email == "findme@example.com"
def test_duplicate_contact():
    """Test adding a duplicate contact."""
    manager = ContactManager()
    contact1 = Contact(name="Unique", email="unique@example.com",
phone="1112223333", address="200 Main St")
    manager.add_contact(contact1)
    contact2 = Contact(name="Unique", email="unique2@example.com",
phone="4445556666", address="300 Main St")
    with pytest.raises(ValueError):
        manager.add_contact(contact2)

```

## 4.5 Deployment

- Prepare a deployment checklist, including environment setup, dependency installation, and initial data load.
- Choose a deployment strategy that minimizes downtime and ensures data integrity.

Unfortunately, it is a very big topic to cover in this book.

## 4.6 Maintenance

- Monitor the application for issues, and regularly update the documentation.
- Incorporate user feedback to improve the system and add new features as required.

You can use logging for that ;)

## 4.7 Feedback Loop

- Gather user feedback through surveys, bug reports, and feature requests.
- Regularly review and analyze feedback to identify areas for improvement or new requirements.
- Implement changes based on feedback in iterative development cycles.

Repeat stage 4.1 - 4.7 for new feature to be developed. Don't forget to refer to refactoring at least on the quarterly basis.

**Attention:** Combining all techniques described within this section can lead to a high quality code!

## 5. Homework

Refactor all your projects based on the requirements below.

### 1. Code Structure:

- Your code should be neatly structured and divided into functions or classes as appropriate.
- Include comments to describe your logic and thought process.

### 2. Testing:

- Include test cases to demonstrate that your applications work correctly.
- Don't forget to add test cases which suppose to break your application, ensuring that it exceptions are handled.

### 3. Performance:

- Try to optimise and speed up your code during review.
- Find yourself a person who could review your code and provide mentorship support.

**IMPORTANT:** Follow Application Development Life Cycle during refactoring and add new functionality to your projects. Good Luck!

# Lesson 22. Documenting Python Code

"Just cheat sheets for you and your clients."

## 1. Introduction

Well-documented code can save countless hours of debugging and troubleshooting by providing clear guidance and explanation of the code's functionality and usage.

### 1.1 Types of docs

**Internal Documentation:** Consists of comments and docstrings within the codebase that explain the purpose, logic, and usage of various parts of the program.

**External Documentation:** Includes README files, documentation of the project functionality, user manuals, and developer guides that describe how to use the software, how it works, and how to contribute to it.

I recommend to write both types of docs to make your app clear as possible either for the user or team.

## 2. Code Comments and Docstrings

### 2.1 Inline comments

As you now it already, comments in Python are preceded by a hash (#) symbol and should explain the "why" behind a block of code rather than the "how".

#### Best Practices

- Inline comments are placed on the same line as a statement.
- They should be used to clarify complex pieces of code.
- They should be meaningful and add value to the understanding of the code.

#### Example

In the bad approach, the comments are redundant, merely repeating what the code does.

```
x = 5 # Assign 5 to x
y = x + 2 # Add 2 to x and assign to y
```

#### Example

The good approach, however, provides a rationale for why the operation is performed, adding value to the code's readability.

```
x = 5
y = x + 2 # Compensate for border width in calculation
```

The biggest Python's achievement is that in the majority of cases the code is self-explanatory, but it doesn't mean that we have to forget about inline comments or docstrings.

## 2.2 Docstrings

**Docstring Conventions and Standards:** Docstrings provide a built-in way of documenting Python modules, functions, classes, and methods. They are enclosed in triple quotes ("""") and should follow the PEP 257 conventions.

**IMPORTANT:** A good docstring should describe what the function/class does, its arguments, return values, and any exceptions raised.

### Example

Avoid doing this!

```
def add(a, b):
    # Nothing here
    return a + b
```

In the case above we are not really sure which types should we pass to the `add` function, that is probably one of the biggest disadvantage.

## 2.3 Docstring Components

1. **Brief Description:** A concise summary of what the function does. This should be in the imperative mood (e.g., "Return" instead of "Returns").
2. **Parameters Section:** Lists each parameter name, expected type, and a short description. While the types can also be indicated in the function signature (as of Python 3.5+), repeating them in the docstring can enhance readability and clarity.
3. **Returns Section:** Describes the return value's type and purpose.

Check out the following examples and notice how understandability has been improved significantly.

### Example

```
def add(a, b):
    """
    Add two numbers and return the result.
    Parameters:
    a (int): The first number.
    b (int): The second number.
```

```
Returns:
int: The sum of a and b.
"""
return a + b
```

## Example

```
def divide(dividend, divisor):
    """
    Divide the dividend by the divisor and return the quotient and remainder.
    Parameters:
    dividend (int): The number to be divided.
    divisor (int): The number by which to divide.
    Returns:
    tuple: A tuple containing the quotient and remainder (quotient, remainder).
    Raises:
    ValueError: If the divisor is zero.
    """
    if divisor == 0:
        raise ValueError("Divisor cannot be zero.")
    quotient = dividend // divisor
    remainder = dividend % divisor
    return quotient, remainder
```

## Example

```
def find_max(numbers):
    """
    Find and return the maximum number in a list.
    Parameters:
    numbers (list of int): The list of numbers to search.
    Returns:
    int: The maximum number in the list. Returns None if the list is empty.
    Example:
    >>> find_max([1, 3, 2])
    3
    """
    return max(numbers) if numbers else None
```

Revisit your existing projects and update them with docstrings using right now!

## 3. Annotations

Python's type system, while dynamically typed at runtime, supports optional type hints that enable static type checking. This feature, introduced in Python 3.5 through [PEP 484](#), allows developers to annotate their code with type hints, making it more readable, maintainable, and less prone to errors.

The main idea behind annotations is that we define the expected data type which should be passed to the function/method or the variable.

**VERY IMPORTANT:** Annotations don't guarantee that exactly this type will be passed, it is not about enforcement, but more about documentation.

## 3.1 Basic Types

These are the foundational types that correspond to Python's built-in types:

- **int:** For integers.
- **float:** For floating-point numbers.
- **bool:** For Boolean values (True or False).
- **str:** For strings.
- **bytes:** For byte sequences.

### Example

```
name: str = "Alice"
age: int = 30
is_student: bool = True
```

## 3.2 Composite Types

Composite types, also known as collection types, allow you to specify the type of elements in a collection.

- **list[Type]:** A list where all elements are of the specified type.
- **tuple[Type, ...]:** A tuple with specified element types. Use `Tuple[Type, ...]` for variable-length tuples with elements of the same type.
- **dict[KeyType, ValueType]:** A dictionary with specified types for keys and values.
- **set[Type]:** A set where all elements are of the specified type.

### Example:

```
names: list[str] = ["Alice", "Bob", "Charlie"]
coordinates: tuple[int, int, int] = (10, 20, 30)
student_grades: dict[str, float] = {"Alice": 85.5, "Bob": 92.0}
unique_numbers: set[int] = {1, 2, 3, 4, 5}
```

## 3.3 Specialized Types

Python's typing module also includes more specialized types for more complex scenarios:

- **Optional[Type]:** Indicates a variable that can be of a specified type or `None`.
- **Union[Type1, Type2, ...]:** Indicates a variable that can be any one of the specified types.
- **Callable[[ArgType1, ArgType2, ...], ReturnType]:** Represents a callable (function or object with `__call__`) with specified argument and return types.
- **Any:** A special type indicating that the variable can be of any type. Use sparingly, as it essentially opts out of type checking for the variable.



### Example:

```
from typing import Optional, Union, Callable, Any
def add(a: int, b: int) -> int:
    return a + b
maybe_number: Optional[int] = None
string_or_number: Union[str, int] = 5
calculator: Callable[[int, int], int] = add
anything: Any = "Hello" # Can be any type
```

**Note:** Using Any is an extremely bad practice which can lead to unpleasant consequences and bugs, if we do some specific operation which are related to specific data types.

## 3.4 Type Aliases

Type aliases allow you to define custom types for complex type hints, improving code readability.

### Example:

```
Vector = list[float]
Point = tuple[float, float, float]
velocity: Vector = [1.5, -2.0, 0.0]
location: Point = (10.0, 20.5, 30.0)
```

## 3.5 Advanced Type Hints

**Generics:** Python's typing module allows for the definition of generic types, making it possible to create container types that can hold objects of any type, specified at runtime. This is particularly useful for classes that act as wrappers or containers for other objects.

Somewhere here, I will refer to word "type checker" which will be covered in the next section. For now, just take a look and try to understand the purpose of these annotations.

### Example

```
from typing import TypeVar, Generic
T = TypeVar('T') # Declare type variable
class Box(Generic[T]):
    def __init__(self, item: T) -> None:
        self.item = item
    def get(self) -> T:
        return self.item
box = Box[int](123) # Box containing an integer
print(box.get()) # Outputs: 123
```

**NewType:** You can create distinct types that are treated as separate types by static type checkers but are runtime-equivalent to their base types. This is useful for adding semantic meaning to base types.

### Example

```
from typing import NewType
UserId = NewType('UserId', int)
def get_user_name(user_id: UserId) -> str:
    return "John Doe" # Placeholder for actual lookup
user_id = UserId(5232)
print(get_user_name(user_id))
```

**Literal Types:** Literal types indicate that a variable or parameter can only have specific literal values. This is particularly useful when a function accepts a limited set of string or integer values.

### Example

```
from typing import Literal
def handle_status(status: Literal['open', 'closed']) -> None:
    print(f"Handling a {status} status")
handle_status('open') # Valid
handle_status('pending') # Type checker will flag this as an error
```

**TypedDict:** For dictionaries with a fixed set of keys, where each key has a specific type, TypedDict offers a way to specify type hints for each key-value pair explicitly.

### Example

```
from typing import TypedDict
class User(TypedDict):
    name: str
    age: int
user: User = {'name': 'Alice', 'age': 30} # Type checker enforces dict structure
```

**Protocols:** Introduced in Python 3.8, Protocols allow for duck typing, defining a set of methods that a class must implement without specifying a specific inheritance hierarchy.

### Example:

```
from typing import Protocol
class SupportsClose(Protocol):
    def close(self) -> None:
        ...
    def close_resource(resource: SupportsClose) -> None:
        resource.close()
class MyResource:
```

```
def close(self) -> None:
    print("Resource closed")
close_resource(MyResource()) # Type checker ensures MyResource has a close
method
```

## 3.6 Setup the Type Checker

When it comes to enforcing and checking the type hints in your Python code, [Mypy](#) is the go-to tool. Mypy is a static type checker that helps you catch type errors before runtime.

So basically all those annotations which we have added in the previous sections will be validated by `mypy`. In case of errors or incorrect usage of functions, the type checker will be able to spot them. Thus, developers can prevent bugs with incorrect types provided to the function.

**NOTE:** Defining types can speed up your code!

### Quick Start

1. **Installation:** Install Mypy with pip:

```
pip install mypy
```

2. **Basic Usage:** To check a single Python file for type errors, run:

```
mypy your_script.py
```

Mypy will analyze your code and report any type inconsistencies based on the annotations you've provided.

3. **Checking Multiple Files:** You can also check multiple files or entire directories by listing them:

```
mypy file1.py file2.py directory/
```

4. **Configuration:** For more complex projects, Mypy can be configured via a `mypy.ini` or `pyproject.toml` file in your project's root directory.

```
[mypy]
python_version = 3.8
ignore_missing_imports = True
```

In some cases you would want to run the commands with some of these flags:

- `--ignore-missing-imports`: Ignores errors about missing `import` statements.
- `--strict`: Enables all of Mypy's strictness flags for thorough checks.
- `--follow-imports=silent`: Follows import statements but doesn't check the imported modules.
- `--exclude`: Excludes specific files or directories from being checked.

I would recommend to refer to the official documentation for more details. It is a great tool which helps you to ensure that typing is defined correctly across the project.

We won't focused on this part too much, as it's beyond of the scope of this lesson, but it's great to know that such tool exists.

## 4. External Documentation

### 4.1 README

A README file is often the first document readers encounter in your project. It's not just an introduction; it's your project's handshake with the outside world. A well-crafted README effectively communicates the purpose of your project, how to use it, and how others can contribute.

#### Components

1. **Project Name and Logo**: Start with the project's name, and if you have one, include a logo.
2. **Introduction**: A brief description of the project and its purpose.
3. **Installation Instructions**: Clear, concise steps to get your project running on another machine.
4. **Usage**: Examples of how to use your project, including code snippets and command-line examples.
5. **Contributing**: Guidelines on how others can contribute to your project, including coding standards, test procedures, and how to submit pull requests.
6. **License**: Information on the project's license, allowing others to understand how they can use, modify, and distribute your work.
7. **Credits**: Acknowledgments of contributors, external resources, or dependencies.

I have been using this template for the whole life and now would love to share it with you! If you have any suggestions, LMK what can be improved.

#### Example

```
# Project Name
![[Project Logo](path/to/logo.png)]
## Introduction
Briefly introduce your project and its goals.
## Installation
Provide a step-by-step guide to install your project:
E.g
git clone https://yourproject.git
cd yourproject
pip install -r requirements.txt
## Usage
Showcase how to use your project with code snippets or command-line examples:
```

## `## Contributing`

`Explain how others can contribute to your project. Include links to issues, coding standards, and contribution guidelines.`

## `## License`

`State the license under which your project is released, for example, MIT or GPL-3.0.`

## `## Credits`

`Acknowledge contributors, third-party libraries, and any other resources that helped your project.`

You can use either my template to describe your project in README or come up with a new one for your specific needs, again, there is no strict enforcements on the format, just make sure it's consistent, well-written and described in details.

## 4.2 End-Users

Try to include the following into your docs for the end user:

1. **Getting Started:** A beginner-friendly introduction to the project, highlighting basic functionalities and simple use cases.
2. **Step-by-Step Guides:** Detailed tutorials addressing specific tasks or problems, guiding the user from start to finish.
3. **FAQs:** A list of frequently asked questions (and answers) that users may have when using your project.

***NOTE:*** You will need to find out about hostings and documentation builder utils such as `sphinx/mcdocs` and `readthedocs` to expose your documentation into the world.

Don't forget to regularly update the documentation to reflect changes in the project. Be up to date!

## 5. Homework

Update your existing projects with the following requirements:

1. Add type annotations to your functions and methods to clearly specify expected input types and return types.
2. Ensure all functions, classes, and modules have descriptive docstrings that follow the conventions discussed.
3. Create or update a `README.md` file for each project with the following sections and implement internal documentation.
4. Try writing documentation for the person who will use your application as an end user.

# Lesson 23: Regular Expressions

The power of pattern matching with precision.

## 1. Regular Expressions

Regular expressions (`regex` or `regexp`) are a powerful tool for processing text.

They allow you to specify a pattern of text to search for in a string.

### 1.1 Use cases

Generally, we need regular expressions for string data and the following use cases:

Use Case	Description	Notes/Examples
Validation	Ensuring that strings match a specific format to ensure data integrity and consistency.	Validating formats like email addresses, phone numbers, URLs.
Search and Replace	Finding and replacing substrings within a larger text body.	Useful in code editing, data cleaning, and log processing.
Data Extraction	Extracting information from structured text for analysis or format migration.	Parsing data from log files, spreadsheets, or HTML documents.
Text Parsing	Splitting text into tokens or segments based on patterns.	Aids in analyzing complex strings or constructing parsers.
Complex Pattern Matching	Identifying patterns within text not easily described by standard string methods.	Identifying specific word sequences or characters with variable amounts of whitespace.

#### Example

I want to scare you in advance and show a complicated regular expression used to validate an email address, so that you know what you could expect from the lesson:

```
import re
# Email validation pattern
email_pattern = r"^[a-zA-Z0-9_+-.]+@[a-zA-Z0-9-]+\.[a-zA-Z0-9-]+$"
email = "example@test.com"
# Check if the pattern matches the email
if re.match(email_pattern, email):
    print("Valid email address.")
else:
    print("Invalid email address.")
```

It's extremely important and useful tool to have in a programming arsenal, but don't get scary we will find out how to work with regex very soon.

## 2. Basic Patterns

Regular expressions use a combination of literal characters and special characters to define patterns for matching strings.

For working with regular expressions in Python we would need to import and use `re` module. During this lesson, I will try to introduce some functionality from this module and explain everything within inline comments above.

### 2.1 Literal Characters

Literal characters are the simplest form of pattern matching in regular expressions.

They match exactly the characters specified in the regex pattern. For instance, the regex pattern `cat` will match the string `"cat"` in any larger string.

#### Example

```
import re
text = "The cat sat on the mat."
pattern = "cat"
# Finding "cat" in the text
match = re.search(pattern, text)
if match:
    print("Match found:", match.group())
else:
    print("No match found.")
```

#### Output

```
Match found: cat
```

#### Assignment 1:

Find words UK and Britain in the following poem.

#### Poem

```
poem =
"""
In the heart of the UK, where stories entwined,
Across the grand landscapes, where the sun dares to shine.
A tapestry rich, in green and in grey,
Where history whispers, in Britain, they say.
A land where the past and future remain.
```

## 2.2 Special Characters and Sequences

Special characters and sequences represent specific instructions in regex (I mean that how the regex itself should look like).

Character	Description	Example Pattern	Example Match
.	(Dot) Matches any single character except newline (\n).	a.c	Matches "abc", "arc", "a3c".
^	(Caret) Matches the start of a string.	^Hello	Matches "Hello" at the beginning of a string.
\$	(Dollar) Matches the end of a string.	end\$	Matches "The end" in "This is the end".
*	(Asterisk) Matches 0 or more occurrences of the preceding element.	bo*	Matches "boooo" in "A ghost said boooo".
+	(Plus) Matches 1 or more occurrences of the preceding element.	a+	Matches "a" and "aaaa" in "caaaaaat".
?	(Question Mark) Makes the preceding element optional, matching 0 or 1 occurrence.	colou?r	Matches both "color" and "colour".
{n}	(Curly Brackets) Matches exactly n occurrences of the preceding element.	a{2}	Only matches "aa" in "caaat".

Let's use a regular expression to find all instances that match a pattern in a given text. We'll combine several special characters to demonstrate their usage.

**IMPORTANT:** Try not to move to the explanation parts for the following examples, and based on the table, identify what exactly this pattern wants to achieve.

### Example

**Pattern:** ^T.\*end\$



**Text:** "The quick brown fox jumps over the lazy dog to reach the end"

```
import re
text = "The quick brown fox jumps over the lazy dog to reach the end"
pattern = "^T.*end$"
# Using re.findall to search for the pattern
matches = re.findall(pattern, text)
print("Matches found:", matches)
```

## Output

```
Matches found: ['The quick brown fox jumps over the lazy dog to reach the end']
```

## Explanation

This pattern seeks strings that start with "T", followed by any characters (including none), and end with "end".

## Example

**Pattern:** ^a.\*z\$

**Text:** "A to z"

```
import re
text = "A to z"
pattern = "^a.*z$"
matches = re.findall(pattern, text, re.IGNORECASE)
print("Matches found:", matches)
```

## Output

```
Matches found: ['A to z']
```

## Explanation

This regex pattern is designed to find strings that start with "a" and end with "z", with any characters in between.

## Example

**Pattern:** \d+

**Text:** "There are 123 apples"

```
import re
text = "There are 123 apples"
pattern = "\d+"
matches = re.findall(pattern, text)
print("Matches found:", matches)
```

## Output

```
Matches found: ['123']
```

## Explanation

This regex pattern aims to find one or more digits in the text.

## Example

**Pattern:** `f. ?o`

**Text:** "The quick brown fo jumps over the lazy dog to reach the end. ffo also matches."

```
import re
text = "The quick brown fo jumps over the lazy dog to reach the end. ffo also matches."
pattern = "f. ?o"
matches = re.findall(pattern, text)
print("Matches found:", matches)
```

## Output

```
Matches found: ['fo', 'ffo']
```

## Explanation

The pattern `f. ?o` matches strings that start with "f", followed by This regex pattern is designed to find "fo" with zero or one character between "f" and "o".

# 3. Character Classes

Character classes allow you to match specific sets of characters within a string. There are 2 types of character classes: predefined and custom.

Let's take a look on them:

## 3.1 Predefined

That is the default way, I would even say, a built in option of working regular expressions

Class	Description	Equivalent
\d	Matches any digit	[0-9]
\D	Matches any non-digit character	[^0-9]
\s	Matches any whitespace character (including spaces, tabs, and line breaks)	
\S	Matches any non-whitespace character	
\w	Matches any word character (letters, digits, and underscores)	
\W	Matches any non-word character	

## 3.2 Custom

Class	Description
[abc]	Matches any one of the characters a, b, or c.
[^abc]	Matches any character that is not a, b, or c.
[a-z]	Matches any lowercase letter.
[A-Z]	Matches any uppercase letter.
[0-9]	Matches any digit (same as \d).

To be honest, there is nothing much can be told in terms of theory, anywhere across this lesson. Let's dive into practice!

### Example

**Pattern:** \d{2,4}

**Text:** "The year 2023 marks the event."

```
import re
text = "The year 2023 marks the event."
pattern = "\d{2,4}"
matches = re.findall(pattern, text)
print("Matches found:", matches)
```

## Output

```
Matches found: ['2023']
```

## Explanation

This regex pattern aims to find sequences of digits where there are 2 to 4 digits in a row, matching years or other numerical data within that range.

## Example

**Pattern:** `[A-Za-z]+`

**Text:** "Regex101 is a great tool!"

```
import re
text = "Regex101 is a great tool!"
pattern = "[A-Za-z]+"
matches = re.findall(pattern, text)
print("Matches found:", matches)
```

## Output

```
Matches found: ['Regex', 'is', 'a', 'great', 'tool']
```

## Explanation

The pattern `[A-Za-z]+` matches one or more consecutive letters, capturing words in the sentence.

## Example 3.5

**Pattern:** `^[^0-9\s]+`

**Text:** "Error 404: Not Found."

```
import re
text = "Error 404: Not Found."
pattern = "[^0-9\s]+"
matches = re.findall(pattern, text)
print("Matches found:", matches)
```

## Output

```
Matches found: ['Error', ':', 'Not', 'Found.']
```

## Explanation

Matches sequences of characters that are not digits or whitespaces.

### Example 3.6

**Pattern:** `\d+`

**Text:** "There are 15 apples in 4 baskets."

```
import re
text = "There are 15 apples in 4 baskets."
pattern = "\d+"
matches = re.findall(pattern, text)
print("Matches found:", matches)
```

## Output

```
Matches found: ['15', '4']
```

## Explanation

Find one or more digits in the text, effectively extracting all numerical values.

---

### Example

**Pattern:** `\w+`

**Text:** "Hello, world! Regex is amazing."

```
import re
text = "Hello, world! Regex is amazing."
pattern = "\w+"
matches = re.findall(pattern, text)
print("Matches found:", matches)
```

## Output

```
Matches found: ['Hello', 'world', 'Regex', 'is', 'amazing']
```

## Explanation

Matches sequences of word characters (letters, digits, underscores), capturing words in the sentence.

## Example

**Pattern:** `[aeiou]`

**Text:** "Regular expressions are powerful."

```
import re
text = "Regular expressions are powerful."
pattern = "[aeiou]"
matches = re.findall(pattern, text)
print("Matches found:", matches)
```

## Output

```
Matches found: ['e', 'u', 'a', 'e', 'e', 'i', 'o', 'a', 'e', 'o', 'e', 'u']
```

## Explanation

Matches any vowel in the text, targeting specific subsets of characters.

## 4. Quantifiers

Quantifiers in regular expressions define how many instances of a character, group, or character class must be present for a match to occur.

Quantifier	Description	Example Pattern	Matches
*	Matches 0 or more occurrences of the preceding element.	<code>a*b</code>	"b", "ab", "aab", "aaab", ...
+	Matches 1 or more occurrences of the preceding element.	<code>a+b</code>	"ab", "aab", "aaab", ...
?	Matches 0 or 1 occurrence of the preceding element, making it optional.	<code>a?b</code>	"b", "ab"
<code>{n}</code>	Matches exactly <code>n</code> occurrences of the preceding element.	<code>a{2}b</code>	"aab"
<code>{n,}</code>	Matches <code>n</code> or more occurrences of the preceding element.	<code>a{2,}b</code>	"aab", "aaab", "aaaab", ...
<code>{n,m}</code>	Matches between <code>n</code> and <code>m</code> occurrences	<code>a{2,3}b</code>	"aab",

Quantifier	Description	Example Pattern	Matches
	of the preceding element, inclusive.		"aaab"

## 4.1 Greedy vs Lazy Quantification

Quantifiers are greedy by default, meaning they match as many occurrences of the pattern as possible.

For a better performance, we could make them lazy by appending a `?` to them, causing them to match as few characters as needed for the pattern to succeed.

I guess, let's dive into practice again?

### Example

**Pattern:** `a.*c`

**Text:** "abcabc"

```
import re
text = "abcabc"
pattern = "a.*c"
matches = re.findall(pattern, text)
print("Matches found:", matches)
```

### Output

```
Matches found: ['abcabc']
```

### Explanation

The greedy quantifier `.*` matches as many characters as possible between `a` and `c`, capturing the entire string "abcabc".

### Example

**Pattern:** `a.*?c`

**Text:** "abcabc"

```
import re
text = "abcabc"
```

```
pattern = "a.*?c"
matches = re.findall(pattern, text)
print("Matches found:", matches)
```

## Output

```
Matches found: ['abc', 'abc']
```

## Explanation

By making the quantifier lazy (.\*?), the pattern matches as little as possible, resulting in two separate matches "abc" and "abc", instead of the entire string.

---

## Example

**Pattern:** a{2,3}

**Text:** "aaaaa"

```
import re
text = "aaaaa"
pattern = "a{2,3}"
matches = re.findall(pattern, text)
print("Matches found:", matches)
```

## Output

```
Matches found: ['aaa', 'aa']
```

## Explanation

Matches between 2 and 3 occurrences of a. Actually, that is a great example which shows how to specify exact or ranges of repetitions in a pattern itself.

# 5. Anchors and Boundaries

Anchors and boundaries do not match characters but rather match positions within the input text.

They are used to assert that the required match is at a particular position in the text.

## 5.1 Boundaries



The word boundary anchor `\b` is used to denote the boundaries of words. It allows a regular expression to specify that a given pattern must occur at the beginning or end of a word within the text.

- `\bWORD\b`: Matches the exact word "WORD".

## Example

**Pattern:** `\bcat\b`

**Text:** "The cat sat on the mat."

```
import re
text = "The cat sat on the mat."
pattern = r"\bcat\b"
matches = re.findall(pattern, text)
print("Matches found:", matches)
```

## Output

```
Matches found: ['cat']
```

## Explanation

This pattern uses word boundaries to find "cat" as a whole word in the text, ensuring it doesn't match substrings of larger words.

## 5.3 Start and End Anchors

The start `^` and end `$` of string anchors are used to match the beginning and end of the entire text, respectively.

- **^start**: Matches "Start" at the beginning of the text.
- **End\$**: Matches "End" at the end of the text.

## Example

**Pattern:** `^The`

**Text:** "The start of the sentence."

```
import re
text = "The start of the sentence."
pattern = "^The"
matches = re.findall(pattern, text)
```

```
print("Matches found:", matches)
```

## Output

```
Matches found: ['The']
```

## Explanation

The pattern `^The` matches the word "The" only if it appears at the start of the text.

## Example

**Pattern:** `sentence.$`

**Text:** "This is the end of the sentence."

```
import re
text = "This is the end of the sentence."
pattern = "sentence.$"
matches = re.findall(pattern, text)
print("Matches found:", matches)
```

## Output

```
Matches found: ['sentence.']
```

## Explanation

The pattern `sentence.$` matches "sentence." only if it is at the very end of the text.

# 6. Grouping and Capturing

Grouping and capturing allow you to treat multiple characters as a single unit, extract information from matches, and perform operations on captured groups.

## 6.1 Parentheses

Parentheses `()` are used in regular expressions to group parts of the pattern.

This can be useful for applying quantifiers to a sequence of characters or for isolating parts of a pattern for capturing or backreferencing.

## Example

**Pattern:** (ab)+

**Text:** "ababab is a repetitive pattern."

```
import re
text = "ababab is a repetitive pattern."
pattern = r"(ab)+"
matches = re.findall(pattern, text)
print("Matches found:", matches)
```

## Output

```
Matches found: ['ab']
```

## Explanation

This pattern uses parentheses to group ab and apply the + quantifier, matching one or more occurrences of the "ab" sequence.

**IMPORTANT:** The output shows the last captured group.

## Example

**Capturing Groups:** By default, groups created with parentheses not only group parts of the pattern but also capture the matched text for later use, such as extracting data.

**Pattern:** (ab)(cd) **Text:** "abcdef"

```
import re
text = "abcdef"
pattern = r"(ab)(cd)"
match = re.search(pattern, text)
if match:
    print("First group:", match.group(1))
    print("Second group:", match.group(2))
```

## Output

```
First group: ab

Second group: cd
```

## Explanation

This pattern captures two groups: (ab) and (cd). The .group(1) and .group(2) methods are used to access these captured groups.

## Example

**Non-Capturing Groups:** If you want to use parentheses to group parts of your pattern without capturing the matched text, you can use non-capturing groups syntax `(?:pattern)`.

**Pattern:** `(ab)(?:cd)`

**Text:** "abcdef"

```
import re
text = "abcdef"
pattern = r"(ab)(?:cd)"
match = re.search(pattern, text)
if match:
    print("First group:", match.group(1))
    # Second group is not captured
```

## Output

```
First group: ab
```

## Explanation

The pattern `(ab)(?:cd)` captures "ab" as the first group and uses `(?:cd)` to group "cd" without capturing it. This allows for grouping without affecting the numbering of other capturing groups.

In some cases we can use the following flags:

Flag	Modifier	Description	Example Use Case
Case Insensitivity	<code>re.IGNORECASE</code> or <code>re.I</code>	Makes the match case-insensitive, allowing patterns to match letters regardless of case.	Matching "cat", "Cat", "CAT", etc., with the same pattern.
Multiline	<code>re.MULTILINE</code> or <code>re.M</code>	Treats the start (^) and end (\$) characters as working across multiple lines, allowing them to match at the start	Matching patterns at the beginning or end of each line in a text block, rather than only at the very start

Flag	Modifier	Description	Example Use Case
		or end of any line within a string.	or end.
Dot Matches All	<code>re.DOTALL</code> or <code>re.S</code>	Makes the dot (.) special character match all characters, including the newline character, which it does not match by default.	Matching across lines with a single pattern, where the newline character would typically stop the match.

## 7. Practice

I have created a `large_text.txt` file in `Intermediate/assets`.

We will take a look at the couple of objectives, and the rest will be included into homework.

### Objective #1:

Use regular expressions to identify and separate different sections of the text (e.g., Overview, Departments, Courses Offered, Notable Alumni, Contact Information).

### Objective #2:

Find all instances of "University", "Department", and "Course" in the text, regardless of case.

### Objective #3:

List all department names, assuming each is listed at the start of a new line following "Departments:".

### Objective #4:

Extract the university's overview paragraph, including any newlines within it.

## Objective #5:

Capture course names and categorize them as either "Introduction" or "Advanced".

## Objective #6

Identify and extract email addresses and phone numbers

## Objective #7

Highlight all exact occurrences of the words "Regex", "Text", and "Pattern"

## Example

```
"""
I decided to take [Objective #1, Objective #4 and Objective #6]
"""
def read_file_content(filepath):
    """Reads and returns the content of the specified file."""
    with open(filepath, 'r', encoding='utf-8') as file:
        return file.read()

# Objective 1:
def split_sections(text):
    """Splits the text into sections based on headers.

    The pattern of regex matches headers followed by a newline
    """

    pattern = r"^(.*?):\n"
    sections = re.split(pattern, text, flags=re.MULTILINE)
    sections = dict(zip(sections[1::2], sections[2::2]))

    for title, content in sections.items():
        print(f"Section: {title}\nContent: {content}\n\n")

# Objective 4:
def extract_overview(text):
    """Extracts the overview paragraph from the text."""
    pattern = r"University of Regex\n(.*?)(?=\n\n|\nDepartments:)"
    match = re.search(pattern, text, flags=re.DOTALL)
    if match:
        print("Overview:", match.group(1))
    return "Overview not found."

# Objective 6:
def extract_contact_info(text):
    """Extracts email addresses and phone numbers from the text."""
    email_pattern = r"\b[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Z|a-z]{2,}\b"
    phone_pattern = r"\+\d{11}"
```

```

    emails = re.findall(email_pattern, text)
    phones = re.findall(phone_pattern, text)
    print("Emails:", emails)
    print("Phones:", phones)
text = read_file_content('Intermediate/assets/large_text.txt')
split_sections(text)
extract_overview(text)
extract_contact_info(text)

```

## Output

```

Section: Departments
Content: - Computer Science
- Linguistics
- Data Science
Section: Courses Offered
Content: - Introduction to Regular Expressions
- Advanced Pattern Matching
- Text Processing with Python
Section: Notable Alumni
Content: - Jane Doe, Data Scientist at Regex Inc.
- John Smith, Software Engineer at Patterns Co.
Section: Contact Us
Content: Visit our website: www.universityofregex.edu
Email: info@universityofregex.edu
Phone: +447921419244
"""
Overview: Established in 2021, the University of Regex offers comprehensive
courses on regular expressions, text processing,
and pattern matching across various programming languages.
Emails: ['info@universityofregex.edu']
Phones: ['+44792141924']

```

Complete the remaining assignments as the first part of your homework, and think where they could be applicable in already existing projects you've written. Happy Coding!

## 8. Homework

### Task 1: Email Validator

**Objective:** Write a function to validate email addresses using a regular expression.

#### Requirements:

- Your function should verify that the email addresses have the correct format: `username@domain.com`.
- The username part should contain only letters, numbers, dots, hyphens, and underscores.

- The domain should contain only letters and dots.
- Validate a list of email addresses and print out whether each is valid or invalid.

```
import re
def validate_email(email):
    pattern = r''
    if re.match(pattern, email):
        return "Valid email address."
    else:
        return "Invalid email address."
emails = ["example@test.com", "bademail@com", "another.email@test.co.uk"]
for email in emails:
    print(validate_email(email))
```

## Task 2: Extract Dates

**Objective:** Extract all dates from a text in the format dd/mm/yyyy.

### Requirements:

- Write a function that searches a given text for dates matching the specified format.
- Ensure that your regex accounts for the variations in day and month (e.g., 01/01/2022 and 1/1/2022).

```
def extract_dates(text):
    pattern = r''
    dates = re.findall(pattern, text)
    return dates
text = "Important dates include 12/12/2022, 01/01/2023 and 31/12/2022."
print(extract_dates(text))
```

## Task 3: Log File Analysis

**Objective:** Analyze a server log file to extract and count warning and error messages.

### Requirements:

- Assume the log entries for warnings start with "WARN" and errors with "ERROR".
- Your function should return the count of each type of message.

```
log_entries = """
INFO Successful operation.
WARN System instability detected.
ERROR Unable to retrieve data.
WARN Check your network connection.
ERROR Disk full.
```



```
"""
def analyze_log(log):
    # Define regex here, btw it's a good case for using re.findall() ;)
    # Though it might be tricky, I beleive in you
    warn_count = None
    error_count = None
    return {"Warnings": warn_count, "Errors": error_count}
print(analyze_log(log_entries))
```

## Task 4: Password Strength Checker

**Objective:** Create a function to assess the strength of a password.

**Requirements:**

- A strong password must have at least eight characters, include uppercase and lowercase letters, numbers, and at least one special character.
- Use regular expressions to validate the password and return whether it's strong or not.

```
def check_password_strength(password):
    pattern = r'^'          # This is very hard..
    if re.match(pattern, password):
        return "Strong password."
    return "Weak password."
print(check_password_strength("StrongPass1$"))
print(check_password_strength("weakpass"))
```

## Task 5: Phone Number Formatter

**Objective:** Write a function that formats a 10-digit phone number as (xxx) xxx-xxxx.

**Requirements:**

- The input will be a string of 10 digits.
- Your function should format it into the pattern described.

```
def format_phone_number(number):
    pattern = r''
    formatted = re.sub()
    return formatted
print(format_phone_number("1234567890"))
```

## Task 6: Extract Hashtags

**Objective:** Extract all hashtags from a given text.

## Requirements:

- Hashtags start with # and contain alphanumeric characters without spaces.
- Return all found hashtags in a list.

```
def extract_hashtags(text):  
    pattern = r'  
    hashtags = re.findall(pattern, text)  
    return hashtags  
  
text = "Loving the #sunshine and perfect weather in #California!"  
print(extract_hashtags(text))
```

# Lesson 24: Algorithms

"Like the gears in a well-oiled machine, algorithms synchronize data and logic to achieve seamless performance."

## 1. What are Algorithms?

Algorithms in Software Engineering are pre-defined collections of steps and instructions that let you solve complex tasks in an efficient manner.

They can be implemented to accomplish tasks such as: sorting and searching data, performing mathematical computations, processing text, or simulating real-world processes.

The choice of algorithm depends on the problem at hand and factors such as the efficiency, speed, and resource consumption required.

## 2. Algorithmic Complexity (Big O Notation)

Big O notation is used in computer science to describe the performance or complexity of an algorithm. It assumes a worst case scenario, thereby offering classification of an algorithm that doesn't rely on specific circumstances such as hardware or size of input, supplementing it with a variable  $n$ .

Here's a table of basic Big O notations, ordered from least to most complex, where  $n$  is input size:

Big O Notation	Name	Description
$O(1)$	Constant	Execution time remains constant regardless of the input size.
$O(\log n)$	Logarithmic	The execution time grows logarithmically as the input size increases.
$O(n)$	Linear	The execution time grows linearly with the input size.
$O(n^2)$	Polynomial	The execution time grows quadratically with the input size.
$O(2^n)$	Exponential	The execution time doubles with each additional element in the input.

## 2.1 $O(1)$ Constant



With constant complexity name speaks for itself, it means that algorithm execution time will remain constant regardless of the input size ( $n$ ).

An example of  $O(1)$  algorithm would be something as simple as accessing an element of a list with an index.

### Example

```
list1 = [1, 2, 3, 4, 5]
list2 = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
print(list1[3])
print(list2[3])
```

### Output

4

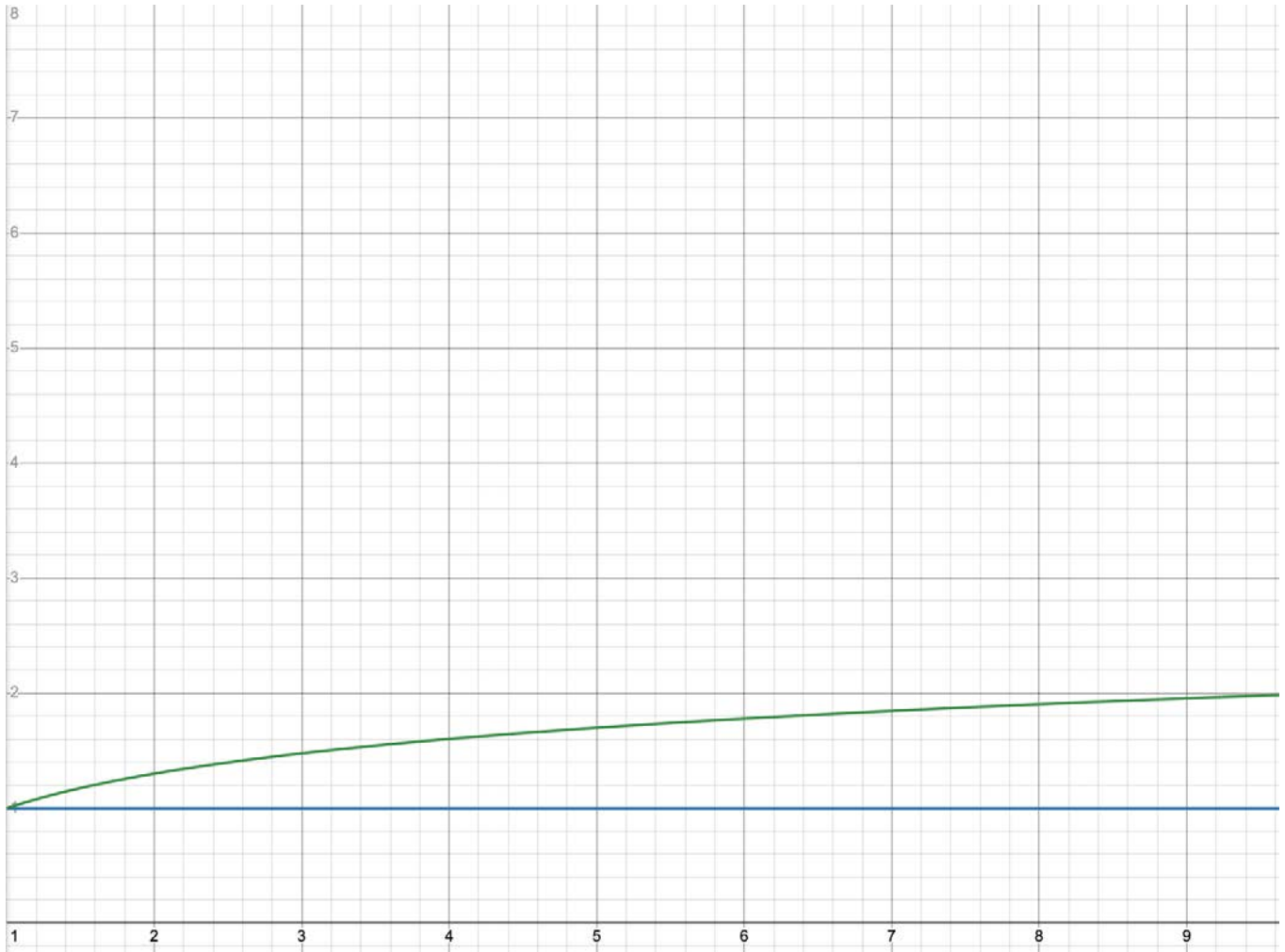
4

## Explanation

Here, accessing an element (4) will take the same amount of time for `list1` and `list2`, regardless of the fact that `list2` is two times larger than `list1`

## 2.2 $O(\log(n))$ Logarithmic

The execution time grows logarithmically as the input size increases.



Logarithmic complexity is a little difficult.

On practice, it means that an algorithm halves a data structure with each execution, an easy-to-understand example of  $O(\log n)$  would be binary search.

## Example

Let's say we have a list of elements and we need to find an element with a particular value, for instance 4 in a sorted list.

```
list1 = [-231, -20, 0, 1, 3, 4, 42, 54]
def binary_search(sorted_list, start, end, target):
    midpoint = (start + end) // 2
    if start >= end:
        return False
    if target > sorted_list[midpoint]:
        return binary_search(sorted_list, midpoint + 1, end, target)
    elif target < sorted_list[midpoint]:
        return binary_search(sorted_list, start, midpoint - 1, target)
    else:
        return sorted_list[midpoint]
if result := binary_search(list1, 0, len(list1), 4):
    print("element found:", result)
else:
    print("element not found")
```

## Output

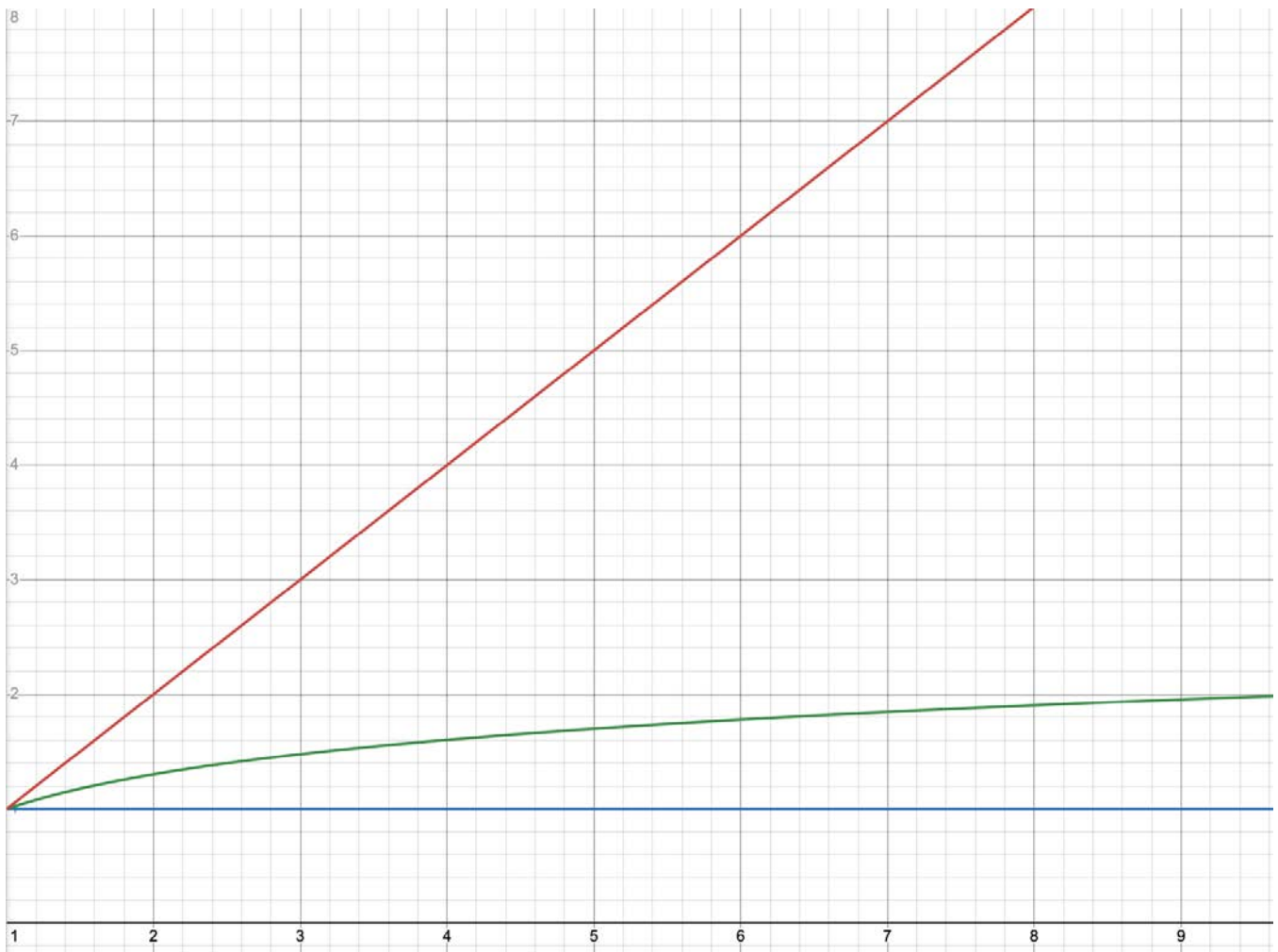
```
element found: 4
```

## Explanation

1. list1, its start and end indexes and target element are passed as arguments to **recursive** binary\_search.
2. if start exceeds end the function returns False indicating there is no such element.
3. if target is greater or less than midpoint element of an element, binary\_search calls itself passing exactly half of the list to itself, as if target is more than middle, target will be in the second **half** of the least, and vice versa.
4. if target equals to midpoint element we return it, as we have found what we were looking for.

## 2.3 O(n) Linear

Linear complexity means that the algorithm visits every element from the list exactly once meaning that execution time grows linearly with the input size.



In regard to python, iteration through a list would be considered to have linear complexity, because we are looking for an each element.

An example of  $O(n)$  algorithm would be script that counts number of occurrences each element in a list.

### Example:

```
list1 = ['a', 'c', 'a', 'b', 'b', 'b', 'a', 'c', 'b']
def counter(list_to_count):
    counts = {}
    for element in list_to_count:
        if element in counts:
            counts[element] += 1
        else:
            counts[element] = 1
    return counts
print(counter(list1))
```

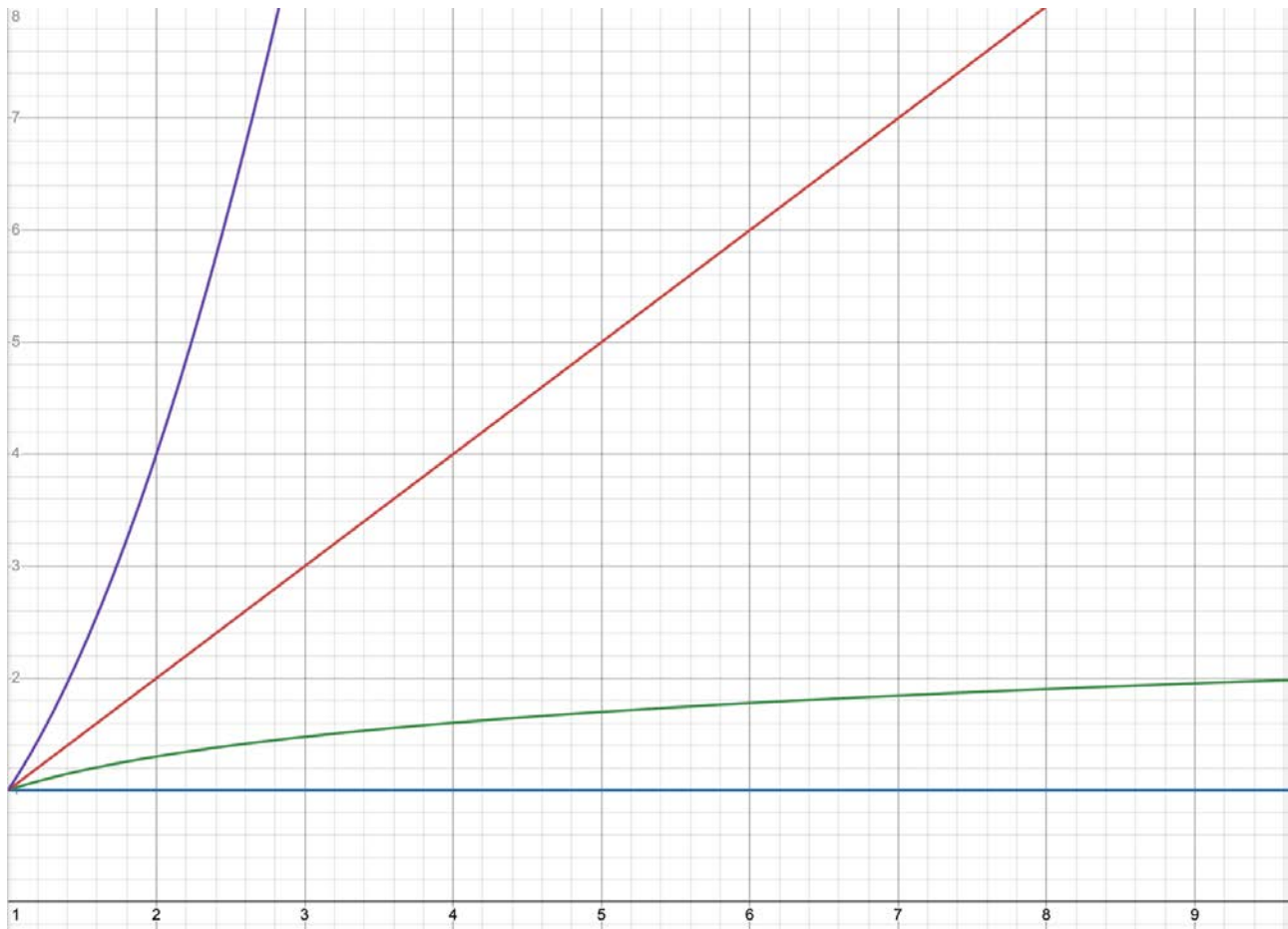
### Output:

```
{'a': 3, 'c': 2, 'b': 4}
```

This algorithm has a time complexity of  $O(n)$ , where  $n$  is the number of elements in the input list, because it iterates through the list once, performing constant-time operations for each element.

## 2.4 $O(n^2)$ Polynomial

The execution time grows quadratically, cubically or even greater-ly with the input size.



See how the others pale in comparison to polynomial time? A polynomial function of the input is polynomial time.

If one loop through a list is  $O(n)$ , one loop with one nested loop must be  $O(n^2)$ . An example of an algorithm that has  $O(n^2)$  complexity is a basic list flattener.

```
list1 = [['a', 'c'], 'a', ['b', 'b', 'b', 'a'], 'c', 'b']
def flattener(list_to_flatten):
    flattened_list = []
    for elem in list_to_flatten:
        if isinstance(elem, list):
            for elem_j in elem:
                flattened_list.append(elem_j)
        else:
            flattened_list.append(elem)
    return flattened_list
print(flattener(list1))
```

**Output**



```
['a', 'c', 'a', 'b', 'b', 'b', 'a', 'c', 'b']
```

## Explanation

Here, we iterate through the elements of `list1` and constantly executing `if` to check if this element is a list itself.

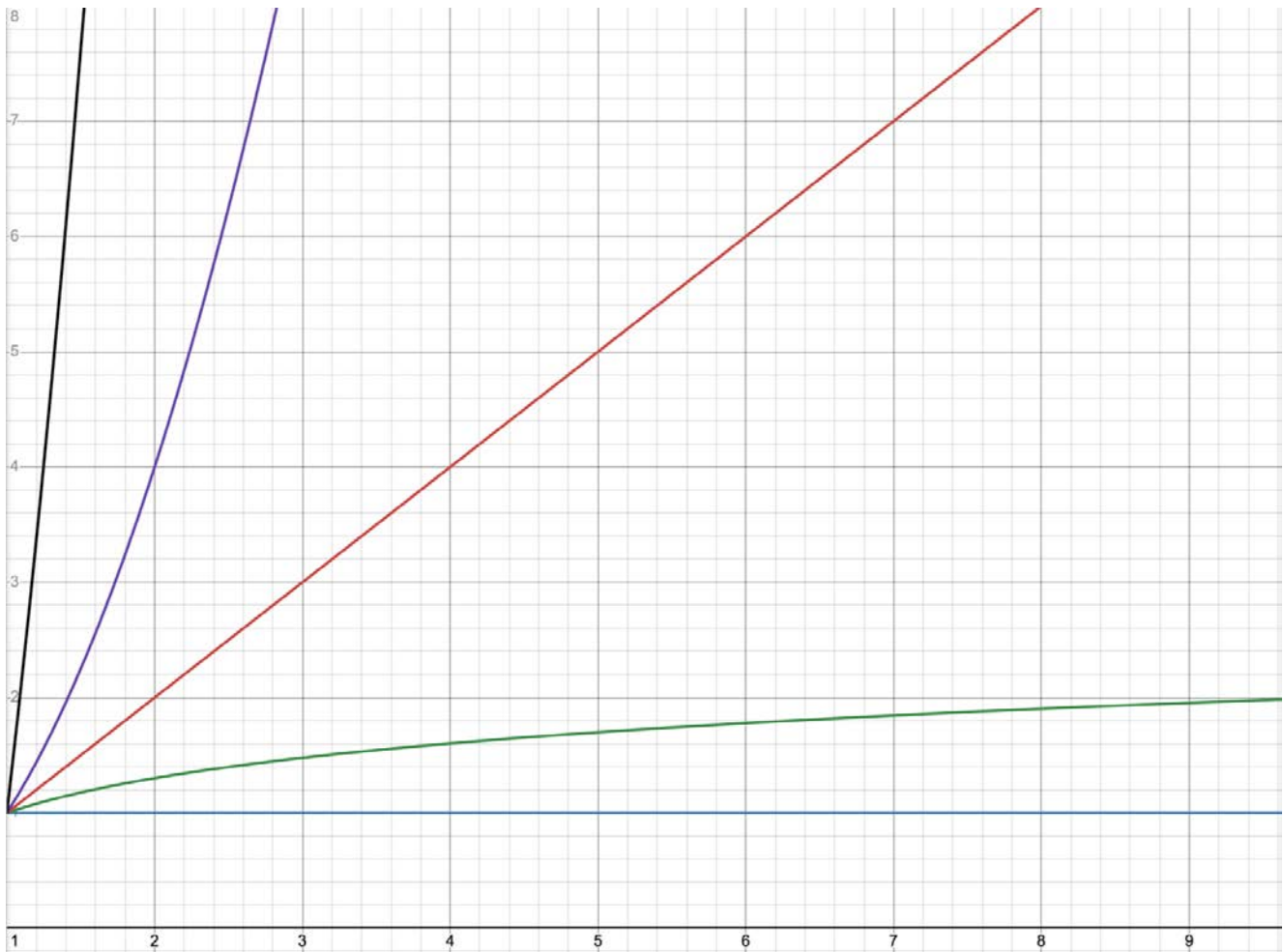
If so, we iterate through this element and record all nested elements into local `flattened_list` variable.

Otherwise, it records element straight away. Obviously, this code has some limitations, as it doesn't account for nested lists in nested lists, but it is still a good example.

**NOTE:** This is a good example as it also showcases the fact that big O notation accounts for the worst key scenario as here we describe algorithms complexity as  $O(n^2)$  assuming that all elements in `list1` are lists themselves, even though in reality this might not be the case.

## 2.5 $O(2^n)$ Exponential

The execution time doubles with each additional element in the input.



An exponential growth is one of the fastest ways in which an algorithm's runtime can increase, and it's typically seen in algorithms that solve problems through recursion, where the solution involves generating all possible cases.

A classic example of an algorithm with  $O(2^n)$  complexity is the recursive calculation of Fibonacci numbers.

The recursive algorithm for computing the  $n$ th Fibonacci number directly implements the definition of exponential complexity.

### Example

```
def fibonacci(n):
    if n <= 1:
        return n
    else:
        return fibonacci(n - 1) + fibonacci(n - 2)

n = 10
print(f"fibonacci({n}) = {fibonacci(n)}")
```

### Output

```
fibonacci(10) = 55
```

### Explanation

Specifically, to calculate `fibonacci(n)`, it calculates `fibonacci(n-1)` and `fibonacci(n-2)` and sums up their results. Each of those calls then makes two more calls, and this process repeats until reaching the base case ( $n \leq 1$ ).

**NOTE:** The inefficiency of the exponential growth in computational requirements illustrates the importance of optimizing recursive algorithms, often through techniques such as memoization or by using iterative approaches.

You may have noticed during previous lessons that execution of `fibonacci` function took too long. Try to simplify the complexity of your applications and aim to make them as fast as possible.

## 3. Sorting Algorithms

Sorting algorithms organize the elements of a list or array into a certain sequence, for example in ascending or descending order. These algorithms aid in data organization allowing for optimized access and processing.

### 3.1 Bubble Sort

Bubble Sort is a straightforward comparison-based algorithm where each pair of adjacent elements is compared, and the elements are swapped if they are not in order.

### Example:

```
def bubble_sort(arr):
    n = len(arr)
    for i in range(n - 1):
        for j in range(0, n - i - 1):
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
    return arr
arr = [64, 34, 25, 12, 22, 11, 90]
sorted_arr = bubble_sort(arr)
print("Sorted array:", sorted_arr)
```

### Output:

```
Sorted array: [11, 12, 22, 25, 34, 64, 90]
```

### Explanation:

This example iterates over the list, comparing each element with the next one and swapping them if they're in the wrong order. This process repeats until no swaps are needed, indicating that the list is sorted. It's a simple but inefficient algorithm, mainly used for educational purposes.

**Complexity:**  $O(n^2)$  in the worst case, as it needs to make a series of passes through the list for each element.

## 3.2 Merge Sort

Merge Sort is a Divide and Conquer algorithm. It divides the input array in half, recursively sorts each half, and then merges the sorted halves back together.

### Example:

```
def merge_sort(arr):
    if len(arr) > 1:
        mid = len(arr) // 2
        L = arr[:mid]
        R = arr[mid:]
        merge_sort(L)
        merge_sort(R)
        i = j = k = 0
        while i < len(L) and j < len(R):
            if L[i] < R[j]:
                arr[k] = L[i]
                i += 1
            else:
                arr[k] = R[j]
                j += 1
            k += 1
        while i < len(L):
            arr[k] = L[i]
```

```

        i += 1
        k += 1
    while j < len(R):
        arr[k] = R[j]
        j += 1
        k += 1
    return arr
arr = [38, 27, 43, 3, 9, 82, 10]
sorted_arr = merge_sort(arr)
print("Sorted array:", sorted_arr)

```

### Output:

```
Sorted array: [3, 9, 10, 27, 38, 43, 82]
```

### Explanation:

Merge Sort recursively splits the list into halves until it has lists of single elements and then merges those atomic elements in sorted order to produce sorted sublists, which are then merged into a final sorted list. It's much more efficient than Bubble Sort, especially for large datasets.

**Complexity:**  $O(n \log n)$  in all cases because it divides the list into two halves and takes linear time to merge two halves.

## 4. Searching Algorithms

Searching algorithms are designed for finding an item or group of items with specific properties within a collection of items.

### 4.1 Linear Search

Linear Search is a simple method where the target value is searched sequentially in the list until a match is found or the end of the list is reached.

### Example:

```

def linear_search(arr, x):
    for i in range(len(arr)):
        if arr[i] == x:
            return i
    return -1
arr = [5, 8, 1, 3, 7]
x = 3
result = linear_search(arr, x)
print("Element found at index:", result)

```

### Output:

```
Element found at index: 3
```

### Explanation:

The function iterates through each item in the list until it finds the target value, 3, and returns its index, 3. If the element is not found, it returns -1.

**Complexity:**  $O(n)$ , as it may have to check each element at worst.

## 4.2 Binary Search

Binary Search is an efficient algorithm for finding an item from a sorted list of items. It works by repeatedly dividing in half the portion of the list that could contain the item, until you've narrowed down the possible locations to just one.

### Example:

```
list1 = [-231, -20, 0, 1, 3, 4, 42, 54]
def binary_search(sorted_list, start, end, target):
    midpoint = (start + end) // 2
    if start >= end:
        return False
    if target > sorted_list[midpoint]:
        return binary_search(sorted_list, midpoint + 1, end, target)
    elif target < sorted_list[midpoint]:
        return binary_search(sorted_list, start, midpoint - 1, target)
    else:
        return sorted_list[midpoint]
if result := binary_search(list1, 0, len(list1), 4):
    print("element found:", result)
else:
    print("element not found")
```

### Output:

```
Element found at index: 3
```

### Explanation:

The function compares the target value (10) with the middle element of the array. If the target value is less than the middle element, it repeats the process on the left subarray. If the target value is greater, it repeats on the right subarray. This process continues until the target value is found or the subarray becomes empty.

**Complexity:**  $O(\log n)$ , as it splits the search area by half with each step.

## 5. Dynamic Algorithms

Dynamic programming is a strategy for solving problems by breaking them down into simpler subproblems, solving each subproblem just once, and storing their solutions.

### 5.1 0/1 Knapsack Problem

The 0/1 Knapsack problem seeks to maximize the total value of items that can be carried in a knapsack, considering the weight capacity of the knapsack.

**Definition of the problem:** Given  $N$  items where each item has some **weight** and **value** and also given a bag with capacity  $W$ , [i.e., the bag can hold at most  $W$  weight in it]. The task is to put the items into the bag such that the sum of values associated with them is the maximum possible.

**Example:**

```
def knapsack(values, weights, capacity):
    n = len(values)
    dp = [[0 for x in range(capacity + 1)] for x in range(n + 1)]
    for i in range(1, n + 1):
        for w in range(1, capacity + 1):
            if weights[i - 1] <= w:
                dp[i][w] = max(dp[i - 1][w], dp[i - 1][w - weights[i - 1]] +
                                values[i - 1])
            else:
                dp[i][w] = dp[i - 1][w]
    return dp[n][capacity]
values = [60, 100, 120]
weights = [10, 20, 30]
capacity = 50
print(f" Maximum value in knapsack = {knapsack(values, weights, capacity)}")
```

**Output:**

```
Maximum value in knapsack = 220
```

**Explanation:**

This solution uses dynamic programming to build up a table `dp` where each entry `dp[i][w]` represents the maximum value that can be achieved with the first  $i$  items and a weight limit of  $w$ . The solution to the problem is found in `dp[n][capacity]`.

**Complexity:**  $O(nW)$ , where  $n$  is the number of items and  $w$  is the capacity of the knapsack. This is because it iterates through each item and for each item through all weights up to  $w$ .

I am not a big fan of algorithms in general, but have a huge respect to them. I would suggest you to understand how they work and you will be able to analyse your applications to find areas for improvements.

Good luck and let's move on to Advanced Section! Happy Algorithming!

# Lesson 25: Advanced OOP

"The place where code transcends functionality and becomes an art form"

## 1. Mixins

Mixins are a sort of class designed to offer optional methods or functionality to other classes.

They are a form of multiple inheritance, allowing developers to add the same functionality to multiple classes without repeating code.

**IMPORTANT:** Unlike traditional base classes, mixins are specifically designed to be combined with other classes, not to stand on their own.

### 1.1 The Purpose of Mixins

They are used to modularize functionality, making it easy to add or remove features from objects without affecting the inheritance hierarchy of the classes. They can:

- Provide a set of methods that can be used in different classes.
- Compose behaviors in classes.
- Add functionality to classes without modifying them directly.

Generally, the main idea is to make mixins generic as possible, defining functionality to use them in different classes which serve different purpose.

### 1.2 Implementation

A mixin is typically implemented as a class that does not work by itself. As I mentioned before, it must be combined with another class to make sense.

#### Example

```
class JsonMixin:
    import json
    def to_json(self):
        return self.json.dumps(self.__dict__)
class Vehicle:
    def __init__(self, brand, model):
        self.brand = brand
        self.model = model
class Car(JsonMixin, Vehicle): # Mixin
    def __init__(self, brand, model, engine_type):
        super().__init__(brand, model)
        self.engine_type = engine_type
my_car = Car("Tesla", "Model S", "Electric")
print(my_car.to_json()) # Call external functionality from ``JsonMixin`` class
```



## Output

```
{"brand": "Tesla", "model": "Model S", "engine_type": "Electric"}
```

## Explanation

In this example, `JsonMixin` provides a `to_json` method, which is then available to the `Car` class through multiple inheritance. This is a simple yet powerful way to add functionality to classes without affecting their logic.

Let's create another mixin that adds logging functionality to any class.

## Example

```
import logging
class LoggingMixin:
    @property
    def logger(self):
        name = '.'.join([self.__class__.__module__, self.__class__.__name__])
        return logging.getLogger(name)
    def log_info(self, message):
        self.logger.info(message)
    def log_error(self, message):
        self.logger.error(message)
logging.basicConfig(level=logging.INFO)
class DataProcessor(LoggingMixin):
    def process_data(self, data):
        self.log_info(f"Processing data: {data}")
        if data == "error":
            self.log_error("An error occurred while processing data.")
processor = DataProcessor()
processor.process_data("some data")
processor.process_data("error")
```

## Output

```
INFO:__main__.DataProcessor:Processing data: some data

INFO:__main__.DataProcessor:Processing data: error

ERROR:__main__.DataProcessor:An error occurred while processing data.
```

## Explanation

`LoggingMixin` uses the `@property` decorator for `logger` to ensure that the logger is specific to the class it's mixed into, using the class's fully qualified name.

This makes the logs clearer and more informative because they're automatically tagged with the class name. As well it remains generic as possible and can be injected into any class being an extremely powerful tool.

## 1.3 Best Practices

**1.Single Responsibility:** Each mixin should be focused on a single, clear purpose, adhering to the Single Responsibility Principle.

```
import json
class JsonMixin:
    def to_json(self):
        """Serialize data to a JSON format."""
        return json.dumps(self.__dict__)
class Person(JsonMixin):
    def __init__(self, name, age):
        self.name = name
        self.age = age
# Usage
person = Person("John Doe", 30)
print(person.to_json()) # {"name": "John Doe", "age": 30}
```

**IMPORTANT:** `JsonMixin` can be integrated in ANY class, I am intentionally including the same example as above to show that we can use it not only with a class `Car` but with a `Person` as well.

### Output

```
{"name": "John Doe", "age": 30}
```

**2.Avoid State in Mixins:** Ideally, mixins should not store state. If they must, be cautious of conflicts with the classes they are mixed into.

```
class DebuggingMixin:
    def debug_method_call(self, method_name):
        print(f"Method called: {method_name}")
class Calculation(DebuggingMixin):
    def add(self, a, b):
        self.debug_method_call('add')
        return a + b
    def subtract(self, a, b):
        self.debug_method_call('subtract')
        return a - b
# Usage
calc = Calculation()
print(calc.add(10, 5))
print(calc.subtract(10, 5))
```

## Output

```
Method called: add
15
Method called: subtract
5
```

**3. Use Descriptive Names:** Since mixins can be combined in various ways, their names should be as descriptive as possible to clarify their purpose and functionality, as everything in Python.

```
import base64
class EncryptionDecryptionMixin:
    def encrypt_data(self, data):
        """Encrypts data using base64 encoding."""
        return base64.b64encode(data.encode('utf-8'))
    def decrypt_data(self, data):
        """Decrypts data using base64 decoding."""
        return base64.b64decode(data).decode('utf-8')
class SecureCommunicator(EncryptionDecryptionMixin):
    def send_secure_message(self, message):
        encrypted_message = self.encrypt_data(message)
        print(f"Sending encrypted message: {encrypted_message}")
    def receive_secure_message(self, encrypted_message):
        message = self.decrypt_data(encrypted_message)
        print(f"Received message: {message}")
comm = SecureCommunicator()
encrypted = comm.encrypt_data("Hello World")
print(f"Encrypted: {encrypted}")
print("Decrypted:", comm.decrypt_data(encrypted))
```

## Output

```
Encrypted: b'SGVsbG8gV29ybGQ='
Decrypted: Hello World
```

**4. Be Mindful of the Method Resolution Order (MRO):** Python's method resolution order means that the order of base classes affects the methods used.

```
class BaseFeature:
    def feature(self):
        print("Base feature")
class EnhancementMixin:
    def feature(self):
        print("Enhanced feature")
class AdditionalFeatureMixin:
    def feature(self):
```

```

        super().feature()
        print("Additional feature")
class Product(EnhancementMixin, AdditionalFeatureMixin, BaseFeature):
    pass
product = Product()
product.feature() # Will call EnhancementMixin's feature due to MRO

```

Again, based on my experience mixins are very underestimated, but hopefully you will be able to find where they could be applied potentially.

## 2. Metaclasses

Metaclasses define how a class behaves, they define the rules for class objects. A class is an instance of a metaclass, just as an object is an instance of a class.

### 2.1 Syntax

In Python, the type `type` is the built-in metaclass that is used by default, but custom metaclasses can be created by inheriting from `type`. In order to create a metaclass you need to be inheriting from `type` and defining the `__new__` or `__init__` method.

#### Example

```

class Meta(type):
    def __new__(cls, name, bases, dct):
        # custom logic here
        return super().__new__(cls, name, bases, dct)

```

To use a metaclass, you specify it in the class definition using the `metaclass` keyword.

```

class MyClass(metaclass=Meta):
    pass

```

The `__new__` method in Python is a special method used for creating and returning a new instance of a class.

It is important to understand, unlike `__init__`, which initializes an already existing instance, `__new__` is responsible for actually creating the instance.

#### Example

```

class MyClass:
    def __new__(cls):
        print("Creating instance")
        instance = super().__new__(cls)
        return instance
    def __init__(self):
        print("Initializing instance")
# When you create a new instance of MyClass

```

```
obj = MyClass()
```

## Output

```
# Creating instance

# Initializing instance
```

In some cases, you might override `__new__` when you need to control the creation of a new instance, such as enforcing a Singleton Pattern (ensuring a class only ever has one instance) or modifying the instantiation process of a subclass.

## 2.2 Real world examples

Suppose you want to add a debugging method to a range of classes to print their attributes in a formatted manner for easier debugging.

Instead of adding the method to each class manually, you can create a metaclass that automatically injects this method into any class that uses it, that's how we ensure that we have a controll on the stage of creation of a new instance.

### 2.2.1 Debugging Metaclass

```
class DebugMeta(type):
    def __new__(cls, name, bases, dct):
        # Define a new method
        def debug(self):
            attrs = "\n".join(f"{key}={value}" for key, value in
self.__dict__.items())
            print(f"Debugging {self.__class__.__name__}: \n{attrs}")

        # Add the method to the class
        dct["debug"] = debug

        return super().__new__(cls, name, bases, dct)
class Product(metaclass=DebugMeta):
    def __init__(self, name, price):
        self.name = name
        self.price = price
class Customer(metaclass=DebugMeta):
    def __init__(self, id, name):
        self.id = id
        self.name = name
# While creating instances
# (before ``__init__`` ``__new__`` will be called and assign attributes of
classes which use ``DebugMeta``
product = Product("Laptop", 1200)
customer = Customer("001", "John Doe")
```

```
# Using the new method
product.debug()
customer.debug()
```

## Output

```
Debugging Product:
name=Laptop
price=1200
Debugging Customer:
id=001
name=John Doe
```

## Explanation

- The `DebugMeta` metaclass defines a `debug` method inside the `__new__` method and adds it to every class that specifies `DebugMeta` as its metaclass.
- By using this metaclass, any class can automatically gain the `debug` method without explicitly defining it. This is powerful for adding functionality like logging, debugging, serialization, and more across multiple classes.

Now, developers have flexibility in extending class functionality and this can be particularly useful in large applications or libraries.

### 2.2.2. Django's ORM (Object-Relational Mapping)

Django, Python web framework, uses metaclasses to define models that represent database tables. The `ModelBase` metaclass in Django's ORM system allows developers to define models using simple class syntax, which the metaclass then translates into database fields and tables.

This abstraction enables developers to work with databases in a more Pythonic way, without having to write SQL queries for basic operations.

#### Example Use Case:

```
from django.db import models
class Person(models.Model):
    name = models.CharField(max_length=100)
    age = models.IntegerField()
    class Meta:
        db_table = 'person_table'
```

## Explanataion

`models.Model` uses a metaclass to process the class attributes (name and age) and convert them into database columns.

The metaclass also handles inheritance, database schema generation, and integrates with Django's migrations system to apply changes to the database schema over time.

Unfortunately, Django framework is beyond the scope of this book, as it is enormously big framework which will require a separate book to be written in order to describe its functionality, but you can refer to the [official documentation](#) and try building web apps by yourself, as your knowledge should be enough already.

### 2.2.3 SQLAlchemy's Declarative Base

SQLAlchemy, a popular SQL toolkit and Object-Relational Mapping (ORM) library for Python, utilizes metaclasses to define a declarative base class.

#### Example

```
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy import Column, Integer, String
Base = declarative_base()
class User(Base):
    __tablename__ = 'users'
    id = Column(Integer, primary_key=True)
    name = Column(String)
    age = Column(Integer)
```

#### Explanation

The `declarative_base()` function in SQLAlchemy uses a metaclass to create a base class, which then automatically maps class properties to database table columns. Due to metaclasses the ORM simplifies the creation of models and their associated database operations.

**IMPORTANT:** Be careful using metaclasses, they can introduce complexity and should only be used when simpler solutions like class inheritance or decorators are insufficient.

Metaclasses can automatically validate or modify member attributes. This can be useful for type checking or automatically adding getter/setter methods to attributes.

## 3. Type checking

You can annotate variables, function parameters, and return types using custom classes just as you would with built-in types. This tells the reader of the code, as well as static type checkers, exactly what kind of object is expected.

#### Example:

```
class Car:
    def __init__(self, make: str, model: str):
        self.make = make
        self.model = model
def display_car_info(car: Car) -> str:
    return f"{car.make} {car.model}"
# Correct usage
```

```
my_car: Car = Car("Tesla", "Model S")
info: str = display_car_info(my_car)
wrong_car: int = 123
display_car_info(wrong_car)  # Mypy will flag this as an error
```

## 3.1 Conditional Imports

When working with type hints that refer to classes defined in external modules, you might encounter a situation where you want to avoid importing those modules directly at runtime.

Python's `typing.TYPE_CHECKING` constant can be used in these cases to conditionally import modules for type annotations without affecting runtime performance.

### Example

```
from typing import TYPE_CHECKING
if TYPE_CHECKING:
    from some_external_module import ExternalClass
def func(arg: 'ExternalClass') -> None:
    pass
```

Note the use of a string literal `'ExternalClass'` in the function signature. This is a forward declaration, which is necessary because the actual `ExternalClass` is not imported at runtime.

Having this practices can reduce circular imports, which is the worst error to be tackled, as well, improve readability and the code will be less prone to errors.

## 3.2 Runtime Type Checking

Python's dynamic nature allows for flexibility in handling different types, but there are scenarios where enforcing type safety at runtime is necessary, especially when interfacing with external systems or libraries.

### 3.2.1 `isinstance()`

The `isinstance()` function checks if an object is an instance of a specific class or a tuple of classes. It's a straightforward way to validate types at runtime, ensuring that the data conforms to the expected type before proceeding.

### Example

```
def process_data(data):
    if not isinstance(data, dict):
        raise ValueError("Expected a dictionary")
    # process data here
```



In this case we want to work only with a `dict` class and its subclasses, inheritance is taken into consideration as well.

## Example

Suppose we have a class hierarchy where `Animal` is a base class, and `Dog` and `Cat` are subclasses. We might want to write a function that behaves differently based on whether it's given a `Dog` or a `Cat`.

```
class Animal:
    pass
class Dog(Animal):
    def bark(self):
        return "Woof!"
class Cat(Animal):
    def meow(self):
        return "Meow!"
def make_sound(animal):
    if isinstance(animal, Dog):
        print(animal.bark())
    elif isinstance(animal, Cat):
        print(animal.meow())
    else:
        raise TypeError("make_sound only accepts Dog or Cat instances")
# Usage
make_sound(Dog())
make_sound(Cat())
```

## Output

Woof!

Meow!

In this way, we handle different scenarios based on the object type, calling `isinstance()` function.

### 3.2.2 `type()`

While `isinstance()` checks an object's type against a class or a tuple of classes considering inheritance, `type()` is used to get the exact type of an object without considering subclassing. This can be useful for type checking that needs to ignore the inheritance hierarchy.

## Example

```
class MyBaseClass:
    pass
class MyDerivedClass(MyBaseClass):
```

```

    pass
obj = MyDerivedClass()
if type(obj) is MyDerivedClass:
    print("obj is exactly MyDerivedClass")
else:
    print("obj is not exactly MyDerivedClass")

```

### Output

```
obj is exactly MyDerivedClass
```

Or we can implement something similar to the example avoiding subclass interference.

```

if type(obj) is MyBaseClass:
    print("obj is exactly MyBaseClass and not a subclass")
else:
    print("obj is a subclass of MyBaseClass or an unrelated class")

```

### Output

```
obj is a subclass of MyBaseClass or an unrelated class
```

Based on the table we could see when and which function should be used.

Personally, I don't like using `type()` in production code, as it can be dangerous if we want to work within inheritance paradigm and can lead to unexpected bugs or changing behaviors. But again, you will not find out what is best for your application unless trying.

### 3.2.3 `isinstance()` VS `type()`

Feature	<code>isinstance()</code>	<code>type()</code>
Checks inheritance	Yes, considers an object an instance if it's derived from the class.	No, checks for the object's immediate type only.
Use case	Ideal for polymorphic behavior where subclass instances should be treated as instances of the base class.	When you need to distinguish an object's exact type, especially to differentiate between a class and its subclass.
Syntax	<code>isinstance(object, class_or_tuple)</code>	<code>type(object) is SomeClass</code>

### 3.2.4 Type Guards

Type guards are constructs that explicitly check and narrow down the type of variables within a certain scope, making it safer to perform operations that are type-specific. You would want to use them when dealing with union types or the `Any` type, where the specific type might not be clear

### Example

```
from typing import Union
def double(value: Union[int, str]):
    if isinstance(value, str):
        # The type of value is narrowed down to str here
        return value * 2
    elif isinstance(value, int):
        # The type of value is narrowed down to int here
        return value + value
print(double(11))
print(double('11'))
```

### Output

```
22

1111
```

### Explanation

In this code, `value` can be either an `int` or a `str`. The `isinstance()` checks act as type guards, narrowing down the type of `value` within each block and allowing for type-specific operations. That can be useful in case we want to handle different operations with different approaches.

## 4. Duck Typing

The main concept of duck typing is that, a function can accept any object that has the required attributes or methods, regardless of the object's class.

"If it looks like a duck, swims like a duck, and quacks like a duck, then it probably is a duck"

### Example

```
def quack_and_fly(thing):
    thing.quack()
    thing.fly()
    print("This thing looks like a duck and quacks like a duck.")
class Duck:
    def quack(self):
```

```

        print("Quack, quack!")
    def fly(self):
        print("Flap, flap!")
class Airplane:
    def fly(self):
        print("Whoosh!")
class Person:
    def quack(self):
        print("I'm quacking like a duck!")
    def fly(self):
        print("I'm flapping my arms!")
duck = Duck()
airplane = Airplane()
person = Person()
quack_and_fly(duck)      # This works fine
quack_and_fly(person)   # This works fine
# This will raise an AttributeError, because `airplane` has no `quack` method.
quack_and_fly(airplane)

```

## Output

```

Quack, quack!
Flap, flap!
This thing looks like a duck and quacks like a duck.
I'm quacking like a duck!
I'm flapping my arms!
This thing looks like a duck and quacks like a duck.
Traceback (most recent call last):
  File "<string>", line 32, in <module>
    File "<string>", line 2, in quack_and_fly
AttributeError: 'Airplane' object has no attribute 'quack'

```

## Explanation

- Despite their different types, all these objects can be used in the `quack_and_fly()` function
- In case with `airplane`, we can use `Type Guards` to check the types passed or call `hasattr()` function to check if an object has a certain attribute or method before calling it.
- Due to the fact that `Python` is a dynamical language, we can't really strictly enforce `Python` to define the type explicitly.

Duck typing is more than just a theoretical concept. It has practical applications in real-world scenarios, especially in web development.

## Example

Consider a web application framework that needs to handle different types of `HTTP` requests. Instead of checking the type of request, you can rely on the presence of a method to handle it.

```

class GetRequestHandler:
    def handle(self, request):

```

```

        # process the GET request
        return "Handling GET request"
class PostRequestHandler:
    def handle(self, request):
        # process the POST request
        return "Handling POST request"
def process_request(handler, request):
    return handler.handle(request)
# Both GetRequestHandler and PostRequestHandler can be passed to
process_request
get_handler = GetRequestHandler()
post_handler = PostRequestHandler()
print(process_request(get_handler, 'test'))    # As long as it has a handle
method, it works
print(process_request(post_handler, 'test'))

```

## Output

```

Handling GET request

Handling POST request

```

## Example

In data analysis, you might encounter different data sources. Here's how duck typing allows you to write generic data loading functions:

```

class CSVDataLoader:
    def load_data(self, source):
        print(f>Loading data from a CSV file: {source}")
class ExcelDataLoader:
    def load_data(self, source):
        print(f>Loading data from an Excel file: {source}")
class SQLDataLoader:
    def load_data(self, source):
        print(f>Loading data from a SQL database: {source}")
def load_data_from_any_source(loader, source):
    loader.load_data(source)
csv_loader = CSVDataLoader()
excel_loader = ExcelDataLoader()
sql_loader = SQLDataLoader()
load_data_from_any_source(csv_loader, "data.csv")
load_data_from_any_source(excel_loader, "data.xlsx")
load_data_from_any_source(sql_loader, "database.sqlldb")

```

## Output

```

Loading data from a CSV file: data.csv
Loading data from an Excel file: data.xlsx

```

**IMPORTANT:** Duck typing requires careful handling, though it has a high flexibility, it can lead to potential pitfalls inside the app and situations where it's a disadvantage, rather than an advantage.

During designing your application, don't hesitate to look at this table, it can help you to decide the best approach to be used:

## 4.1 Duck Typing

OOP Concept	Advantages	Potential Drawbacks	Use-Case Scenarios
Duck Typing	<ul style="list-style-type: none"> <li>- Flexibility</li> <li>- Less boilerplate</li> <li>- Natural polymorphism</li> </ul>	<ul style="list-style-type: none"> <li>- Possible runtime errors</li> <li>- Less explicit type safety</li> </ul>	<ul style="list-style-type: none"> <li>- Small scripts</li> <li>- When behavior is a priority over type</li> </ul>

```
class Duck:
    def quack(self):
        print("Quack")
def make_sound(animal):
    animal.quack()
daffy = Duck()
make_sound(daffy)
```

## 4.2 Explicit Type Checking

OOP Concept	Advantages	Potential Drawbacks	Use-Case Scenarios
Explicit Type Checking	<ul style="list-style-type: none"> <li>- Clear type contracts</li> <li>- Compile-time error detection</li> </ul>	<ul style="list-style-type: none"> <li>- More boilerplate</li> <li>- Less flexibility</li> </ul>	<ul style="list-style-type: none"> <li>- Large systems</li> <li>- Safety-critical applications</li> </ul>

```
def process_data(data):
    if not isinstance(data, dict):
        raise ValueError("Expected a dictionary")
    # process data here
```

## 4.3 Abstract Base Classes (ABC)

OOP Concept	Advantages	Potential Drawbacks	Use-Case Scenarios
Abstract Base Classes	<ul style="list-style-type: none"> <li>- Enforces an interface</li> <li>- Explicit contracts between code parts</li> </ul>	<ul style="list-style-type: none"> <li>- Requires more upfront design</li> <li>- Can be overkill for simple cases</li> </ul>	<ul style="list-style-type: none"> <li>- Plugin systems</li> <li>- Framework development</li> </ul>

```

from abc import ABC, abstractmethod
class AbstractAnimal(ABC):
    @abstractmethod
    def sound(self):
        pass
class Dog(AbstractAnimal):
    def sound(self):
        print("Woof!")
fido = Dog()
fido.sound()

```

## 4.4 Static Typing (Type Hints, `mypy`)

OOP Concept	Advantages	Potential Drawbacks	Use-Case Scenarios
Static Typing	<ul style="list-style-type: none"> <li>- Early error detection</li> <li>- Improved IDE support and code completion</li> </ul>	<ul style="list-style-type: none"> <li>- Additional complexity in annotations</li> <li>- Steeper learning curve for new Python users</li> </ul>	<ul style="list-style-type: none"> <li>- Large codebases</li> <li>- Applications with a long lifecycle</li> </ul>

```

def greet(name: str) -> str:
    return f'Hello, {name}'
greeting = greet("Alice")
print(greeting)

```

Ultimately, the choice of when and how to use these concepts depends on the specific requirements of your project, your team's preferences, and the need to balance development speed with code safety and maintainability.

Personally, I prefer more explicit type checking, static typing for large projects, and Abstract Base Classes. Not a big fan of duck typing, as all the time you have a feeling that something will break :)

Enough theory for today, try it yourself!

## 5. Quiz

### Question 1:

What is the primary purpose of using mixins in object-oriented programming?

- A) To serve as the primary base class for all classes in an application.
- B) To provide a set of methods that can be used in various unrelated classes to enhance functionality without using inheritance.
- C) To enforce strict data typing in Python.
- D) To replace the functionality of interfaces in Python.

### Question 2:

In the context of duck typing, which of the following statements is true?

- A) Duck typing refers to the type system used by Python to manage memory more efficiently.
- B) Duck typing allows a function to accept any object that meets the required interface, regardless of its specific type.
- C) Duck typing is a programming style where the type of the variable is known only at runtime.
- D) Duck typing requires classes to inherit from a common base to be interchangeable in a function.

### Question 3:

What does this snippet illustrate?

```
class EncryptionMixin:
    def encrypt(self, message):
        # Simulated encryption logic
        return message[::-1]
class DecryptionMixin:
    def decrypt(self, message):
        # Simulated decryption logic
        return message[::-1]
class SecureCommunicator(EncryptionMixin, DecryptionMixin):
    pass
comm = SecureCommunicator()
encrypted = comm.encrypt("hello")
print(comm.decrypt(encrypted))
```

- A) Singleton pattern
- B) Factory method pattern
- C) Use of mixins for adding encryption and decryption methods to a class
- D) The use of the decorator pattern to enhance method functionality

### Question 4:

What is a metaclass in Python primarily used for?

- A) To prevent the modification of a class once it has been created.
- B) To define a class for an ORM framework like SQLAlchemy or Django.



- C) To create classes and control the behavior of class creation.
- D) To implement mandatory methods in derived classes like an abstract base class.

**Question 5:**

Which Python feature allows the creation of a class whose instances can only have one instantiation no matter how many times the class is instantiated?

- A) Encapsulation
- B) Inheritance
- C) Metaclass with Singleton Pattern
- D) Duck typing with type hints

## **6. Homework**

I want you to be creative and submit H/W tasks for review in the related MR. Find your own usage of patterns and techniques described above!

# Lesson 26: Context Managers

"The gatekeepers of scoped behavior"

## 1. Introduction

Context managers are a feature in Python that provides a convenient way to manage resources.

In Python are implemented using the `with` statement, providing a way to allocate and release resources precisely when you need them.

The most common use case we already encountered is file handling, ensuring that a file is closed once operations on it are completed, regardless of whether an error occurs.

### 1.1 The `with` Statement

The `with` statement simplifies exception handling by encapsulating common preparation and cleanup tasks in so-called context managers. It ensures that resources are properly managed.

#### Example

```
with open('example.txt', 'r') as file:
    content = file.read()
    print(content)
```

#### Explanation

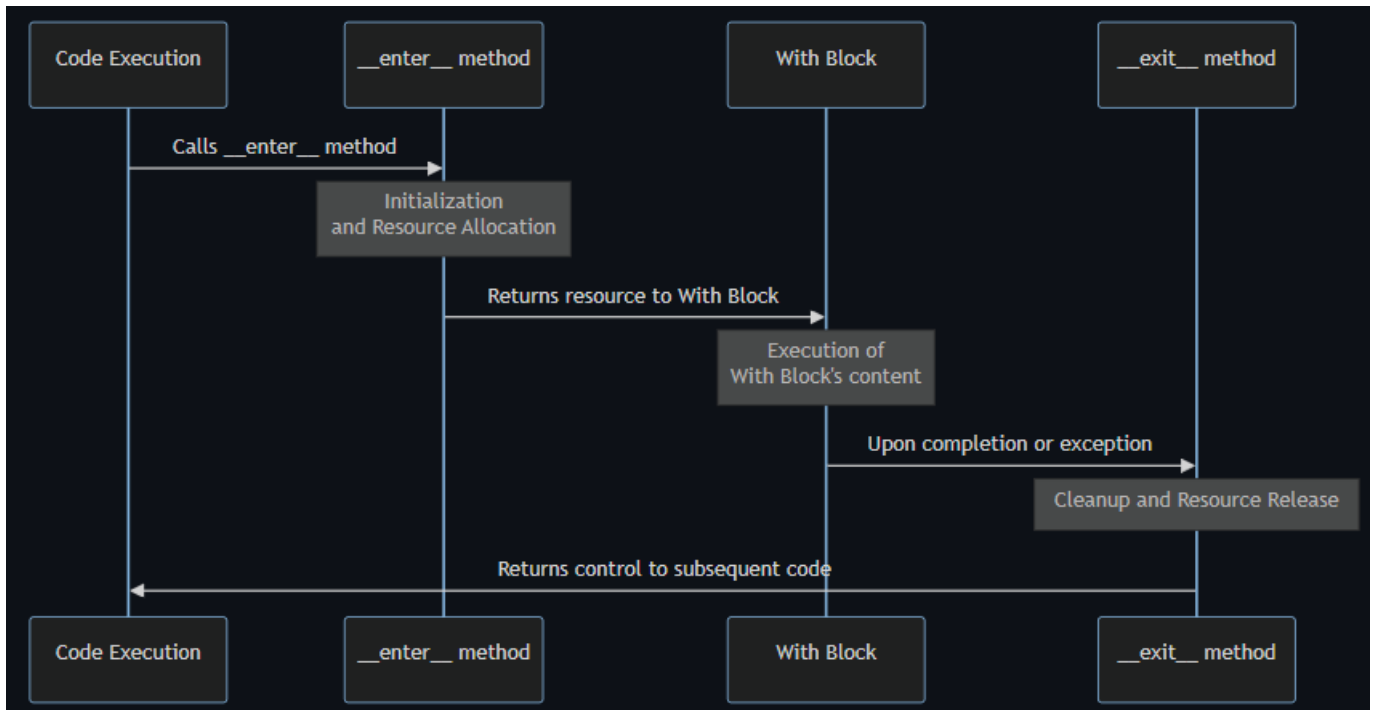
In this example, `open()` is a context manager that ensures the file is automatically closed after the block of code is executed, even if an exception is raised within the block.

This eliminates the need for explicit `file.close()` calls and makes the code cleaner and more readable.

### 1.2 Overview

**IMPORTANT:** Note that, `with` statement is syntax sugar of the programming language, under the hood it implements two magic methods:

- `__enter__`: Executed at the beginning of the block following the `with` statement. It returns the resource to be managed (e.g., a file object).
- `__exit__`: Executed at the end of the `with` block, regardless of whether an exception occurred. It handles the cleanup, like closing a file.



Let's have some practice!

### 1.3.1 Managing Temporary Files

#### Example

**Objective:** Create a temporary file that is automatically removed after use.

```

import os
import tempfile
class TemporaryFile:
    def __enter__(self):
        self.file = tempfile.NamedTemporaryFile(delete=False)
        return self.file
    def __exit__(self, exc_type, exc_val, exc_tb):
        self.file.close()
        os.remove(self.file.name)
with TemporaryFile() as temp_file:
    temp_file.write(b'Hello World!')
    print(f"Temporary file created at: {temp_file.name}")
  
```

#### Explanation

This context manager creates a temporary file in the system's designated temporary directory.

- Upon entering the context (triggering the `__enter__` method) in `with` statement, it returns a file object that can be used to write data.
- Exiting the context (triggering the `__exit__` method) automatically closes the file and removes it from the filesystem.

### 1.3.2 Execution Time

**Objective:** Measure and print the time taken to execute a code block, aiding in profiling and optimization.

#### Example

```
import time
class Timer:
    def __enter__(self):
        self.start = time.time()
        return self
    def __exit__(self, exc_type, exc_val, exc_tb):
        self.end = time.time()
        print(f"Elapsed time: {self.end - self.start:.2f} seconds")
with Timer() as t:
    # Some time-consuming operations
    for _ in range(1000000):
        pass
```

#### Explanation

The Timer context manager captures the current time upon entry and calculates the elapsed time upon exit.

- Upon entering the context (triggering the `__enter__` method) in `with` statement, it starts a countdown.
- Exiting the context (triggering the `__exit__` method) automatically finishes the countdown and prints time taken to execute a code within a context manager.

Amazing, do you agree?

### 1.3.3 Feature Toggling

**Objective:** Temporarily enable or disable application features, useful for testing or conditional feature deployment.

#### Example

```
class FeatureToggle:
    def __init__(self, feature, enabled=True):
        self.feature = feature
        self.enabled = enabled
        self.original_state = None
    def __enter__(self):
        self.original_state = getattr(settings, self.feature, None)
        setattr(settings, self.feature, self.enabled)
    def __exit__(self, exc_type, exc_val, exc_tb):
        setattr(settings, self.feature, self.original_state)
with FeatureToggle('NEW_FEATURE', enabled=True):
    # The NEW_FEATURE is temporarily enabled
    pass
```

## Explanation

This context manager temporarily change the state of a feature flag in an application's settings.

- On entering it sets the feature's state to the desired value and stores the original state.
- Upon exit, it restores the feature to its original state.

I beleive that you will find practical appliance of context managers, as you can see, it's a valuable and reliable tool.

## 2. contextlib

The `contextlib` module in Python provides utilities for working with context managers and the `with` statement. One of its most powerful features is the `contextmanager` decorator, which allows you to write a context manager using generator syntax, making it easy to create custom context managers without needing to define a class with `__enter__` and `__exit__` methods.

Now instead of classes we can use generator functions, simplifying development, readability and interaction with a codebase. Let's take a look and I will re-write the previous examples.

### 2.1 Temporary Change of Directory

```
import os
from contextlib import contextmanager
@contextmanager
def change_dir(destination):
    try:
        cwd = os.getcwd()
        os.chdir(destination)
        yield
    finally:
        os.chdir(cwd)
with change_dir("/tmp"):
    # Operations in /tmp
    print("Working in", os.getcwd())
# Back to original directory
print("Back to", os.getcwd())
```

### 2.2. Enabling Debug Mode Temporarily

For applications with a debug mode, you might want to enable it temporarily for a block of code:

```
from contextlib import contextmanager
@contextmanager
def debug_mode(enabled=True):
    original_debug = settings.DEBUG
    settings.DEBUG = enabled
    try:
        yield
```

```

    finally:
        settings.DEBUG = original_debug
with debug_mode():
    # Code that runs with debug mode enabled
    perform_debug_operations()

```

## 2.3 Error Handling

Context managers can also be used to elegantly handle exceptions that occur within the `with` block.

```

@contextmanager
def exception_handler():
    try:
        yield
    except Exception as e:
        print(f"Handled exception: {e}")
with exception_handler():
    raise ValueError("Something went wrong")

```

The examples above are more conceptual, than practical, but it's important that you understand the philosophy behind the context managers:

- **Step1:** Enter the context state.
- **Step2:** Proceed within a local scope (Do something temporally).
- **Step3:** Exit the context state and release resources.

## 3. Nested context managers

Python allows nesting of context managers, which can be useful when dealing with multiple resources that need to be managed together.

### 3.1 Managing Multiple Files

**Objective:** Reading from one file and writing to another simultaneously.

**Example**

```

with open('input.txt', 'r') as input_file, open('output.txt', 'w') as output_file:
    for line in input_file:
        output_file.write(line.upper())

```

In some cases it can be useful, if both files have direct ties to each other.

## 3.2 Nested Timeout

**Objective:** Performing nested action where each call has its own timeout period.

### Example

```
from contextlib import contextmanager
@contextmanager
def timeout(time):
    def signal_handler(signum, frame):
        raise TimeoutError("Operation timed out")

    # Do something here?
    try:
        yield
    finally:
        signal.alarm(0)
with timeout(10):
    with timeout(5):
        # Perform operation that must complete within 5 seconds
        # But the overall block should not exceed 10 seconds
        pass
```

We can set a time range using context manager between the beginning and ending.

## 3.2 Best Practices

Category	Do	Don't
Resource Management	Use context managers to explicitly manage resources, ensuring they are always properly released.	Don't use context managers where a simple try-finally block would suffice for resource management.
Exception Handling	Ensure that exceptions are properly handled or propagated when implementing <code>__exit__</code> methods or using <code>@contextmanager</code> .	Don't ignore exceptions as they can lead to hidden bugs and unreliable application behavior.

Avoid complex logic that can make the context manager difficult to read and understand. Keep it simple!

## 4. Quiz

### Question 1:

What does the `__enter__` method do in a context manager?

- A) It handles exceptions that occur within the context.
- B) It initializes the resource that needs to be managed.
- C) It cleans up and releases the resources.
- D) It validates input parameters for the resource.

### Question 2:

Which of the following is a correct way to use the `contextlib` module's `contextmanager` decorator for creating a context manager?

- A) Using a class with `__enter__` and `__exit__` methods.
- B) Using a generator function with a single `yield`.
- C) Using a regular function with multiple return statements.
- D) Using a class without implementing any methods.

### Question 3:

What is the primary benefit of using context managers for file handling in Python?

- A) Increasing the file read and write speed.
- B) Automatically closing the file regardless of whether an error occurs.
- C) Encrypting and decrypting file contents automatically.
- D) Compacting the file to save disk space.

### Question 4:

What content does 'log.txt' contain after execution?

```
with open('log.txt', 'w') as f:
    f.write('Begin logging\n')
    raise ValueError('Something went wrong!')
    f.write('End logging\n')
```

- A) 'Begin logging\nEnd logging\n'
- B) 'Begin logging\n'
- C) The file is empty.
- D) 'End logging\n'



**Question 5:**

What will happen if an exception is raised in a context manager before reaching the `yield` in a `contextmanager` decorated generator?

- A) The code will continue to execute normally.
- B) The exception will be suppressed.
- C) The `__exit__` method is called immediately.
- D) The part after `yield` will not execute.

**Question 6:**

How can you ensure that a block of code within a `with` statement completes execution before proceeding?

- A) By using the `pass` statement.
- B) By using the `finally` clause.
- C) By nesting multiple `with` statements.
- D) By handling all exceptions within the context manager.

## 5. Homework

### Task 1: Database Connection Manager

**Objective:** Create a context manager that handles opening and closing a database connection.

**Requirements:**

- Simulate opening a connection to a database.
- Ensure the connection is closed after operations, even if an error occurs.
- Handle exceptions gracefully and log them.

Happy Programming!

# Lesson 27: Concurrent and Parallel Programming

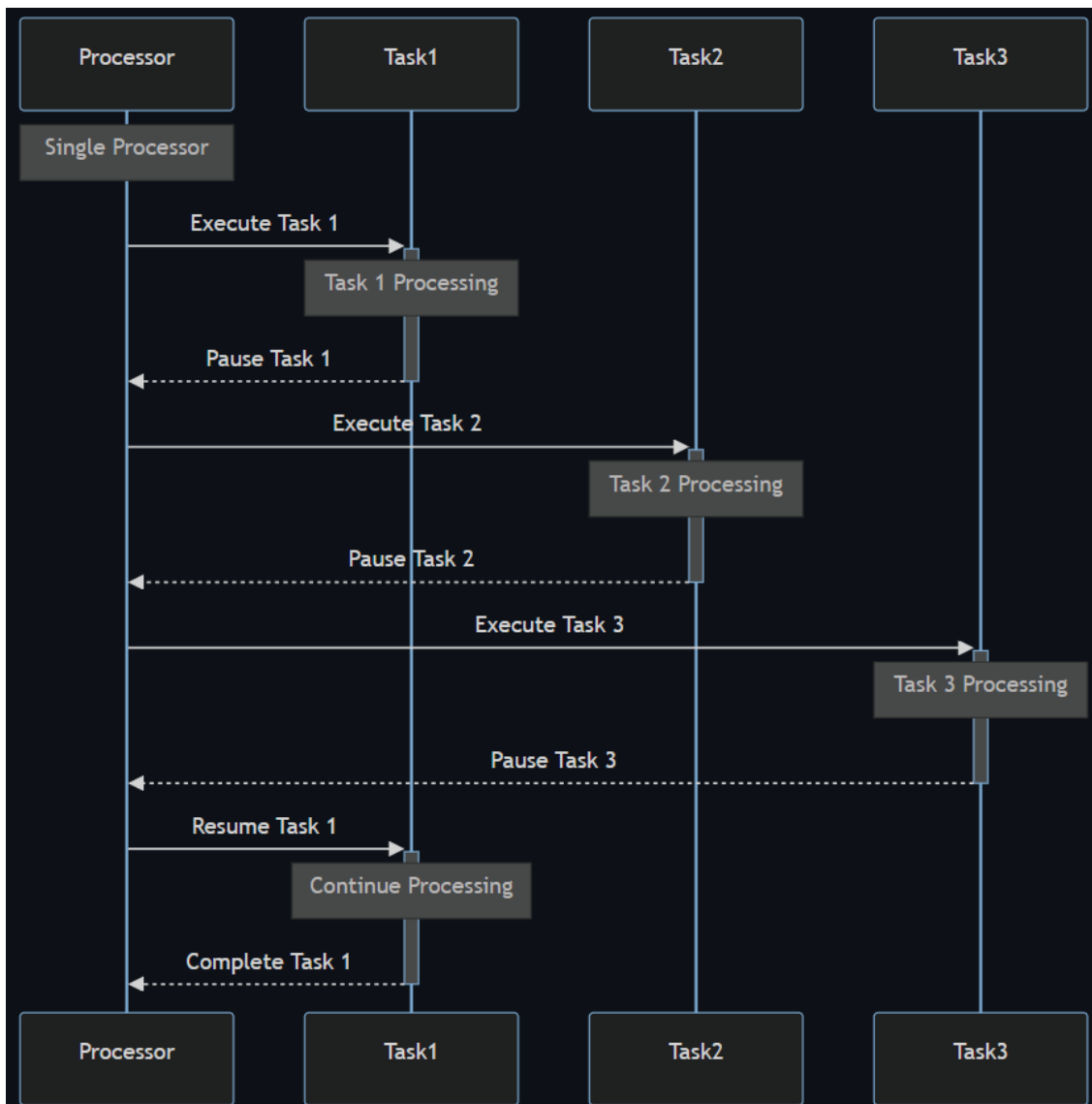
"The art of programming is the art of organizing complexity, of mastering multitude and avoiding its bastard chaos as effectively as possible." - Edsger Dijkstra

## 1. Concurrent Programming

The main problem of Python is being extremely slow in terms of performance. But we can achieve better results using practices described in sections [1-5], which is critical for developing high-performance applications in Python.

### 1.1 Definition

**Concurrency:** The ability of a program to be decomposed into parts that can run independently of each other. It's about dealing with lots of things at once.



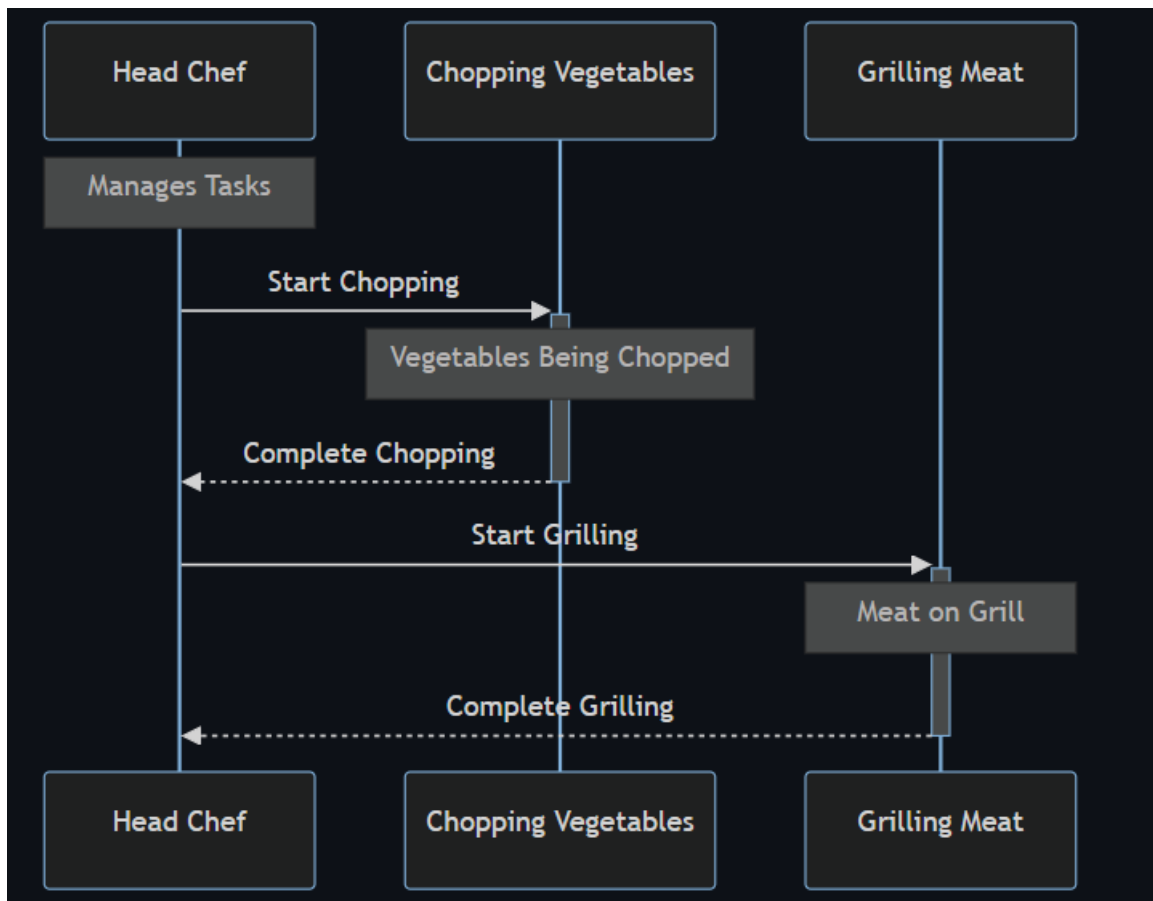
## 1.2 The Restaurant Kitchen

**Objective:** The kitchen has one head chef (the processor) and several tasks that need to be done to prepare a meal, such as chopping vegetables and grilling meats.

To represent the concurrency correctly, chef is moving between all tasks.

### 1.2.1 Concurrent

Let's take a look at how should it work:



### Explanation

- **Head Chef (Processor):** Manages and switches between tasks, but can physically only do one task at a time.
- **Tasks (Concurrent Processes):** Chopping vegetables, grilling meats, These tasks are set up and monitored concurrently.
- **Efficiency:** By managing multiple tasks and moving between them, the kitchen operates much faster.

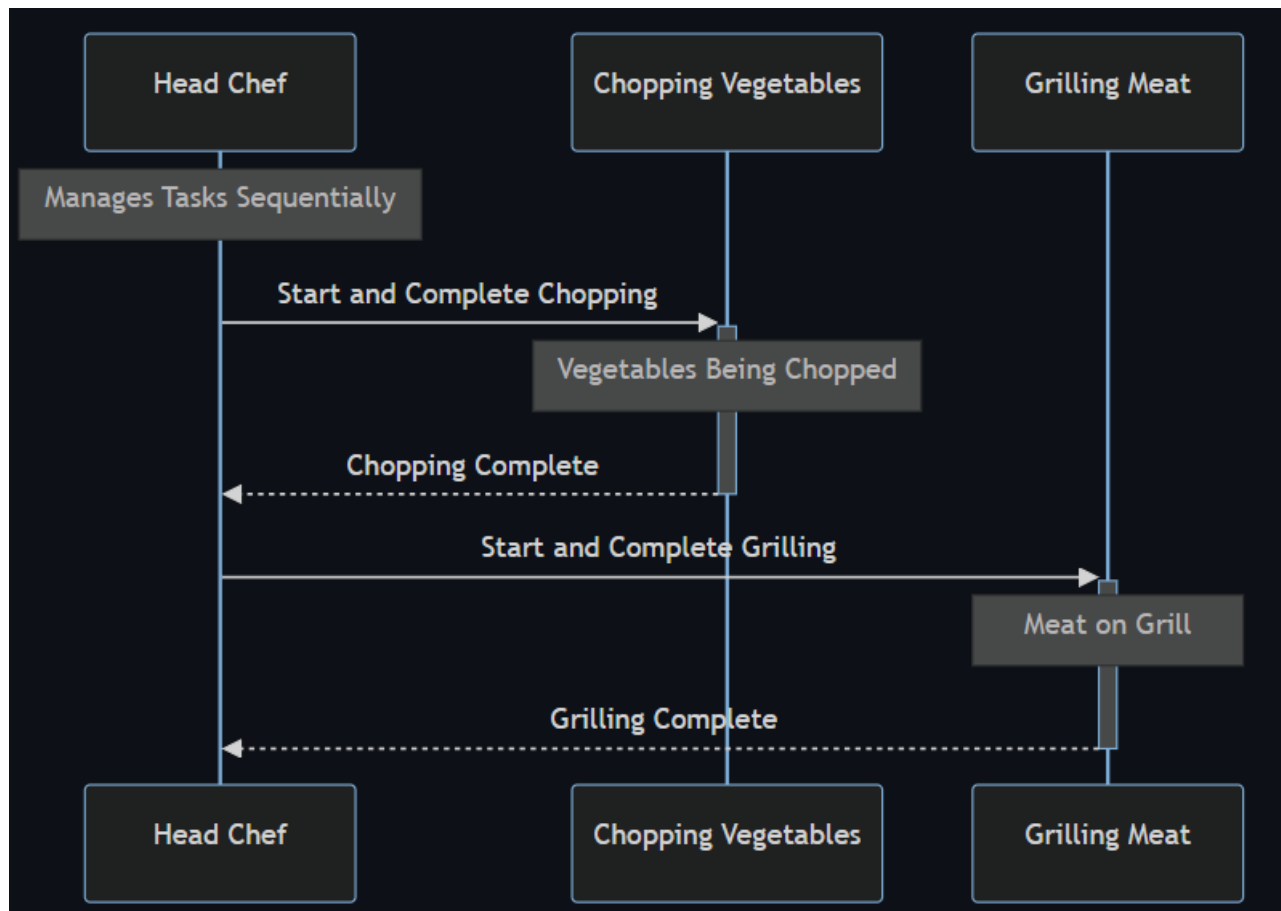
Chef is able to perform only one task at a time (due to single-threading / GIL (Global Interpreter Lock) in Python ), though all components of the meal are prepared concurrently.

GIL will be covered during the next lesson, for now just try to understand the idea behind concurrent programming.

## 1.2.2 Synchronous

In a synchronous kitchen operation, tasks are completed one after another without overlap. The head chef focuses on one task completely before moving on to the next task.

That's the way we implemented applications before.



## 1.3 The Bank

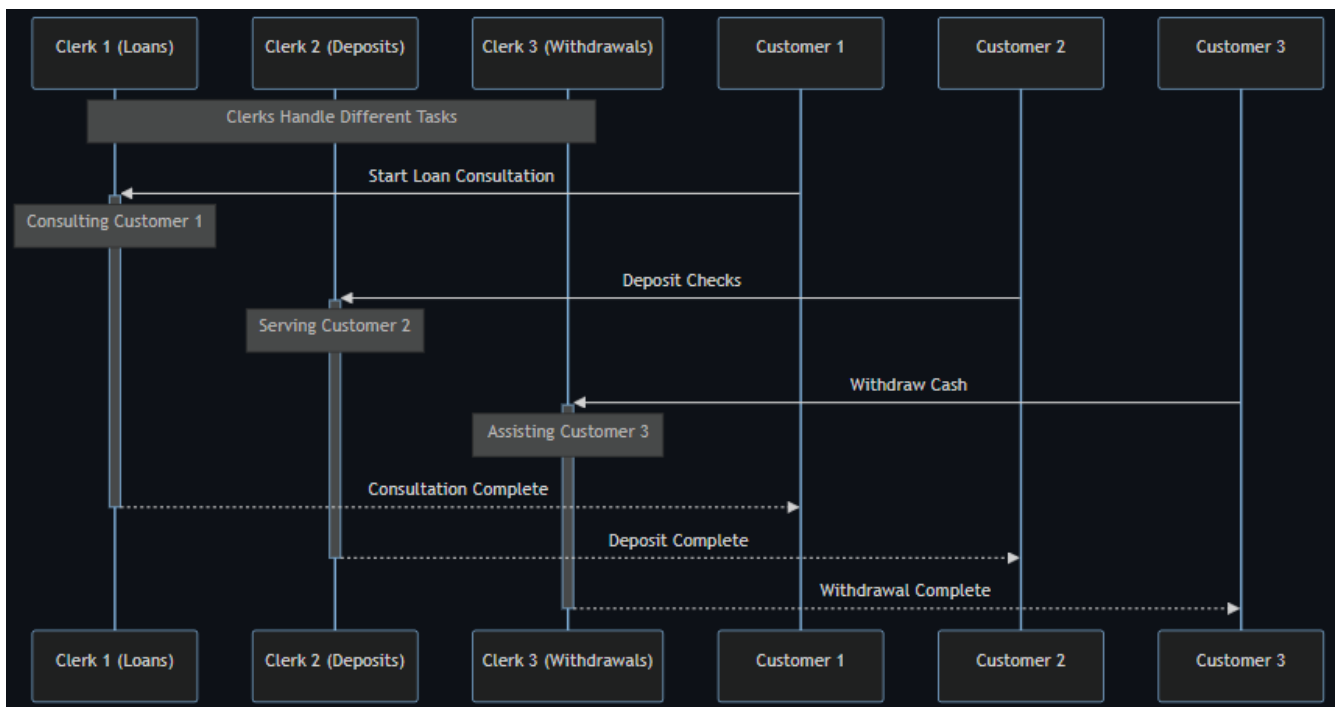
**Objective:** Imagine a bank with a single service counter and multiple customers needing different services: depositing checks, withdrawing cash, and consulting on loan products.

### 1.3.1 Concurrent

In a concurrent version, multiple clerks (processors) are available to handle different banking tasks simultaneously, leading to efficient customer service and reduced waiting times.

This setup allows the bank to serve multiple customers at the same time, with each clerk focused on a specific task.

Let's take a look at how such model works in practice:

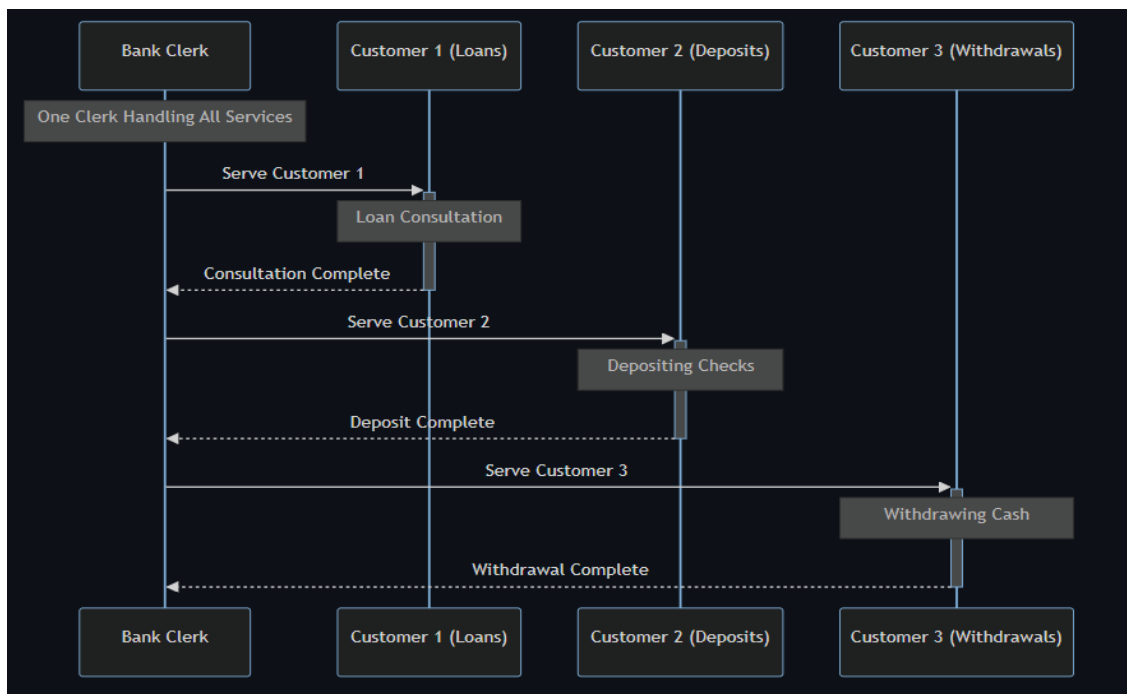


## Explanation

- **Clerks (Processors):** Each clerk is capable of handling a different banking task simultaneously.
- **Tasks (Concurrent Processes):** Loan consultations, depositing checks, and withdrawing cash are examples of tasks being handled concurrently.
- **Efficiency:** This concurrent operation allows the bank to maximize its throughput, serving more customers efficiently and reducing overall wait times.

### 1.3.2 Synchronous

In a smaller bank with only one clerk, customers must be served one at a time. This synchronous approach means the clerk focuses on completing one customer's transaction before starting the next.



Again, this may lead to increased wait times, which is a bad case for our application. That's why concurrent approach will be the best in terms of building your applications.

## 2. Parallel Programming

As processors have evolved, increasing their clock speed has become more challenging due to physical and thermal limitations.

The solution has been to add more cores, allowing for more operations to be carried out simultaneously.

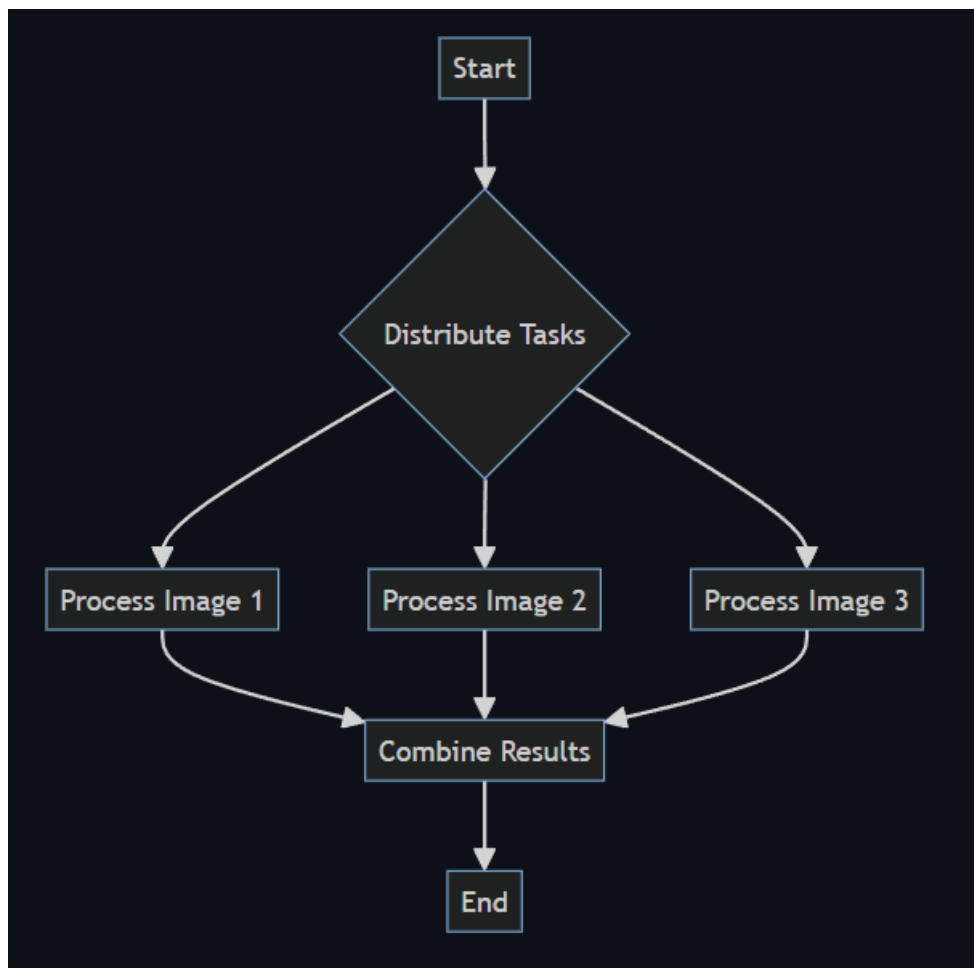
Parallel programming is designed to take advantage of this architecture by dividing tasks across these cores, significantly speeding up processing times for suitable tasks.

### 2.1 Definition

**Parallel programming:** is an approach that allows for the execution of many computations or processes simultaneously.

### 2.2 Image Processing App

**Objective:** In a photo editing application, multiple images need to be processed to apply a filter. Each image can be processed independently of the others, making this task an excellent candidate for parallel programming.



### Explanation

- **Task Distribution:** The application divides the batch of images into individual tasks.
- **Parallel Processing:** Each image is processed in parallel, utilizing separate CPU cores.
- **Efficiency:** This approach significantly reduces the total processing time compared to processing each image sequentially.

Just as in life, time is the most valuable asset which should be used accordingly, that's why we usually refer to techniques above - to save some time, even if it's a time of execution.

## 3 Key Differences

Let's summarise everything into the table, so that during designing of your application, you as a programmer would refer to:

Aspect	Concurrent Programming	Parallel Programming
Definition	Focuses on managing and executing multiple tasks that can run	Involves executing multiple computations or processes simultaneously, typically by utilizing multiple CPU cores.

Aspect	Concurrent Programming	Parallel Programming
	independently but not necessarily simultaneously.	
<b>Execution</b>	Tasks are decomposed into parts that can run independently, often interleaved or overlapping in time.	Tasks are executed at the same time, utilizing the computing power of multiple processing units.
<b>Use Case</b>	Best for I/O-bound operations where tasks spend a significant amount of time waiting for external operations like network or disk I/O.	Best for CPU-bound tasks that require heavy computation and can be divided into independent subtasks for simultaneous execution.
<b>Python Tools</b>	Threading, AsyncIO	Multiprocessing, <code>concurrent.futures (ProcessPoolExecutor)</code>
<b>GIL Impact</b>	Affected by Python's GIL, making true parallel execution of threads impossible in CPython. However, tasks that are I/O-bound can still benefit from concurrency.	Bypasses the GIL by using separate processes instead of threads, allowing true parallel execution of code.



Happy self-reflection and welcome to the Fast Python!

Sorry for raw and boring theory, without it we won't be able to create async applications.

## **4. Homework**

Come up with concurrent and parallel flows from the real world and visualise them in diagrams.

# Lesson 28: Threading

"Concurrency is not parallelism, it's better." - Rob Pike

## 1. Introduction to Threads

In the world of software development, the ability to perform multiple operations simultaneously can significantly enhance the responsiveness and performance of an application. Python's threading module offers a powerful tool for achieving such concurrency.

### 1.1 Definition of Threads

**Thread** often referred to as a lightweight process, is the smallest unit of processing that can be performed in an OS.

In most modern operating systems, a thread exists within a process and shares resources such as memory, yet can execute independently. Threads within the same process can execute concurrently, making efficient use of CPU resources.

**NOTE:** In order to use threading in Python we would need to import a built in module `threading`.

Let's take a look on the following examples:

#### Example

```
import threading
def print_numbers():
    for i in range(5):
        print(i)
# Create a thread to run the print_numbers function
thread = threading.Thread(target=print_numbers)
thread.start()
# Continue executing the main program while the thread runs concurrently
print("Thread started!")
```

#### Output

```
0
1
Thread started!
2
3
4
```

**NOTE:** Thread is a separate object of `<class 'threading.Thread'>`.

## 1.2 Processes vs Threads

There is one important entity called process, people always confuse them with threads. Simply speaking, process is a separate application, a separate program, and has a few distinguishing qualities from threads.

Take a look at the table below:

Aspect	Process	Thread
Memory	Separate memory space	Shared memory space
Creation	Slow and resource-intensive	Quick and efficient
Communication	Requires inter-process communication (IPC)	Directly communicate via shared variables
Dependency	Operates independently	Part of a process

You can view the processes calling `top` command on Linux/Mac OS systems or running the task manager o Windows.

### Example

```
$ top
```

### Output

```
top - 14:30:52 up 5 days, 4:45, 1 user, load average: 1.55, 1.95, 1.65
Tasks: 404 total, 1 running, 403 sleeping, 0 stopped, 0 zombie
%Cpu(s): 1.6 us, 3.3 sy, 0.0 ni, 94.3 id, 0.0 wa, 0.0 hi, 0.0 si, 0.0 st
MiB Mem : 15346.1 total, 301.4 free, 7173.1 used, 7871.6 buff/cache
MiB Swap: 2048.0 total, 2045.1 free, 2.9 used, 7485.5 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM    TIME+  COMMAND
 3561 hacking    20   0 6011036 647128 158104 S   23.5   4.1 222:17.29 gnome-shell
 3277 hacking    20   0   25.1g 294564 195248 S   11.8   1.9 221:18.28 Xorg
111069 hacking    20   0  13552    480    3456 R   11.8   0.0   0:00.05 top
   608 systemd+    20   0   14836   6756   5912 S    5.9   0.0   2:49.58 systemd-oomd
 5181 hacking    20   0   32.7g 725668 208848 S    5.9   4.6 84:02.69 chrome
106625 root        20   0      0      0      0 D    5.9   0.0   0:08.64 kworker/u32:0+events_unbound
110979 hacking    20   0 1134.1g 318836 128032 S    5.9   2.0   0:08.76 chrome
111008 hacking    20   0 1132.0g 145568 108188 S    5.9   0.9   0:01.25 chrome
```

As it was mentioned in a previous lesson, in Python you would need to use different modules to interact with threads/processes - `threading/multiprocessing`

Let's dive into practice and focus on `threading` this module.

## 1.3 Threading Use Cases

Threading is particularly beneficial in scenarios where an application needs to maintain responsiveness to user input while performing other tasks in the background, such as:

- **GUI Applications:** Keeping the UI responsive while processing data.
- **I/O Bound Applications:** Performing multiple network or disk operations concurrently.
- **Real-Time Data Processing:** Monitoring input from real-time data sources without blocking.

Let's take a look on a couple of examples below:

### 1.3.1 File Downloads

**Objective:** Consider an application that needs to download multiple files from the internet simultaneously.

The default implementation of apps which is known for us already will be in a synchronous manner.

#### Example

```
import time
file_urls = ["http://example.com/file1.jpg", "http://example.com/file2.jpg",
             "http://example.com/file3.jpg"]
def download_file_sync(file_url):
    print(f"Starting download from {file_url}")
    time.sleep(3) # Simulate download time
    print(f"Completed download from {file_url}")
    print("\n")
start_time_sync = time.time()
for url in file_urls:
    download_file_sync(url)
end_time_sync = time.time()
print(f"All files have been downloaded sequentially in {end_time_sync -
start_time_sync} seconds.")
```

#### Output

```
Starting download from http://example.com/file1.jpg
Completed download from http://example.com/file1.jpg
Starting download from http://example.com/file2.jpg
Completed download from http://example.com/file2.jpg
Starting download from http://example.com/file3.jpg
Completed download from http://example.com/file3.jpg
All files have been downloaded sequentially in 9.006875276565552 seconds.
```

#### Explanation

We can see from the output that downloading files sequentially, could be time-consuming, and it's not the best approach which could be.

The application can download multiple files in parallel using threading. This significantly reduces the overall time required for all downloads to complete.

Let's re-write it applying concurrency:

## Example

```
import threading
import time
def download_file(file_url):
    print(f"Starting download from {file_url}")
    time.sleep(3) # Simulate download time
    print(f"Completed download from {file_url}")
file_urls = ["http://example.com/file1.jpg", "http://example.com/file2.jpg",
"http://example.com/file3.jpg"]
start_time = time.time()
for url in file_urls:
    threading.Thread(target=download_file, args=(url,)).start()
# Assuming downloads finish in 3 seconds, we wait slightly longer
time.sleep(3.5)
end_time = time.time()
print(f"All files have been downloaded concurrently in {end_time - start_time}
seconds.")
```

**Note:** In real-world applications, relying on `sleep` to wait for threads can be unreliable. Proper synchronization methods or thread management techniques should be used instead.

## Output

```
Starting download from http://example.com/file1.jpg
Starting download from http://example.com/file2.jpg
Starting download from http://example.com/file3.jpg
Completed download from http://example.com/file1.jpg
Completed download from http://example.com/file2.jpg
Completed download from http://example.com/file3.jpg
All files have been downloaded concurrently in 3.5027055740356445 seconds.
```

We can make a conclusion, that concurrent approach is much better and faster than sequential.

## 1.4 Web Server Request Handling

**Objective:** Let's create a web-server which handles the incoming requests from the client and check how Synchronous/Concurrent approaches could impact the time of request processing in general.

### Example

Synchronous approach can lead to significant delays, especially if each request involves time-consuming operations.

```
import time
def handle_client_request_sync(request_id):
    print(f"Synchronously handling request {request_id}")
```

```

        time.sleep(2) # Simulate request processing time
        print(f"Request {request_id} completed\n")
requests = [1, 2, 3, 4, 5]
start_time_sync = time.time()
for request_id in requests:
    handle_client_request_sync(request_id)
end_time_sync = time.time()
print(f"All requests have been handled sequentially in {end_time_sync -
start_time_sync} seconds.")

```

## Output

```

Synchronously handling request 1
Request 1 completed
Synchronously handling request 2
Request 2 completed
...
Synchronously handling request 5
Request 5 completed
All requests have been handled sequentially in 10.005 seconds.

```

Again, getting the total time - the sum of processing all requests. Let's re-write this:

## Example

Here, the web server handles each client request in a separate thread, allowing multiple requests to be processed in parallel. This significantly improves response time of the server.

```

import threading
import time
def handle_client_request(request_id):
    print(f"Concurrently handling request {request_id}")
    time.sleep(2) # Simulate request processing time
    print(f"Request {request_id} completed")
requests = [1, 2, 3, 4, 5]
start_time = time.time()
threads = []
for request_id in requests:
    thread = threading.Thread(target=handle_client_request, args=(request_id,))
    threads.append(thread)
    thread.start()
# Assuming each request finishes in 2 seconds, wait a bit longer than that
time.sleep(2.5)
end_time = time.time()
print(f"All requests have been handled concurrently in {end_time - start_time}
seconds.")

```

## Output

```

Concurrently handling request 1

```

```
Concurrently handling request 2
...
Concurrently handling request 5
Request 1 completed
Request 2 completed
...
Request 5 completed
All requests have been handled concurrently in 2.501 seconds.
```

**Note:** Real-world server applications would manage thread completion more robustly than using `sleep`.

Same here, don't hesitate to use threading in your applications to reduce the total execution time, but first I would want to take a deeper look on all features of `threading` module and introduce you to potential rabbit holes, which are extremely hard to debug :)

## 2. Threading

In computing, a thread is similar to each task you perform — it's a sequence of instructions that can be executed independently while contributing to the overall process.

### 2.1 Starting Threads

Starting a thread means initiating a separate flow of execution. Think of it as hiring another cook in the kitchen to handle a different task concurrently, so breakfast gets ready faster.

#### Example

```
from threading import Thread
import time
def boil_water():
    print("Boiling water...")
    time.sleep(3) # Simulating the time it takes to boil water
    print("Water boiled!")
# Create a thread for boiling water
water_thread = Thread(target=boil_water)
# Start the thread
water_thread.start()
```

#### Output

```
Boiling water...

Water boiled!
```

## Explanation

In this example, calling `water_thread.start()` begins the execution of `boil_water` in a separate thread. This allows the program (or the main cook) to perform other tasks without waiting for the water to boil.

## 2.2 Joining Threads

Joining threads is a way of synchronizing tasks in the kitchen.

You wouldn't want to serve breakfast without the toast being ready, right? Joining ensures that the main flow of execution (the main thread) waits for other threads to complete their tasks before proceeding.

### Example

```
from threading import Thread
import time
def boil_water():
    print("Boiling water...")
    time.sleep(3) # Simulating the time it takes to boil water
    print("Water boiled!")
# Create a thread for boiling water
water_thread = Thread(target=boil_water)
# Start the thread
water_thread.start()
# Wait for the water to boil
water_thread.join()
print("Now that the water has boiled, let's make coffee.")
```

### Output

```
Boiling water...
Water boiled!
Now that the water has boiled, let's make coffee.
```

**IMPORTANT:** WITHOUT `join()` method we would not be able to synchronise threads and would have got the following output.

**IMPORTANT:** This can lead to very unpleasant bugs and situations in production.

### Output

```
Boiling water...
Now that the water has boiled, let's make coffee.
Water boiled!
```



## Explanation

Here, `water_thread.join()` acts as an instruction to wait: "Don't start making coffee until the water has boiled." It ensures that the water boiling process is complete before moving on to the next step.

## 2.3 Locks

Locks, or mutexes, are tools for ensuring that only one thread at a time can execute a specific block of code. This is particularly important when multiple threads interact with shared data or resources.

### Example

**Objective:** Imagine you're back in the kitchen, and this time you have only one frying pan, but multiple dishes that require it. To avoid a mess (race condition), you use a lock (the frying pan) to ensure only one dish is prepared at a time.

```
from threading import Thread, Lock
import time
# The shared resource
frying_pan = Lock()
def cook_dish(dish_name):
    with frying_pan:
        print(f"Starting to cook {dish_name} with the frying pan.")
        time.sleep(2) # Simulate cooking time
        print(f"{dish_name} is done!")
dishes = ["Scrambled eggs", "Pancakes", "Bacon"]
# Creating threads for each cooking task
for dish in dishes:
    Thread(target=cook_dish, args=(dish,)).start()
```

### Explanation

In this example, the `frying_pan` lock ensures that only one thread (dish being cooked) can access the critical section (use the frying pan) at any given time. This prevents the dishes from "clashing," which in programming terms, translates to avoiding data corruption or unexpected outcomes due to concurrent modifications.

**IMPORTANT:** In case we would not want to use the Mutex mechanism, we would start cooking everything simultaneously on the `frying_pan` and encounter a mess.

### Output

```
Starting to cook Scrambled eggs with the frying pan.
Scrambled eggs is done!
Starting to cook Pancakes with the frying pan.
```

```
Pancakes is done!
Starting to cook Bacon with the frying pan.
Bacon is done!
```

Frankly speaking mutex mechanisms were the hardest mechanisms to understand, so I would like to include one more example, so that you understand it better.

## Example

**Objective:** Consider a bank account with a balance that can be accessed by multiple transactions at the same time.

**IMPORTANT:** Without proper synchronization, simultaneous deposits and withdrawals could lead to incorrect account balances.

Let's implement the code which simulates deposit and withdraw operations:

```
from threading import Thread, Lock
import random
# Bank account balance
balance = 100
account_lock = Lock()
def deposit():
    global balance
    for _ in range(5):
        with account_lock:
            amount = random.randint(10, 50)
            balance += amount
            print(f"Deposited ${amount}. New balance: ${balance}.")
def withdraw():
    global balance
    for _ in range(5):
        with account_lock:
            amount = random.randint(10, 50)
            if balance >= amount:
                balance -= amount
                print(f"Withdrew ${amount}. New balance: ${balance}.")
            else:
                print("Insufficient funds for withdrawal.")
# Creating threads for depositing and withdrawing
deposit_thread = Thread(target=deposit)
withdraw_thread = Thread(target=withdraw)
deposit_thread.start()
withdraw_thread.start()
deposit_thread.join()
withdraw_thread.join()
print(f"Final account balance: ${balance}")
```

## Output

```
Deposited $12. New balance: $112.
```

```
Deposited $38. New balance: $150.  
Deposited $31. New balance: $181.  
Deposited $28. New balance: $209.  
Deposited $39. New balance: $248.  
Withdrew $26. New balance: $222.  
Withdrew $15. New balance: $207.  
Withdrew $14. New balance: $193.  
Withdrew $23. New balance: $170.  
Withdrew $17. New balance: $153.  
Final account balance: $153
```

## Explanation

In this example, each transaction (deposit or withdrawal) acquires the lock before modifying the balance and releases it afterward, ensuring the balance updates are atomic and preventing race conditions.

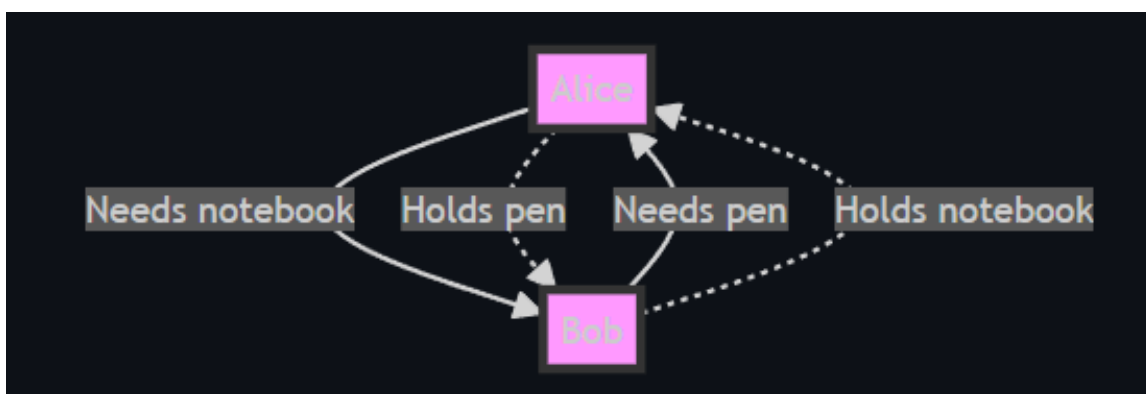
Try modifying the codebase and see what happens without locks.

## 2.4 DeadLocks

In concurrent programming, a deadlock is a situation where two or more threads are blocked forever, waiting for each other to release a resource they need.

Imagine two friends, Alice and Bob, who need to borrow a pen and a notebook to jot down an idea. Alice has the pen, Bob has the notebook, and both need to use the other item at the same time.

If neither is willing to lend their item before getting the other, they end up in a standstill - this is a deadlock.



In terms of threads, a deadlock can occur when:

- Thread A holds Lock 1 and waits for Lock 2.
- Thread B holds Lock 2 and waits for Lock 1.
- Neither thread can proceed, leading to a permanent block.

### 2.4.1 How to avoid deadlocks?

**Lock Ordering:** Ensure that all threads acquire locks in the same order, even if they need to acquire multiple locks. This consistency prevents circular wait conditions.

## Example

```
from threading import Thread, Lock
import time
# Define the locks (resources)
lockA = Lock()
lockB = Lock()
# Bad practice: Potential for deadlock
def thread1_bad():
    with lockA:
        print("Thread 1 acquired Lock A")
        time.sleep(1) # Simulate work, increasing the chance of deadlock
        with lockB:
            print("Thread 1 acquired Lock B")
            # Perform some action
        print("Thread 1 completed work")
def thread2_bad():
    with lockB:
        print("Thread 2 acquired Lock B")
        time.sleep(1)
        with lockA:
            print("Thread 2 acquired Lock A")
            # Perform some action
        print("Thread 2 completed work")
# Good practice: Avoiding deadlock
def thread1_good():
    with lockA:
        print("Thread 1 (good) acquired Lock A")
        time.sleep(1)
        with lockB:
            print("Thread 1 (good) acquired Lock B")
            # Perform some action
        print("Thread 1 (good) completed work")
def thread2_good():
    with lockA:
        print("Thread 2 (good) acquired Lock A")
        time.sleep(1)
        with lockB:
            print("Thread 2 (good) acquired Lock B")
            # Perform some action
        print("Thread 2 (good) completed work")
# Running the bad practice example (potential deadlock)
# Uncomment the lines below to run the bad practice scenario.
# Note: This may hang your program due to deadlock.
# t1_bad = Thread(target=thread1_bad)
# t2_bad = Thread(target=thread2_bad)
# t1_bad.start()
# t2_bad.start()
# t1_bad.join()
# t2_bad.join()
# Running the good practice example
t1_good = Thread(target=thread1_good)
t2_good = Thread(target=thread2_good)
t1_good.start()
t2_good.start()
t1_good.join()
```

```
t2_good.join()
```

## Output

```
Thread 1 (good) acquired Lock A
Thread 1 (good) acquired Lock B
Thread 1 (good) completed work
Thread 2 (good) acquired Lock A
Thread 2 (good) acquired Lock B
Thread 2 (good) completed work
```

**Using a Timeout:** When attempting to acquire a lock, using a timeout can prevent a thread from waiting indefinitely. If the lock isn't acquired within the timeout period, the thread can release any locks it holds and retry later, thus breaking potential deadlock cycles.

## Example

```
from threading import Lock
import time
lock1 = Lock()
lock2 = Lock()
def try_to_avoid_deadlock():
    while True:
        if lock1.acquire(timeout=1):
            print("Lock 1 acquired")
            time.sleep(0.5) # Simulate work
            if lock2.acquire(timeout=1):
                print("Lock 2 acquired")
                # Do something with both resources
                lock2.release()
            lock1.release()
            break # Exit after work is done
        print("Retrying for locks")
try_to_avoid_deadlock()
```

## Output

```
Lock 1 acquired

Lock 2 acquired
```

**IMPORTANT:** Whenever possible, use higher-level synchronization primitives like `Queue`, which are designed to handle concurrency in a safe way and can help avoid low-level deadlock issues.

Let's take a look at a practical example, instead of theory

## Example

Imagine two threads needing to access two shared resources: a database connection and a file. To avoid deadlock, we can apply lock ordering.

```
from threading import Thread, Lock
database_lock = Lock()
file_lock = Lock()
def thread1_routine():
    with database_lock:
        print("Thread 1 acquired the database lock")
        with file_lock:
            print("Thread 1 acquired the file lock")
            # Access database and file
def thread2_routine():
    with database_lock:
        print("Thread 2 acquired the database lock")
        with file_lock:
            print("Thread 2 acquired the file lock")
            # Access database and file
thread1 = Thread(target=thread1_routine)
thread2 = Thread(target=thread2_routine)
thread1.start()
thread2.start()
thread1.join()
thread2.join()
```

## Output

```
Thread 1 acquired the database lock
Thread 1 acquired the file lock
Thread 2 acquired the database lock
Thread 2 acquired the file lock
```

By ensuring both threads acquire the `database_lock` before trying for the `file_lock`, we prevent a deadlock scenario where each thread holds one resource and waits for the other.

## 2.5 Conditions

A `Condition` object in threading allows one or more threads to wait until they are notified by another thread.

This is useful when you need to ensure certain conditions are met before a thread continues execution.

## Example

Imagine a scenario in a restaurant kitchen where the chef (Thread A) must wait until the ingredients are prepared by the kitchen staff (Thread B) before cooking.

```

from threading import Condition, Thread
import time
condition = Condition()
dish = None
def chef():
    global dish
    with condition:
        print("Chef is waiting for ingredients to be prepared...")
        condition.wait() # Chef waits until ingredients are ready
        print(f"Chef starts cooking with the prepared ingredients: {dish}.")
def kitchen_staff():
    global dish
    with condition:
        dish = "Fresh Tomatoes and Basil"
        print("Kitchen staff has prepared the ingredients.")
        condition.notify() # Notify the chef that the ingredients are ready
Thread(target=kitchen_staff).start()
time.sleep(1) # Simulating time delay for better output readability
Thread(target=chef).start()

```

## Output

```

Kitchen staff has prepared the ingredients.
Chef is waiting for ingredients to be prepared...

```

## 2.6 Events

An Event is a simpler synchronization object compared to a Condition. An event manages an internal flag that threads can set (`event.set()`) or clear (`event.clear()`). Threads can wait for the flag to be set (`event.wait()`).

### Example

```

from threading import Event, Thread
import time
start_event = Event()
def background_process():
    print("Background process is waiting to start.")
    start_event.wait() # Wait until the event is set
    print("Background process has started.")
def user_signal():
    time.sleep(3) # Simulate the user taking time to provide input
    print("User signals to start the background process.")
    start_event.set() # Signal the background process to start
Thread(target=background_process).start()
Thread(target=user_signal).start()

```

I haven't really used Events and Conditions on practice in production code, only in sandbox, but it's better to know they exist.

## 3. Global Interpreter Lock (GIL)

The Global Interpreter Lock (GIL) is one of the most controversial features of Python. It's a mutex that protects access to Python objects, preventing multiple threads from executing Python bytecodes at once.

### 3.1 Why does GIL exist?

The GIL was introduced to avoid the complexities and potential issues (such as race conditions, deadlocks, and data corruption) associated with multi-threaded access to Python objects.

### 3.2. Race Conditions

A race condition occurs when two or more threads access shared data and they try to change it at the same time.

Because the thread scheduling algorithm can swap between threads at any time, you don't know the order in which the threads will attempt to access the shared data. This can lead to unpredictable outcomes. Be extremely carefull as it is too hard to debug and also it kills your neuro system.

#### Example

Consider two threads incrementing the same counter:

```
from threading import Thread
# Shared variable
counter = 0
# Function to increment counter
def increment():
    global counter
    for _ in range(1000000):
        counter += 1
# Create threads
thread1 = Thread(target=increment)
thread2 = Thread(target=increment)
# Start threads
thread1.start()
thread2.start()
# Wait for both threads to complete
thread1.join()
thread2.join()
print(f"Final counter value: {counter}")
```

#### Output



```
Final counter value with lock: 18045604
```

## Explanation

You might expect the final value of `counter` to be 2,000,000, but due to race conditions, it might be less because increments can be lost when both threads read, increment, and write back the value simultaneously.

The most straightforward and logical way to avoid race conditions is by using locks to synchronize access to shared resources.

```
from threading import Thread, Lock
counter = 0
lock = Lock()
def safe_increment():
    global counter
    for _ in range(1000000):
        with lock:
            counter += 1
thread1 = Thread(target=safe_increment)
thread2 = Thread(target=safe_increment)
thread1.start()
thread2.start()
thread1.join()
thread2.join()
print(f"Final counter value with lock: {counter}")
```

## Output

```
Final counter value with lock: 2000000
```

## Explanation

This example ensures that only one thread can increment the counter at a time, preventing race conditions.

It's extremely hard to find out the real cause of race conditions, so be always careful with shared resources across the program.

## 4. Thread Communication

Consider a server handling web requests, where it's crucial to maintain a log of all requests for monitoring, analysis, and debugging purposes.

However, writing logs directly to a file for every request can significantly impact performance due to I/O operations.

In this scenario, a background log processing system can be implemented using producer-consumer pattern:

## Example

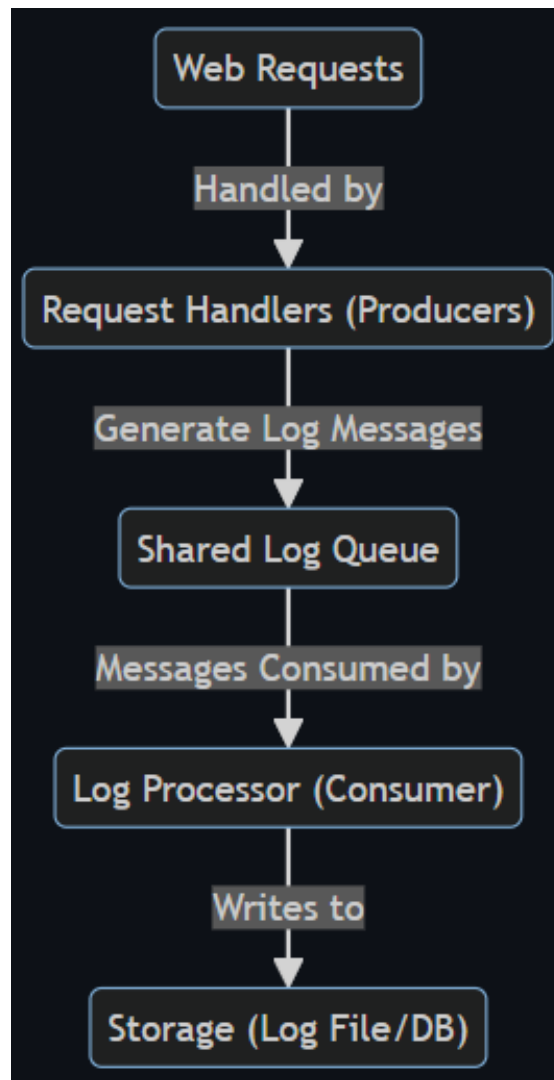
```
import queue
import threading
import time
log_queue = queue.Queue()
def handle_request(request_id):
    print(f"Handling request {request_id}")
    time.sleep(1)
    log_message = f"Request {request_id} handled"
    # Place log message in queue
    log_queue.put(log_message)
def process_logs():
    while True:
        if not log_queue.empty():
            log_message = log_queue.get()
            # Simulate log processing (e.g., writing to a file)
            print(f"Logging: {log_message}")
            log_queue.task_done()
        else:
            # Wait for new log messages to avoid busy waiting
            time.sleep(1)
# Start threads for request handling
for request_id in range(5):
    threading.Thread(target=handle_request, args=(request_id,)).start()
# Start the log processing thread
log_thread = threading.Thread(target=process_logs)
log_thread.start()
```

## Output

```
Handling request 0
Handling request 1
Handling request 2
Handling request 3
Handling request 4
Logging: Request 2 handled
Logging: Request 0 handled
Logging: Request 4 handled
Logging: Request 3 handled
Logging: Request 1 handled
```

**NOTE:** In a real application, the log processing thread would run indefinitely or until a shutdown signal is received. Btw, that's where daemon threads can come in handy.

## Explanation



**Producer (Request Handler):** Each thread handling web requests also generates log messages corresponding to request processing. Instead of writing logs immediately, these messages are placed in a shared queue.

**Consumer (Log Processor):** A logging thread consumes messages from the queue and processes them, writing to a file, database, or external logging service. This separation allows request handlers to remain responsive and offloads the I/O-heavy logging task.

This is my favourite pattern and I recommend you to implement it with high level structures such as `Queue` which handles logic under the hood.

## 5. Daemon Threads

A daemon thread runs in the background and is not meant to hold up the program from exiting. Unlike regular (non-daemon) threads, the program can quit even if daemon threads are still running.

They are typically used for tasks that run in the background without requiring explicit management by the programmer.

### Use Cases

**Background Services:** Daemon threads are ideal for background tasks, such as periodic data backup, system monitoring, or managing connections in a network server.

**Resource Management:** They can be used for automatic resource cleanup, like closing file handles or network connections when not in use.

**Asynchronous Execution:** Executing tasks that should not interfere with the main program flow, such as logging, data fetching, or heartbeats in a network protocol.

### Example

```
import threading
import time
def background_task():
    while True:
        print("Background task running...")
        time.sleep(1)
# Create a daemon thread
daemon_thread = threading.Thread(target=background_task, daemon=True)
daemon_thread.start()
# Main program will run for 5 seconds before exiting
time.sleep(5)
print("Main program is exiting.")
```

### Output

```
Background task running...
Background task running...
Background task running...
Background task running...
Background task running...
Main program is exiting.
```

### Explanation

In this example, `background_task` runs indefinitely as a daemon thread, printing a message every second.

The main program sleeps for 5 seconds and then exits. Because the daemon thread is running in the background, it does not prevent the main program from exiting.

Once the main program exits after 5 seconds, the daemon thread is also terminated.

## 6. Thread Pooling

The `ThreadPoolExecutor` class from the `concurrent.futures` module provides a high-level interface for asynchronously executing callables. The executor manages a pool of worker threads to which tasks can be submitted.

**IMPORTANT:** We don't need manually to use locks or any other synchronisation techniques, that's all handled by `ThreadPoolExecutor`, and generally, the following approach is used in production due to its reliability.

### 6.1 Practice

When creating a `ThreadPoolExecutor`, you can specify the maximum number of threads in the pool. The executor will manage these threads for you, creating new threads as tasks are submitted and reusing idle threads whenever possible.

```
from concurrent.futures import ThreadPoolExecutor
# Create a thread pool with 5 worker threads
with ThreadPoolExecutor(max_workers=5) as executor:
    # The executor manages the worker threads for you
```

Tasks can be submitted to the executor for asynchronous execution using the `submit()` method. This method schedules the callable to be executed and returns a `Future` object representing the execution.

```
future = executor.submit(a_callable, arg1, arg2)
```

I recommend you to read some information about how scheduler of OS decides when and which task should be called. It's indeed interesting information, but I am too tired already to write about this :)

### 6.2 Future Objects

A `Future` object represents the result of an asynchronous computation. It provides methods to check whether the computation is complete, to wait for its result, and to retrieve the result once available.

### 6.3 Syntax

```
# Check if the task is done
if future.done():
    # Get the result of the computation
    result = future.result()
```

Let's put it all together in the following examples:

## Example

Simulating the execution of tasks:

```
import concurrent.futures
import time

def task(n):
    print(f"Executing task {n}")
    time.sleep(2)
    return f"Task {n} completed"

# Create a ThreadPoolExecutor
with concurrent.futures.ThreadPoolExecutor(max_workers=3) as executor:
    # Submit tasks to the executor
    futures = [executor.submit(task, n) for n in range(3)]

    # Retrieve and print the results
    for future in concurrent.futures.as_completed(futures):
        print(future.result())
```

## Output

```
Executing task 0
Executing task 1
Executing task 2
Task 0 completed
Task 2 completed
Task 1 completed
```

## Example

Consider a scenario where you have to process multiple files, such as reading them and performing some analysis or transformation.

```
from concurrent.futures import ThreadPoolExecutor
import os

def process_file(file_path):
    with open(file_path, 'r') as file:
        # Simulate some processing
        data = file.read()
        return f"Processed {os.path.basename(file_path)}: {len(data)} characters"

def main(file_paths):
    # Create a ThreadPoolExecutor
```

```

with ThreadPoolExecutor(max_workers=3) as executor:
    # Use map to concurrently process the files
    results = executor.map(process_file, file_paths)

    # Print the results
    for result in results:
        print(result)
# TODO Create a file test.py in the root directory, before running the code
if __name__ == "__main__":
    file_paths = [
        "test.py",
        "test.py",
        "test.py",
    ]
    main(file_paths)

```

## Output

```

Processed test.py: 790 characters
Processed test.py: 790 characters
Processed test.py: 790 characters

```

That's how IO-bound operations, such as file processing, within Python applications could be run achieving a significant performance improvement.

## 7. More Practice

For a comprehensive application, let's develop a simplified web server that handles incoming HTTP requests in parallel, logs each request asynchronously, and manages resources using threading techniques.

### 7.1 Application Overview

- **Web Server:** Accepts incoming requests and handles them in separate threads.
- **Logger:** Asynchronously logs request data using a producer-consumer pattern.
- **Resource Management:** Uses daemon threads for background tasks and ensures proper synchronization to prevent race conditions and deadlocks.

### 7.2 Implementation

#### Step 1: Setting Up the Web Server

```

import socket
from threading import Thread, Lock
import queue
import time
# Queue for log messages
log_queue = queue.Queue()
def handle_client(connection, address):

```

```

"""Handles client requests."""
try:
    request = connection.recv(1024).decode('utf-8')
    print(f"Received request from {address}: {request}")
    response = 'Hello World'
    connection.sendall(response.encode('utf-8'))

    # Log the request
    log_message = f"Handled request from {address}: {request}"
    log_queue.put(log_message)
finally:
    connection.close()
def start_server(host='127.0.0.1', port=8080):
    """Starts the web server."""
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as server_socket:
        server_socket.bind((host, port))
        server_socket.listen(5)
        print(f"Server listening on {host}:{port}")

        while True:
            conn, addr = server_socket.accept()
            client_thread = Thread(target=handle_client, args=(conn, addr))
            client_thread.start()

```

## Step 2: Logging

```

def process_logs():
    """Processes log messages asynchronously."""
    while True:
        log_message = log_queue.get()
        print(f"Logging: {log_message}")
        log_queue.task_done()
        time.sleep(0.1) # Simulate processing time

```

## Step 3: Resources

Use locks to synchronize access to shared resources if needed. In this example, we might not have explicit shared resources, but if our server interacts with a shared database or file system, locks can ensure data consistency.

Add functionality for where server interacts with a file or DB and use mutex mechanisms.

## Step 4: Run the server

```

if __name__ == "__main__":
    server_thread = Thread(target=start_server)
    server_thread.start()
    server_thread.join()

```



## Step 5: Daemon thread

The log processing thread is set as a daemon thread, ensuring it does not prevent the program from exiting. This is ideal for background tasks that should not block the main application from closing.

```
# Start the log processing thread
log_thread = Thread(target=process_logs, daemon=True)
log_thread.start()
```

## Step 6: Put it alltogether

```
import socket
from threading import Thread, Lock
import queue
import time

# Queue for log messages
log_queue = queue.Queue()

def handle_client(connection, address):
    """Handles client requests."""
    try:
        request = connection.recv(1024).decode('utf-8')
        print(f"Received request from {address}: {request.split('\r\n')[0]}")
        response = 'Hello World'
        connection.sendall(response.encode('utf-8'))

        # Log the request
        log_message = f"Handled request from {address}: {request.split('\r\n')[0]}"
        log_queue.put(log_message)
    finally:
        connection.close()

def start_server(host='127.0.0.1', port=8080):
    """Starts the web server."""
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as server_socket:
        server_socket.bind((host, port))
        server_socket.listen(5)
        print(f"Server listening on {host}:{port}")

        while True:
            conn, addr = server_socket.accept()
            client_thread = Thread(target=handle_client, args=(conn, addr))
            client_thread.start()

def process_logs():
    """Processes log messages asynchronously."""
    while True:
        log_message = log_queue.get()
        print(f"Logging: {log_message}")
        log_queue.task_done()
        time.sleep(0.1) # Simulate processing time

# Start the log processing thread
log_thread = Thread(target=process_logs, daemon=True)
log_thread.start()

if __name__ == "__main__":
```

```
server_thread = Thread(target=start_server)
server_thread.start()
server_thread.join()
```

You would need to open a terminal and send HTTP request to ensure that the server is working fine:

```
curl http://127.0.0.1:8080
```

## Output

```
Received request from ('127.0.0.1', 59910): G
Logging: Handled request from ('127.0.0.1', 59910): GET / HTTP/1.1
Host: 127.0.0.1:8080
User-Agent: curl/7.81.0
Accept: */*
```

Now once you have such a valuable tool as threading it is great for you to look out through your existing projects and try to speed them up!

Happy Pythoning!

## 8. Quiz

### Question 1:

What is a thread in the context of Python programming?

- A) A module for handling errors and exceptions.
- B) A sequence of instructions that can be executed independently within the process.
- C) A tool for managing database connections.
- D) A method for writing data to files.

### Question 2:

Which Python module provides support for threading?

- A) `sys`
- B) `os`
- C) `threading`
- D) `multiprocessing`

### Question 3:

How can threads be beneficial in a GUI application?

- A) They allow for multiple windows to be opened simultaneously.
- B) They make the code simpler and easier to debug.
- C) They help in maintaining responsiveness while performing background tasks.
- D) They decrease the memory usage of the application.

**Question 4:**

What method is used to start a thread in Python?

- A) `.start()`
- B) `.run()`
- C) `.execute()`
- D) `.launch()`

**Question 5:**

What does the `join()` method do in the context of threading?

- A) It combines the output of multiple threads into a single output stream.
- B) It forces a thread to stop executing.
- C) It ensures that a thread completes its execution before the main program continues.
- D) It merges two separate threads into one.

**Question 6:**

What is the purpose of a daemon thread in Python?

- A) To debug the application when an error occurs.
- B) To run in the background without preventing the main program from exiting.
- C) To handle all input/output operations automatically.
- D) To increase the priority of execution in multithreading.

**Question 7:**

Which is a common use case for implementing threading in network applications?

- A) Reducing the cost of server hardware.
- B) Performing multiple network or disk operations concurrently.
- C) Encrypting data transmitted over the network.
- D) Automatically updating the application to the latest version.

**Question 8:**

What potential issue can arise from improper handling of threads?

- A) Syntax errors in the code.
- B) The user interface becomes less attractive.
- C) Race conditions leading to unpredictable outcomes.
- D) Increased application licensing costs.

### Question 9:

How can the Global Interpreter Lock (GIL) affect multithreaded Python programs?

- A) It can increase the memory available to Python programs.
- B) It prevents multiple threads from executing Python bytecodes at once.
- C) It automatically optimizes the program for faster execution.
- D) It encrypts the Python bytecode for security.

### Question 10:

What is the `ThreadPoolExecutor` used for in Python?

- A) Managing a pool of worker threads to execute function calls asynchronously.
- B) Encrypting and securely storing passwords.
- C) Scheduling the execution of programs at specific times.
- D) Compressing files to save disk space.

## 9. Homework

**Objective:** Develop a multithreaded web crawler that not only fetches web pages from a list of URLs concurrently but also handles logging and error management using threading concepts.

### Requirements:

Add everything we have learnt during this lesson, take a look at this snippet and proceed with coding!

```
import threading
import requests
from bs4 import BeautifulSoup
from queue import Queue

class WebCrawler(threading.Thread):
    def __init__(self, url_queue, results_queue, shutdown_event):
        super().__init__()
        self.url_queue = url_queue
        self.results_queue = results_queue
        self.shutdown_event = shutdown_event

    def run(self):
        pass

if __name__ == "__main__":
    urls = []
    url_queue = Queue()
```

```
results_queue = Queue()  
shutdown_event = threading.Event()
```

# Lesson 29: Multiprocessing

"Turbo booster of Python, and nothing to say else"

Multiprocessing in Python is a powerful paradigm that allows developers to achieve true parallelism in their applications, particularly in CPU-bound tasks.

## 1. Multiprocessing vs Multithreading

Let's finally compare and get the detailed analysis of both mechanisms.

	Multiprocessing	Multithreading
Concurrency Model	Parallel execution of processes	Concurrent execution of threads within a single process
Memory Space	Separate memory space for each process	Shared memory space among threads
Use Cases	CPU-bound tasks requiring full CPU utilization	I/O-bound tasks, or when tasks need to share memory and resources
GIL Impact	Bypasses the GIL, allowing full use of multiple CPU cores	Limited by the GIL, not suitable for CPU-bound tasks
Resource Usage	Higher, due to separate memory space and process management overhead	Lower, threads are lighter than processes
Data Safety	Processes do not share memory, reducing data corruption risks	Shared memory can lead to data corruption if not properly managed

### 1.1 Use Cases

Multiprocessing shines in scenarios where tasks are *CPU-bound* and can be performed independently. Such as Data Processing and Computing, though sometimes may be good for Web as well.

Let's dive into the practice, but don't forget to refer to the table we saw in the first lesson while designing your application.

## 2. multiprocessing

### 2.1 Creating Processes

To create a new process, you instantiate a `Process` object and call its `start()` method. Each `Process` requires a target function to execute and can also accept arguments for the target function.

#### Example:

```
from multiprocessing import Process
def greet(name):
    print(f"Hello, {name}!")
if __name__ == '__main__':
    p = Process(target=greet, args=('World',))
    p.start()
    p.join()  # Wait for the process to finish
```

#### Output

```
Hello, World!
```

### 2.2 Process Management

The `Process` class represents an activity that is run in a separate process. The class has methods like `start()`, `join()`, and `terminate()` to manage the process lifecycle. You can also check if the process is still running using `is_alive()`.

#### Example

```
from multiprocessing import Process
import time
def worker():
    """Function to be executed by the process"""
    print("Worker process is running...")
    time.sleep(2)
    print("Worker process is finishing...")
if __name__ == "__main__":
    # Create a Process
    process = Process(target=worker)

    # Start the process
    process.start()
    print("Process started.")

    # Check if the process is alive
    print(f"Is process alive? {process.is_alive()}")

    # Wait for the process to complete
    process.join()
```

```

print("Process joined.")

# Check again if the process is alive
print(f"Is process alive after join? {process.is_alive()}")
# Try to terminate the process (has no effect if already finished)
process.terminate()
print("Process terminated.")

```

## Output

```

Process started.
Is process alive? True
Worker process is running...
Worker process is finishing...
Process joined.
Is process alive after join? False
Process terminated.

```

## 3. Inter-process Communication (IPC)

In multiprocessing, since each process operates in its own memory space, direct data sharing like in multithreading (with shared memory) is not possible. Python's multiprocessing module provides several ways to enable IPC. Pipes and Queues are the most common.

### 3.1 Pipes

A Pipe can be used for two-way communication between processes. Data in a pipe is buffered, which means it's held in a temporary storage area until the recipient retrieves it.

#### Example

```

from multiprocessing import Process, Pipe

def sender(pipe):
    """Function to send data through the pipe."""
    pipe.send(["Hello from the sender!"])
    pipe.close()

def receiver(pipe):
    """Function to receive data through the pipe."""
    print(f"Receiver got message: {pipe.recv()}")

if __name__ == "__main__":
    parent_conn, child_conn = Pipe()

    # Process for sending data
    p1 = Process(target=sender, args=(parent_conn,))

    # Process for receiving data
    p2 = Process(target=receiver, args=(child_conn,))

    p1.start()
    p2.start()

```



```
p1.join()
p2.join()
```

## Output

```
Receiver got message: ['Hello from the sender!']
```

## 3.2 Queues

Queues are thread and process-safe, making them ideal for IPC. They can be used to exchange data between processes.

### Example

```
from multiprocessing import Process, Queue
def writer(queue):
    """Function to write data to the queue."""
    queue.put("Hello from the writer!")
def reader(queue):
    """Function to read data from the queue."""
    print(f"Reader received: {queue.get()}")
if __name__ == "__main__":
    queue = Queue()

    # Process for writing to the queue
    p1 = Process(target=writer, args=(queue,))

    # Process for reading from the queue
    p2 = Process(target=reader, args=(queue,))

    p1.start()
    p2.start()

    p1.join()
    p2.join()
```

## Output

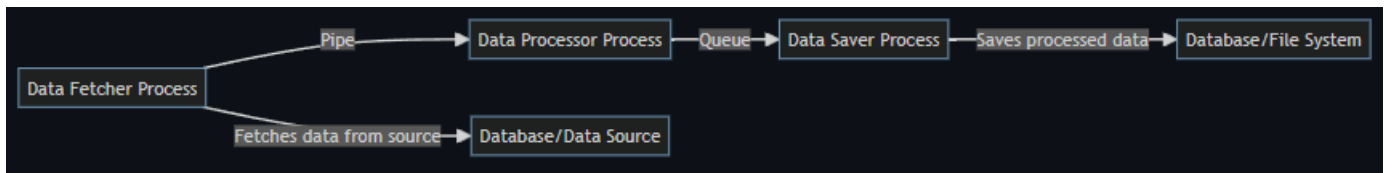
```
Reader received: Hello from the writer!
```

## Explanation

- **Pipes** are best for simple, unidirectional communication between two processes.
- **Queues** are more flexible and suitable for complex data exchange between multiple producers and consumers.

### 3.3 Real-world App

**Objective:** Create a data processing application that needs to perform several tasks: fetching data from a database, processing this data (e.g., filtering, aggregation), and finally saving the results to a new location. This scenario is common in data analytics and ETL (Extract, Transform, Load) operations.



Using multiprocessing we can create a streamlined pipeline that efficiently processes large datasets by dividing the work across multiple processes.

#### 3.3.1 Application Requirements

1. **Data Fetcher Process:** Connects to a database or data source, fetches data, and sends it through a Pipe to the next stage.
2. **Data Processor Process:** Receives raw data through a Pipe, performs processing (filtering, aggregating), and places processed data into a Queue.
3. **Data Saver Process:** Takes processed data from the Queue and saves it to a database, file system, or another storage system.

#### 3.3.2 Implementation

```
from multiprocessing import Process, Pipe, Queue
import time

def fetch_data(sender_conn):
    """Simulates fetching data and sends it to the processor."""
    for i in range(3): # Simulate fetching 3 data batches
        data = f"Data batch {i}"
        print(f"Fetching: {data}")
        sender_conn.send(data)
        time.sleep(1) # Simulate delay
    sender_conn.close()

def process_data(receiver_conn, data_queue):
    """Receives data, processes it, and puts it into a queue."""
    while True:
        data = receiver_conn.recv()
        processed_data = f"{data} - Processed"
        print(f"Processing: {processed_data}")
        data_queue.put(processed_data)

def save_data(data_queue):
    """Takes data from the queue and simulates saving it."""
    while True:
        data = data_queue.get()
        print(f"Saving: {data}")
        time.sleep(1) # Simulate save delay

if __name__ == "__main__":
    parent_conn, child_conn = Pipe()
```

```
data_queue = Queue()
# Creating processes
fetcher = Process(target=fetch_data, args=(parent_conn,))
processor = Process(target=process_data, args=(child_conn, data_queue))
saver = Process(target=save_data, args=(data_queue,))
# Starting processes
fetcher.start()
processor.start()
saver.start()
# Joining fetcher and processor (saver would ideally run indefinitely or
have a stop condition)
fetcher.join()
processor.join()
```

## Output

```
Fetching: Data batch 0

Processing: Data batch 0 - Processed

Saving: Data batch 0 - Processed

Fetching: Data batch 1

Processing: Data batch 1 - Processed

Saving: Data batch 1 - Processed

Fetching: Data batch 2

Processing: Data batch 2 - Processed

Saving: Data batch 2 - Processed
```

## Explanation

Basically, here, we created a 3 separate processes, which are responsible for different operations, we can check them in task manager on Windows or using the following command on Linux/ Mac OS.

```
ps -u $(whoami) | grep python
```

## Output

```
134616 pts/0    00:00:00 python3
134618 pts/0    00:00:00 python3
```

**NOTE:** In a real-world scenario, you would have mechanisms to gracefully stop the saver process (e.g., using an Event or a sentinel value in the Queue).

Doing processing in parallel is much faster than synchronous approach we used before. You can try this out yourself, with a same code provided. And compare execution time.

## 4. Synchronisation

Same as multithreading, multiprocessing module provides several synchronization primitives, such as Lock, Semaphore, Event, and Condition, which help prevent data corruption and ensure data consistency.

### 4.1 Lock

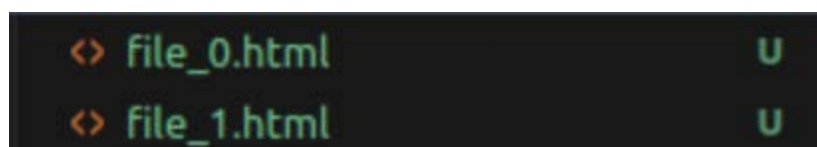
A Lock is a synchronization primitive that can be locked or unlocked. It is used to prevent simultaneous access to a shared resource by multiple processes, ensuring that only one process can access the resource at a time.

#### Example

Synchronize File Access for multiple processes

```
from multiprocessing import Process, Lock
def write_to_file(lock, file_path, data):
    with lock:
        with open(file_path, 'a') as f:
            f.write(data + '\n')
if __name__ == '__main__':
    lock = Lock()
    file_path = 'shared_file.txt'
    processes = [Process(target=write_to_file, args=(lock, file_path, f'Process
{i}') for i in range(5))]
    for p in processes:
        p.start()
    for p in processes:
        p.join()
```

#### Output



```
<> file_0.html      U
<> file_1.html      U
```

## 4.2 Semaphore

A Semaphore is a more general version of a Lock. While a Lock allows only one process to access a certain section of code at a time, a Semaphore allows a fixed number of processes to do so.

### Example

```
from multiprocessing import Process, Semaphore
import time
def access_database(semaphore, process_id):
    with semaphore:
        print(f"Process {process_id} is accessing the database")
        # Simulate database access
        time.sleep(1)
        print(f"Process {process_id} is done accessing the database")
if __name__ == '__main__':
    semaphore = Semaphore(2) # Allows up to 2 processes to access the critical
    section
    processes = [Process(target=access_database, args=(semaphore, i)) for i in
    range(3)]
    for p in processes:
        p.start()
    for p in processes:
        p.join()
```

### Output

```
Process 0 is accessing the database
Process 1 is accessing the database
Process 0 is done accessing the database
Process 2 is accessing the database
Process 1 is done accessing the database
Process 2 is done accessing the database
```

## 4.3 Event

An Event is a synchronization primitive that can be used to notify one or more processes that something has happened. Processes can wait for an event to be set before proceeding.

### Example

```
from multiprocessing import Process, Event
import time
def wait_for_event(event):
    print("Waiting for the event to be set...")
    event.wait()
    print("Event has been set, continuing with the task")
if __name__ == '__main__':
    event = Event()
    process = Process(target=wait_for_event, args=(event,))
    process.start()
```

```
# Simulate doing something
time.sleep(2)
event.set()
process.join()
```

## Output

```
Waiting for the event to be set...
```

```
Event has been set, continuing with the task
```

## 4.4 Condition

A Condition is a synchronization primitive that allows one process to wait for a condition to be met, while allowing other processes to notify it that the condition has been met.

### Example

We can model Producer-Consumer pattern, same what was described in a lesson about threading as well, add a Condition.

```
from multiprocessing import Process, Condition, Array
def producer(condition, shared_array):
    with condition:
        print("Producing items")
        shared_array[0] = 99 # Simulating item production
        condition.notify()
def consumer(condition, shared_array):
    with condition:
        condition.wait() # Wait for an item to be produced
        print(f"Consumed item: {shared_array[0]}")
if __name__ == '__main__':
    condition = Condition()
    shared_array = Array('i', [0]) # Shared array between processes
    p = Process(target=producer, args=(condition, shared_array))
    c = Process(target=consumer, args=(condition, shared_array))
    c.start()
    p.start()
    c.join()
    p.join()
```

These synchronization helping to avoid race conditions and ensure data consistency across concurrent processes.

Don't hesitate to use them, and again, be extremely careful with shared resources!

## 5. Process Pooling

### 5.1 Pool

The `Pool` class allows you to create a pool of worker processes that can execute tasks in parallel. It manages the available processes and assigns tasks to them as they become available.

#### Example:

```
from multiprocessing import Pool
def square(number):
    return number * number
if __name__ == '__main__':
    # Create a pool of 4 worker processes
    with Pool(processes=4) as pool:
        results = pool.map(square, range(10))
        print(results)
```

#### Output

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

### 5.2 Applying Pool

The `Pool` class provides several methods to parallelize tasks. The `map` and `apply` methods are particularly useful for distributing tasks among the pool's worker processes.

### 5.3 map

The `map` function applies a given function to each item of an iterable (e.g., list) and collects the results. It blocks until the result is ready.

```
results = pool.map(square, range(10))
print(results)
```

### 5.4 apply

The `apply` function applies a given function with arguments to a worker process. Unlike `map`, `apply` is used for a single invocation and blocks until the result is ready.

```
result = pool.apply(square, (10,))
print(result)
```

## 5.5 Worker Processes

The `Pool` class also provides ways to manage the worker processes, allowing for asynchronous task execution and callback handling.

### Example

```
def cube(number):  
    return number * number * number  
if __name__ == '__main__':  
    with Pool(processes=4) as pool:  
        async_result = pool.apply_async(cube, (3,))  
        # Do something else  
        print(async_result.get()) # Blocks until result is ready
```

### Output

27

This approach enables non-blocking task submission to the pool, allowing the main program to continue executing while the task is processed in the background.

## 6. Production Approaches

From what we have seen already, we working on a low level with processes, in real world cases, generally programmers prefer to use `Manager` and `ProcessPoolExecutor` classes to let the Python handle synchronisation and reduce potential bugs.

### 6.1 Manager

The `Manager` class in the `multiprocessing` module allows creating shared objects that can be accessed and modified by different processes.

This is especially useful for creating complex data structures shared across processes.

### Example

```
from multiprocessing import Manager, Process  
def modify_shared_list(shared_list):  
    shared_list.append("Hello")  
    shared_list.append("World")  
if __name__ == '__main__':  
    with Manager() as manager:  
        # Create a shared list  
        shared_list = manager.list()  
  
        # Create a process that modifies the shared list  
        p = Process(target=modify_shared_list, args=(shared_list,))  
        p.start()
```



```
p.join()

# Access the modified list
print(shared_list)
```

## Output

```
['Hello', 'World']
```

## 6.2 ProcessPoolExecutor

The `concurrent.futures` module provides a high-level interface for asynchronously executing callables with `ProcessPoolExecutor`.

It simplifies the management of process pools, offering an alternative to the `Pool` class for executing tasks in parallel.

### Example

```
from concurrent.futures import ProcessPoolExecutor
def power(base, exponent):
    return base ** exponent
if __name__ == '__main__':
    with ProcessPoolExecutor(max_workers=4) as executor:
        futures = [executor.submit(power, 2, exp) for exp in range(10)]
        for future in futures:
            print(future.result())
```

## Output

```
1
2
4
8
16
32
64
128
256
512
```

# 7. Practice

## 7.1 Images Converter

- **Goal:** Convert a set of color images to grayscale.
- **Approach:** Use multiprocessing to parallelize the processing, speeding up the operation on systems with multiple CPU cores.
- **Tools:** The `Pillow` library for image processing and Python's `multiprocessing` module for parallel execution.

### Step 1: Install Pillow

First, ensure you have the `Pillow` library installed, as it's used for image processing:

```
pip install Pillow
```

### Step 2: Image Processing Function

This function reads an image, converts it to grayscale, and saves it.

```
from PIL import Image
import os
def process_image(image_path):
    # Open the image file
    with Image.open(image_path) as img:
        # Convert the image to grayscale
        grayscale_img = img.convert("L")

        # Save the processed image
        base, ext = os.path.splitext(image_path)
        new_image_path = f"{base}_grayscale{ext}"
        grayscale_img.save(new_image_path)

    print(f"Processed {image_path} -> {new_image_path}")
```

### Step 3: Parallelize Image Processing

Use the `multiprocessing.Pool` class to process multiple images in parallel.

```
from multiprocessing import Pool
import glob
def main():
    # List of image files to process
    image_files = glob.glob('path/to/images/*.jpg') # Update path as needed

    # Create a pool of worker processes
    with Pool(processes=4) as pool:
        pool.map(process_image, image_files)
if __name__ == "__main__":
    main()
```

## Step 4: Put it alltogether

```
from PIL import Image
import os
from multiprocessing import Pool
import glob

def process_image(image_path):
    # Open the image file
    with Image.open(image_path) as img:
        # Convert the image to grayscale
        grayscale_img = img.convert("L")

        # Save the processed image
        base, ext = os.path.splitext(image_path)
        new_image_path = f"{base}_grayscale{ext}"
        grayscale_img.save(new_image_path)

    print(f"Processed {image_path} -> {new_image_path}")

def main():
    # List of image files to process
    image_files = glob.glob('Advanced/images/*.jpg')

    # Create a pool of worker processes
    with Pool(processes=4) as pool:
        pool.map(process_image, image_files)

if __name__ == "__main__":
    main()
```

## Output

```
Processed          Advanced/images/image1_grayscale.jpg      ->
Advanced/images/image1_grayscale_grayscale.jpg
Processed Advanced/images/image3.jpg -> Advanced/images/image3_grayscale.jpg
Processed Advanced/images/image2.jpg -> Advanced/images/image2_grayscale.jpg
Processed Advanced/images/image1.jpg -> Advanced/images/image1_grayscale.jpg
```

## Explanation:

- The program searches for JPEG images in the specified directory.
- It then creates a pool of worker processes (adjust the `processes` parameter based on your CPU).
- Each image is processed in parallel by converting it to grayscale and saving the result.

## 7.2 Data Processing Pipeline

- **Goal:** Filter and aggregate data from a large dataset in parallel.
- **Approach:** Use multiprocessing to distribute data chunks to different processes for filtering based on a condition, then aggregate the results.
- **Dataset:** For simplicity, we'll simulate a dataset using random numbers or a CSV file if you have one handy.

### Step 1: Prepare the Dataset

For demonstration purposes, let's create a function that generates a large dataset of random integers.

```
import random
def generate_dataset(size):
    return [random.randint(1, 100) for _ in range(size)]
```

## Step 2: Define the Filter and Aggregate Functions

Define a function to filter the dataset based on a condition (e.g., finding even numbers) and a function to aggregate the filtered results (e.g., calculating the sum).

```
def filter_data(data_chunk):
    return [num for num in data_chunk if num % 2 == 0]
def aggregate_data(filtered_data_list):
    return sum(filtered_data_list)
```

## Step 3: Distribute Tasks

Use the multiprocessing module to distribute the dataset chunks across multiple processes for filtering and then aggregate the results.

```
from multiprocessing import Pool
def process_data_chunk(data_chunk):
    """Process a chunk of data: filter and return the aggregation."""
    filtered_data = filter_data(data_chunk)
    return aggregate_data(filtered_data)
if __name__ == "__main__":
    dataset = generate_dataset(1000000) # Generate a dataset with 1,000,000
    items
    chunks = [dataset[i:i + 10000] for i in range(0, len(dataset), 10000)] #
    Divide into chunks of 10,000 items

    with Pool(processes=4) as pool:
        results = pool.map(process_data_chunk, chunks)

    total_aggregate = sum(results)
    print(f"Total aggregate of filtered data: {total_aggregate}")
```

## Step 4: Put it altogether

```
from multiprocessing import Pool
import random
def generate_dataset(size):
    return [random.randint(1, 100) for _ in range(size)]
def filter_data(data_chunk):
    """Filter data based on a condition. Here, we find even numbers."""
    return [num for num in data_chunk if num % 2 == 0]
def aggregate_data(filtered_data_list):
    """Aggregate the filtered data. Here, we calculate the sum."""
    return sum(filtered_data_list)
```

```
def process_data_chunk(data_chunk):
    """Process a chunk of data: filter and return the aggregation."""
    filtered_data = filter_data(data_chunk)
    return aggregate_data(filtered_data)

if __name__ == "__main__":
    dataset = generate_dataset(1000000)
    chunks = [dataset[i:i + 10000] for i in range(0, len(dataset), 10000)]

    with Pool(processes=4) as pool:
        results = pool.map(process_data_chunk, chunks)

    total_aggregate = sum(results)
    print(f"Total: {total_aggregate}")
```

## Output

```
Total: 25564084
```

Happy multiprocessing and multithreading! Find what suits best for your based on the table from the first lesson! And we step into `asyncio`!

## 8. Quiz

### Question 1:

What is the primary advantage of using multiprocessing over multithreading in Python?

- A) Multiprocessing can bypass the Global Interpreter Lock (GIL) and fully utilize multiple CPU cores.
- B) Multiprocessing uses less memory than multithreading.
- C) Multiprocessing is simpler to implement for CPU-bound tasks.
- D) Multiprocessing allows for easier data sharing between threads.

### Question 2:

Which Python module is most suitable for CPU-bound tasks?

- A) `threading`
- B) `multiprocessing`
- C) `asyncio`
- D) `subprocess`

### Question 3:

What is the purpose of using a `Pool` in the `multiprocessing` module?

- A) To manage a group of threads for performing I/O-bound tasks.
- B) To manage a group of processes for performing CPU-bound tasks in parallel.

- C) To lock resources shared among various threads.
- D) To provide a way to execute asynchronous tasks.

#### Question 4:

When using `multiprocessing`, how is data typically passed between processes?

- A) Through shared memory locations.
- B) Via global variables.
- C) Using inter-process communication mechanisms like `Pipes` and `Queues`.
- D) Data cannot be passed between processes; each process must access data independently.

#### Question 5:

What is true about Python's Global Interpreter Lock (GIL) with regard to multiprocessing?

- A) The GIL prevents the effective parallel execution of threads and is not a concern in multiprocessing.
- B) The GIL enhances the performance of multiprocessing by optimizing memory management.
- C) Multiprocessing requires the GIL to manage memory between processes efficiently.
- D) The GIL synchronizes the execution of multiple processes on multi-core systems.

#### Question 6:

Which of the following is NOT a valid way to manage process synchronization in Python's `multiprocessing` environment?

- A) Using a `Lock` to ensure that only one process accesses a critical section of code at a time.
- B) Employing a `Semaphore` to limit the number of processes that can access a particular resource.
- C) Utilizing a `Condition` variable to control workflow between processes.
- D) Implementing a `Goto` statement to control process flow.

#### Question 7:

In the context of `multiprocessing`, what is a `Manager` used for?

- A) To terminate processes that are not responding.
- B) To control access to a shared database.
- C) To create shared data objects that can be modified by multiple processes.
- D) To monitor the CPU usage of each process.

These questions should help solidify your grasp of multiprocessing concepts and encourage you to think about how they apply to real-world programming challenges.

## 9. Homework

**Objective:** Develop a Python script that uses multiprocessing to search for a keyword in a collection of text files simultaneously.

## Requirements

- Before developing, prepare a directory with several large text files.
- Scan the directory for text files.
- Each process should search one text file for a given keyword and record the lines where the keyword appears.
- Collect and aggregate results from all processes to show which files contain the keyword and the line numbers.
- Use `Pool` to distribute the search tasks across multiple processes.
- Ensure synchronization when aggregating results.

Additionally, you can write it in a synchronous manner and compare the time execution, you will be impressed!

# Lesson 30: Asyncio

"The conductor of asynchronous symphonies, and bugs xD"

## 1. Introduction

Asynchronous programming is a concurrency model that allows certain operations, especially I/O-bound tasks, to run without blocking the execution of your program.

It's about doing other work while waiting for an I/O operation to complete, without the need for multi-threading or multi-processing.

### 1.1 Definition

**Non-blocking Execution:** Functions that perform lengthy operations (like network or file I/O) return immediately, allowing the program to continue running.

**Event Loop:** A programming construct that waits for and dispatches events or messages in a program. It facilitates the management of asynchronous tasks.

Let's compare both approaches:

Aspect	Synchronous	Asynchronous
Execution Flow	Sequential. Each task must complete before the next begins.	Tasks run independently, allowing the execution of other tasks in the meantime.
Resource Utilization	Can lead to poor resource utilization during I/O operations, as the program waits for the operation to complete.	Improves resource utilization by freeing up the program to perform other tasks during I/O operations.

## 2. Coroutines

Coroutines allow you to write code that looks sequential but actually executes asynchronously, pausing and resuming at specific points.

It is more about cooperation between routines – waiting for and yielding control to other routines.

Initially, Python used generators (Python < 3.5) to yield values and execute asynchronously. They were a stepping stone towards full asynchronous support.



## 2.2 `async` and `await`

**`async`:** Declares a function as a coroutine. An `async` function can contain `await` expressions, and it doesn't run immediately. Instead, it returns an awaitable object.

### Example

```
async def fetch_data():
    pass
print(type(fetch_data()))
```

### Output

```
<class 'coroutine'>
```

**Note:** This should raise the `RuntimeWarning: coroutine fetch_data was never awaited`.

**`await`:** Pauses the execution of the enclosing coroutine, waiting for an awaitable object (like another coroutine) to complete. This pause allows other tasks to run while waiting, making it non-blocking.

### Example

```
async def main():
    await fetch_data()
```

The use of `async` and `await` makes asynchronous code look and behave more like traditional synchronous code, though it's a concurrent execution.

## 3. Event Loop

Consider Event Loop like the conductor of an orchestra.

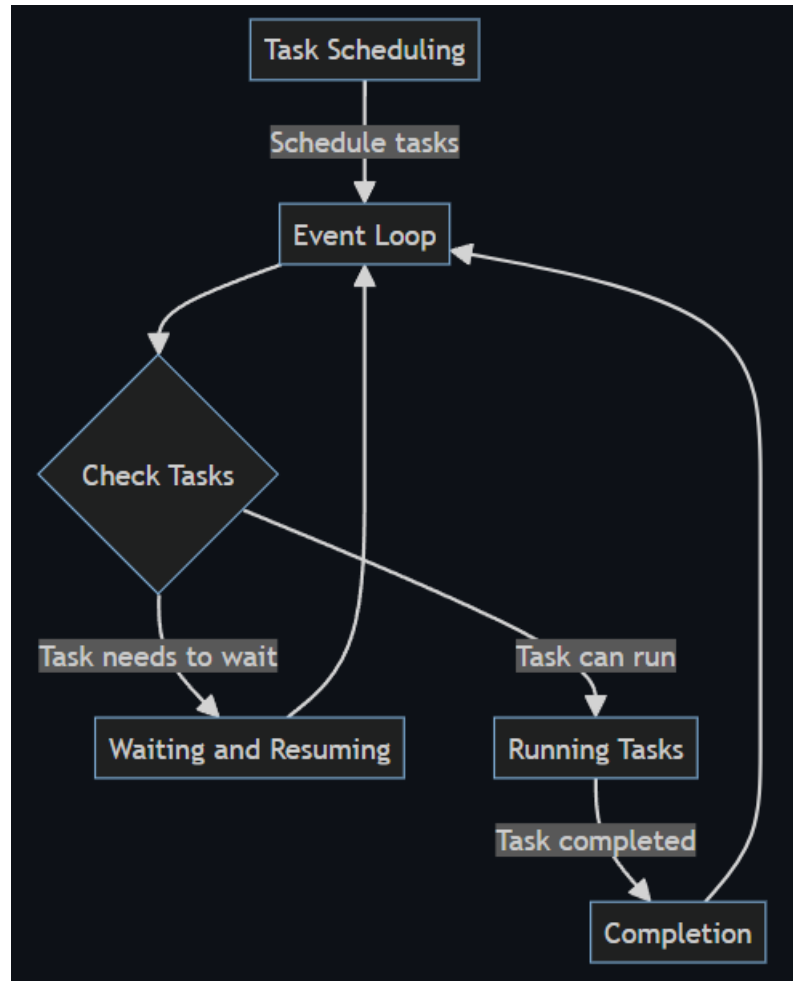
It keeps track of all the tasks that need to run, starts them at the right moment, and manages their execution until they're done.

### 3.1 How it works?

Python's `asyncio` library brings this concept into your programs. It runs an event loop that efficiently manages all your asynchronous tasks.

1. **Task Scheduling:** You tell the event loop about all the tasks (coroutines) you want to run by scheduling them.
2. **Running Tasks:** The event loop starts running the tasks. If a task needs to wait (say, for a file to download), it pauses that task and moves on to the next one.

3. **Waiting and Resuming:** Once the waiting is over (the file is downloaded), the task is resumed right where it left off.
4. **Completion:** This process continues until all tasks are done.



**NOTE:** You usually don't need to create or manage the event loop yourself, `asyncio` provides a high-level API for running asynchronous tasks.

### Example

```
import asyncio
async def main():
    print("Hello")
    await asyncio.sleep(1) # Simulate an I/O operation
    print("world")
# Running the main coroutine with asyncio
asyncio.run(main())
```

### Output

```
Hello

world
```

## Explanation

In this example, `asyncio.run(main())` is your entry point. It starts up the event loop, schedules your `main()` coroutine for execution, and keeps the program running until all tasks are completed.

## 4. Practice

Suppose we need to make an HTTP Request to the server. We would need to use the `aiohttp` library for making asynchronous requests

### Example

**Note:** Install `aiohttp` into `venv` before running `pip install aiohttp`

```
import aiohttp
import asyncio
async def fetch_page(url):
    async with aiohttp.ClientSession() as session:
        async with session.get(url) as response:
            return await response.text()
async def main():
    html = await fetch_page('https://google.com')
    print(html[:100])
asyncio.run(main())
```

### Output

```
<!doctype    html><html    itemscope=""    itemtype="http://schema.org/WebPage"
lang="en-GB"><head><meta cont
```

## 4.2 Async File IO

For file IO, `asyncio` offers a different set of APIs, since disk operations can also block the event loop.

### Example

**Objective:** We need to process a file very fast, using asynchronous approach

**Note:** Install `aiofiles` into `venv`: `pip install aiofiles` and create `example.txt` file before running.

```
import aiofiles
import asyncio
async def read_file(filename):
    async with aiofiles.open(filename, mode='r') as f:
        contents = await f.read()
        print(contents)
asyncio.run(read_file('example.txt'))
```

## Output

```
test
```

```
test
```

Same approach and logic is applied to any I/O bound operation.

## 4.3 Multiple Async I/O Operations

`asyncio.gather` and `asyncio.wait` are two powerful functions for handling multiple asynchronous operations concurrently.

### Example

**Objective:** We want to have a couple of different I/O operations, we would use `asyncio.gather` to create several tasks and run them asynchronously.

```
import asyncio
async def task(number):
    print(f'Starting task {number}')
    await asyncio.sleep(1)
    print(f'Finished task {number}')
    return number
async def main():
    results = await asyncio.gather(task(1), task(2), task(3))
    print(f'Task results: {results}')
asyncio.run(main())
```

## Output

```
Starting task 1
Starting task 2
Starting task 3
Finished task 1
Finished task 2
Finished task 3
Task results: [1, 2, 3]
```

In conclusion, `asyncio` is all about managing asynchronous IO operations simpler and more efficient, much better than threading, and simpler to understand.

## 5. Scheduling Tasks

### 5.1 `asyncio.create_task`

The `asyncio.create_task()` function is used to schedule the execution of a coroutine: it wraps the coroutine into a Task and schedules its execution.

The coroutine itself runs concurrently with other tasks and operations and doesn't block the code itself.

#### Example

```
import asyncio
async def my_coroutine():
    print('My Coroutine')
    await asyncio.sleep(1)
    return 'Coroutine Finished'
async def main():
    # Schedule the coroutine to run as an asyncio Task
    task = asyncio.create_task(my_coroutine())

    # Do other stuff in the meantime
    print('Doing Other Stuff')

    # Wait until the task completes
    result = await task
    print(result)
asyncio.run(main())
```

#### Output

```
Doing Other Stuff
My Coroutine
Coroutine Finished
```

#### Explanation

In this example, `my_coroutine` is scheduled to run as a task, allowing the main function to proceed with "Doing Other Stuff" before waiting for `my_coroutine` to finish.

### 5.2 Task Scheduling and Execution Order

**IMPORTANT:** If a task awaits another operation, the event loop can switch to running another task, effectively using concurrency, processing different operation.

#### Example

Tasks are executed in the order they are scheduled, considering their await expressions.

```
import asyncio
async def first_task():
    print('First Task Start')
    await asyncio.sleep(2)
    print('First Task End')
async def second_task():
    print('Second Task Start')
    await asyncio.sleep(1)
    print('Second Task End')
async def main():
    asyncio.create_task(first_task())
    asyncio.create_task(second_task())
    # Wait a bit for all tasks to finish
    await asyncio.sleep(3)
asyncio.run(main())
```

## Output

```
First Task Start
Second Task Start
Second Task End
First Task End
```

## Example

This example demonstrates that `second_task` can complete before `first_task` despite being scheduled after it, thanks to the asynchronous sleep.

To be honest, that's pretty much everything I wanted to talk in this lesson. Now it's high time for practice!

I would recommend to take a deeper look as well at Async Streaming Patterns, Custom Async Context Managers and Async Iterators for a better understanding of more features in `asyncio`, they are too wide to be covered in this book.

And of course refer to their [official documentation](#).

# 6. Project and Debugging

## 6.1 Web Scraper

**Objective:** Create a simple web scraper that fetches content from multiple URLs concurrently and saves the data to files.

**Tools:** `aiohttp` for asynchronous HTTP requests, `aiofiles` for asynchronous file operations.

**Step 1:** Install `aiohttp` and `aiofiles` using `pip`.

```
pip install aiohttp aiofiles
```

**Step 2:** Use `aiohttp` to make concurrent GET requests to a list of URLs. and write the response content to files using `aiofiles`.

**Step 3:** Put it alltogether.

### Example

```
import aiohttp
import aiofiles
import asyncio

async def fetch_url(session, url):
    async with session.get(url) as response:
        content = await response.text()
    return content

async def save_content(filename, content):
    async with aiofiles.open(filename, 'w') as file:
        await file.write(content)

async def main(urls):
    async with aiohttp.ClientSession() as session:
        tasks = [fetch_url(session, url) for url in urls]
        contents = await asyncio.gather(*tasks)

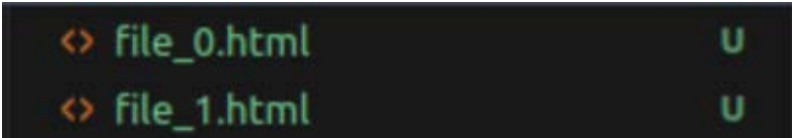
        save_tasks = [save_content(f'file_{i}.html', content)
                        for i, content in enumerate(contents)]
        await asyncio.gather(*save_tasks)

urls = [
    'https://google.com',
    'https://uk.yahoo.com/',
]

asyncio.run(main(urls))
```

### Output

You will see that 2 files are created with websites content in the root directory.



```
<> file_0.html      U
<> file_1.html      U
```

Now let's take a closer look on debugging of the async code

## 6.2 pdb

### Example

```
import pdb
async def main():
    pdb.set_trace()
```

```
result = await some_coroutine()
print(result)
```

## Output

It will stop the execution of the code and can be very helpful for debugging purposes, where you as a programmer will be able to check each object itself.

```
lessons/test.py(35)main()

-> result = await fetch_url()

(Pdb) result
```

## 6.3 debug=True

### Example

```
import logging
import asyncio
logging.basicConfig(level=logging.DEBUG)
asyncio.run(main(), debug=True)
```

## Output

```
DEBUG:asyncio:Using selector: EpollSelector
DEBUG:asyncio:Get address info google.com:443, type=<SocketKind.SOCK_STREAM: 1>, flags=<AddressInfo.AI_ADDRCONFIG: 32>
DEBUG:asyncio:Get address info uk.yahoo.com:443, type=<SocketKind.SOCK_STREAM: 1>, flags=<AddressInfo.AI_ADDRCONFIG: 32>
```

## 7. Quiz

### Question 1:

What is the primary purpose of asynchronous programming?

- A) To increase the speed of CPU-bound operations.
- B) To manage I/O-bound and network operations efficiently without blocking code execution.
- C) To simplify complex algorithms.
- D) To enhance data processing capabilities of multi-core processors.

### Question 2:



What does the `async` keyword signify in a Python function?

- A) It pauses the execution of the function.
- B) It declares the function as a coroutine.
- C) It immediately executes the function.
- D) It makes the function execute multiple times.

### Question 3:

Which statement about the event loop in `asyncio` is true?

- A) The event loop can only run one task at a time.
- B) It blocks the main program until all tasks are completed.
- C) It manages the execution of multiple tasks by pausing and resuming them as needed.
- D) It runs synchronously with other Python threads.

### Question 4:

In Python's `asyncio` library, how do you correctly handle multiple asynchronous tasks concurrently?

- A) Using multiple threads.
- B) By nesting asynchronous functions.
- C) Using `asyncio.gather` to run them.
- D) By calling them sequentially.

## 8. Homework

**Objective:** Build an asynchronous application that makes multiple API calls, aggregates the data, and computes some statistics.

### Requirements:

1. Use `aiohttp` for making API calls concurrently.
2. Make at least three different API calls to a public API (e.g., JSONPlaceholder or any other public API).
3. Aggregate the results and calculate the average or any other statistic of the fetched data.

### Example

```
import aiohttp
import asyncio

async def fetch_api_data(session, url):
    pass

async def main():
    urls = []

asyncio.run(main())
```

**Notes:** You need to install `aiohttp`

```
pip install aiohttp
```

Thanks for completing the course, I really appreciate your attention and hope that my input was useful.

Good luck in future endeavors!

# Quiz Answers

## Lesson 1

1-B  
2-B  
3-B  
4-C  
5-C

## Lesson 2 / 3

1-B  
2-A  
3-B  
4-A  
5-B  
6-Error  
7-D  
8-A

## Lesson 4

1-B  
2-C  
3-A  
4-C  
5-B  
6-B  
7-A  
8-B

## Lesson 5

1-None  
2-B  
3- 010  
011  
012  
4-B  
5-A

## Lesson 6

1-40  
2-[1, 2, 3, 4, 5, 6]  
3-D  
4-D  
5-B  
6-A  
7-C  
8-D  
9-D  
10-B

## Lesson 7

1-40  
2-Error  
3-B,C  
4-A  
5-B  
6-B  
7-B  
8-C

## Lesson 8

1-C  
2-B  
3-A  
4-C  
5-B  
6-B  
7-A  
8-B

## Lesson 9

1-B  
2-B,D  
3-C  
4-B  
5-B

## Lesson 10

1-A  
2-A  
3-C  
4-B  
5-B  
6-D  
7-A  
8-B  
9-C  
10-A  
11-C

## Lesson 11

1-A  
2-B  
3-B  
4-C  
5-A  
6-B

## Lesson 13

1-B  
2-B  
3-C  
4-C  
5-A

## Lesson 14

1-B  
2-B  
3-C  
4-A  
5-B  
6-C  
7-B  
8-A  
9-B  
10-A  
11-B  
12-B

## Lesson 15

1-B  
2-B  
3-B  
4-B  
5-A  
6-B  
7-B  
8-B  
9-A  
10-C  
11-D  
12-B  
13-C  
14-B

## Lesson 16

1-C  
2-A  
3-A  
4-D  
5-B  
6-D  
7-B  
8-B  
9-B  
10-A,C

## Lesson 17

1-A  
2-D  
3-B  
4-C  
5-D  
6-C  
7-B

## Lesson 20

1-A  
2-B  
3-B  
4-C  
5-C

## Lesson 25

1-B  
2-B  
3-C  
4-C  
5-C

## Lesson 26

1-B  
2-B  
3-B  
4-B  
5-C  
6-B,D

## Lesson 28

1-B  
2-C  
3-C  
4-A  
5-C  
6-B  
7-B  
8-C  
9-B  
10-A

## Lesson 29

1-A  
2-B  
3-B  
4-C  
5-A  
6-D  
7-C

## Lesson 30

1-B  
2-B  
3-C  
4-C

