PROGRAMMING PUZZLES

PYTHON EDITION

MATTHEW WHITESIDE

PROGRAMMING PUZZLES
Python Edition

CONTENTS

GETTING STARTED

Welcome to the Programming Puzzles: Python Edition! Before you dive into the world of Python puzzles, let's cover a few essential points to ensure you have a smooth and enjoyable experience.

INTRODUCTION

The puzzles in this book have specifically been designed so that people of all levels can attempt them. You'll find some puzzles for people who are earlier along in their Python journey, and you'll find harder puzzles for people who are more experienced.

The book is broken up into two main sections, challenge puzzles and fun puzzles. The challenge puzzles section contains 50 (+ a few bonus) puzzles that start off at beginner level and get harder as you progress. This is the main section of the book and it's recommended you start on puzzle #1 and work your way through.

The fun puzzles section focuses more around being creative and making use of multiple libraries that Python offers. This section is recommended if you want a break from the challenge puzzles, although they still aren't easy!

Note: it's recommended you know at least the basics of Python before starting any puzzles from this book. Familiarity with fundamental concepts such as variables, conditionals, loops, and functions will greatly benefit you in solving the puzzles.

A note on formatting…

This version of the book has been formatted specifically for kindle. Each section is designed so it's as easy as possible to work with and complete the puzzles e.g. the code won't split over two pages.

Due to this formatting some of the pages might look a bit "empty", but this is intended. Splits half way through puzzles are just annoying!

ENVIRONMENT SETUP

It’s important to get a working environment setup before you begin the puzzles - otherwise you won’t be able to test your solutions! Luckily Python is pretty straightforward to get working.

Python Installation

Python installation will be different for everyone as it depends on your operating system, however as a general guide you can use the following steps:

1. Visit the official Python website at python.org
2. Navigate to the "Downloads" section.
3. Choose the appropriate installer for your operating system (Windows, macOS, or Linux) and **select a version of python that is 3.10 or greater**.
4. Download the installer and follow the instructions to complete the installation. If the installer asks if you’d like to install the pip package manager say yes.

After the installation is complete, you can open a command prompt or terminal and type *python --version* to verify that Python is installed correctly. It should display the version number you installed. If the version number isn’t correct you can try *python3 --version* or *python3.10 --version*.

Code Editor

You will also need a code editor to edit and execute python code, there are many options out there but a few of the mainstream ones are:

- IDLE - comes bundled with Python.
- Visual Studio Code - code.visualstudio.com
- PyCharm - jetbrains.com/pycharm

In the scope of this book any of these editors will do the job, however as you continue your Python journey it's beneficial to move towards a more powerful IDE such as Visual Studio Code or PyCharm.

External Python Libraries

A small number of puzzles make use of external Python libraries. These can be installed using Python's package manager, pip.

You can check if you have pip by executing *python -m pip --version* in your command prompt / terminal. You should use the same python prefix as you did above, so if *python3 --version* worked for you above then you'd do *python3 -m pip --version* to check your pip version.

You can read more about pip at docs.python.org/3/installing/index.html

The external modules we use for the puzzles in this book can be installed by executing the following pip commands in your command prompt / terminal:

- *python -m pip install pygame*
- *python -m pip install PythonTurtle*

Git (optional)

This book comes with a provided git repository that contains starter code and solutions for each puzzle. This can be downloaded directly from github however the proper way is to clone it using a tool called git. If you're familiar with git, you can clone the repository as follows and then skip this section:

- *HTTPS: git clone https://github.com/MatWhiteside/python-puzzle-book.git*
- *SSH: git clone git@github.com:MatWhiteside/python-puzzle-book.git*

If you're unfamiliar with git you'll first need to install it on your computer, I won't cover it here as it can be different depending on your operating system but there are many guides that you'll be able to follow online.

If you can’t get git to work it’s not a big problem. Simply go to the github link in a browser and download the code as a zip file to your computer. You can then extract the file to a directory of your choosing and you’re ready to go. Link: github.com/MatWhiteside/python-puzzle-book

BIG O NOTATION

There are a number of harder puzzles in this book that make reference to the time or space complexity of a problem. An example requirement could read like “ensure your solution has a maximum time complexity of O(n)” - but what does this mean?

The Big O notation is often used in identifying how complex a problem is and defines the worst case complexity for a particular piece of code. The Big O notation is written as: O(...).

Examples

O(1), also known as constant, indicates that the given code will always take the exact same amount of time regardless of the input. A constant function could look like:

```
def add_five(input_num):
    return input_num + 5
```

If the input_num = 0 or the input_num = 99999 the code still only has to execute one line to carry out the addition. Therefore, we have a constant time complexity.

O(n), also known as linear, indicates that the complexity increases with our input size. Our input is represented by the variable n. Let’s take a look at what a linear function could look like:

```
def print_list(input_list):
    for item in input_list:
        print(item)
```

If the input_list has a length of 5, print(item) will execute 5 times. If the input_list has a length of 9999, print(item) will execute 9999 times. Therefore, we have a linear time complexity.

O(n^2), also known as quadratic, indicates that the time complexity increases by our input squared as our input size increases. Let’s take a look:

```
def print_list_nested(input_list):
    for item in input_list:
        for inner_item in input_list:
            print(item, " + ", inner_item)
```

If the input_list has a length of 5, print(item, " + ", inner_item) will execute 25 times. If the input_list has a length of 1000, print(item, " + ", inner_item) will execute 1,000,000 times.

If you're confused, don't worry. There aren't mentions of the Big O notation until later in the book, and when you get there if you still need help you'll be able to find lots of resources online. Just google something along the lines of "Big O notation explained".

FINAL POINTS BEFORE STARTING…

Hopefully you're now fully set up and ready to start solving puzzles! There are just a few final points to note before you get going…

1. It doesn't matter if the solutions to your puzzles break for bad inputs. We're not writing production code here, if a puzzle states that the input will be a list of integers then assume that you will receive a list of integers. Do however think about edge cases e.g. what if the input list is empty? What if the input number is negative?

2. All of the challenge puzzles have starter code that's designed to get you going quickly. Most of them have just one function defined, however that doesn't mean you can't define more functions yourself. It's only a guide, you can even choose to not use the starter code at all; it's up to you!

3. A number of puzzles have caveats defined, these must be met to successfully complete the puzzle. If you're struggling, attempt the puzzle without the caveats met and then work out how you could improve your solution to meet them after.

4. Any code that is provided makes use of typing in the function definitions to make it obvious what the inputs and output of the function are meant to be. If you're not familiar with typing in python, don't worry it doesn't change anything from your side.

   In fact, if you really don't like the typing, feel free to delete it from the function definitions and carry on as usual!

Example with typing:

```
def filter_strings_containing_a(input_strs: list[str])
          -> list[str]:
```

Example with typing removed:

```
def filter_strings_containing_a(input_strs):
```

CHALLENGE PUZZLES

Welcome to the first section of the book, challenge puzzles. These puzzles are designed to start off easy and increase in difficulty as you work your way through.

Each puzzle will describe a task and ask you to implement a function to solve the task. To allow you to focus on the important part there is also starter code provided so you can get straight into coding!

Should you get stuck on any puzzle there is a hints section further down in the book, I highly recommend you use that section before looking at any solutions. The best way to progress is to struggle on a puzzle and then solve it by yourself.

Important: don’t import any libraries unless specifically told in the puzzle!

PUZZLE 1

Task

Define a function *filter_strings_containing_a* that takes one parameter:

Name	Type	Example Input
input_strs	list of str	["apple", "banana", "cherry", "date"]

When called, the function should return a new list containing only strings that contain the letter “a”.

Starter Code

```
def filter_strings_containing_a(input_strs: list[str])
    -> list[str]:
    # Your implementation here

print(filter_strings_containing_a(
    ["apple", "banana", "cherry", "date"])
)
```

Examples

```
Input:  ["apple", "banana", "cherry", "date"]
Output: ["apple", "banana", "date"]

Input:  []
Output: []

Input:  ["bbbb", "cccc"]
Output: []
```

PUZZLE 2

Task

Define a function *sum_if_less_than_fifty* that takes two parameters:

Name	Type	Example Input
num_one	int	20
num_two	int	25

When called, the function should return either:

- The sum of the two numbers if the sum is less than 50
- *None* if the sum of the two numbers is more than or equal to 50

Starter Code

```
def sum_if_less_than_fifty(num_one: int, num_two: int)
    -> int | None:
    # Your implementation here

print(sum_if_less_than_fifty(20, 20))
```

Examples

```
Inputs:
 - num_one = 20
 - num_two = 20
Output: 40
Inputs:
 - num_one = 20
 - num_two = 30
Output: None
Inputs:
 - num_one = 20
 - num_two = 100
Output: None
```

PUZZLE 3

Task

Define a function *sum_even* that takes one parameter:

Name	Type	Example Input
input_nums	list of int	[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

When called, the function should return the sum of even integers in the list.

Starter Code

```
def sum_even(input_nums: list[int]) -> int:
    # Your implementation here

print(sum_even([1, 2, 3, 4, 5, 6, 7, 8, 9, 10]))
```

Examples

```
Input: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
Output: 30

Input: [10, 20, 30, 40, 50]
Output: 150

Input: [9, 7, 5, 3, 1]
Output: 0
```

PUZZLE 4

Task

Define a function *remove_vowels* that takes one parameter:

Name	Type	Example Input
input_str	str	"Hello, World!"

When called, the function should return a new string with all the vowels removed.

Starter Code

```
def remove_vowels(input_str: str) -> str:
    # Your implementation here

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

Examples

```
Input: "Hello, World!"
Output: "Hll, Wrld!"

Input: "aeiouAEIOU"
Output: ""

Input: "zzxxxccvvvbbnnmmmLLKKJJHH"
Output: "zzxxxccvvvbbnnmmmLLKKJJHH"
```

PUZZLE 5

Task

Define a function *get_longest_string* that takes one parameter:

Name	Type	Example Input
input_strs	list of str	["cat", "dog", "bird", "lizard"]

When called, the function should return the longest string in the list. If there are ties, return the string that appears first in the list.

Starter Code

```
def get_longest_string(input_strs: list[str]) -> str:
    # Your implementation here

print(get_longest_string(["cat", "dog", "bird", "lizard"]))
```

Examples

```
Input: ["cat", "dog", "bird", "lizard"]
Output: "lizard"

Input: ["cat", "dog", "bird", "wolf"]
Output: "bird"

Input: ["a", "b", "c", "d"]
Output: "a"
```

PUZZLE 6

Task

Define a function *filter_even_length_strings* that takes one parameter:

Name	Type	Example Input
input_strs	list of str	["cat", "dog", "fish", "elephant"]

When called, the function should return a new list with all the strings that have an even number of characters.

Starter Code

```
def filter_even_length_strings(input_strs: list[str])
    -> list[str]:
    # Your implementation here

print(filter_even_length_strings(
    ["cat", "dog", "fish", "elephant"]
))
```

Examples

```
Input: ["cat", "dog", "fish", "elephant"]
Output: ["fish", "elephant"]

Input: ["q", "w", "e", "r", "t", "y"]
Output: []

Input: ["qq", "ww", "ee", "rr", "tt", "yy"]
Output: ["qq", "ww", "ee", "rr", "tt", "yy"]
```

PUZZLE 7

Task

Define a function *reverse_elements* that takes one parameter:

Name	Type	Example Input
input_nums	list of int	[1, 2, 3, 4, 5]

When called, the function should return a new list with all of the elements in the original list reversed.

Starter Code

```
def reverse_elements(input_nums: list[int]) -> list[int]:
    # Your implementation here

print(reverse_elements([1, 2, 3, 4, 5]))
```

Examples

```
Input: [1, 2, 3, 4, 5]
Output: [5, 4, 3, 2, 1]

Input: []
Output: []

Input: [20, 15, 25, 10, 30, 5, 0]
Output: [0, 5, 30, 10, 25, 15, 20]
```

PUZZLE 8

Task

Define a function *filter_type_str* that takes one parameter:

Name	Type	Example Input
input_list	list of str or int	["hello", 1, 2, "www"]

When called, the function should return a new list containing only the strings from the original list.

Starter Code

```
def filter_type_str(input_list: list[str | int]) -> list[str]:
    # Your implementation here

print(filter_type_str(["hello", 1, 2, "www"]))
```

Examples

```
Input: ["hello", 1, 2, "www"]
Output: ["hello", "www"]

Input: []
Output: []

Input: [1, 2, 3, 4, 5]
Output: []
```

PUZZLE 9

Task

Define a function *string_to_morse_code* that takes one parameter:

Name	Type	Example Input
input_str	str	"HELLO, WORLD!"

When called, the function should return the more code equivalent of the input string. The function should meet the following requirements:

- A morse code “dot” should be represented by a full stop
- A more code “dash” should be represented by a hyphen
- A space should be used between each morse code letter e.g. “.-**(space)**-...”
- The function should be able to support the following characters in the input string:
    - Alphanumeric characters, uppercase and lowercase
    - Special characters: , . : ? ‘ - / ( ) “ @ = + !
- Should a space be encountered in the input string, it should be represented as a forward slash in the output string

Starter Code

```
def string_to_morse_code(input_str: str) -> str:
    morse_dict = {"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": "--..",  "0": "-----",
                  "1": ".----", "2": "..---", "3": "...--",
                  "4": "....-", "5": ".....", "6": "-....",
                  "7": "--...", "8": "---..", "9": "----.",
                  ",": "--..--",".": ".-.-.-",":": "---...",
                  "?": "..--..","'": ".----.","-": "-....-",
                  "/": "-..-.", "(": "-.--.", ")": "-.--.-",
                  '"': ".-..-.","@": ".--.-.","=": "-...-",
                  "+": ".-.-.", "!": "-.-.--"}

    # Your implementation here


print(string_to_morse_code("HELLO, WORLD!"))
```

Examples

```
Input: "HELLO, WORLD!"
Output: ".... . .-.. .-.. --- --..-- / .-- --- .-. .-.. -.. -.-.--"

Input: "abcdefghijklmnopqrstuvwxyz,.:?'-/()\"@=+!"
Output: ".- -... -.-. -.. . ..-. --. .... .. .--- -.- .-.. -- -. --
- .--. --.- .-. ... - ..- ...- .-- -..- -.-- --.. --..-- .-.-.- ---
... ..--.. .----. -....- -..-. -.--. -.--.- .-..-. .--.-. -...- .-
.-. -.-.--"

Input: ""
Output: ""
```

PUZZLE 10

Task

Define a function *get_second_largest_number* that takes one parameter:

Name	Type	Example Input
input_nums	list of int	[1, 2, 3, 4, 5]

When called, the function should return the second largest number in the list. If there is no second largest number, the function should return *None*.

Starter Code

```
def get_second_largest_number(input_nums: list[int])
    -> int | None:
    # Your implementation here

print(get_second_largest_number([1, 2, 3, 4, 5]))
```

Examples

```
Input: [1, 2, 3, 4, 5]
Output: 4

Input: [3, 45, 345, 435, 345, 43, 56, 34, 234, 34]
Output: 345

Input: [1]
Output: None
```

PUZZLE 11

Task

Define a function *format_number_with_commas* that takes one parameter:

Name	Type	Example Input
input_num	int	1000000

When called, the function should return a string representation of the number with commas as thousand separators.

Starter Code

```
def format_number_with_commas(input_num: int) -> str:
    # Your implementation here

print(format_number_with_commas(1000000))
```

Examples

```
Input: 1000000
Output: "1,000,000"

Input: 12345
Output: "12,345"

Input: -99999999
Output: "-99,999,999"
```

PUZZLE 12

Background

ASCII is a standard data-encoding format for electronic communication between computers. ASCII assigns standard numeric values to letters, numerals, punctuation marks, and other characters used in computers.

To convert a character to its ASCII code equivalent we can reference an ASCII table

- See: https://www.asciitable.com

The table has five headings: Dec, Hx, Oct, Html and Chr. To convert a character to its ASCII equivalent we find it in the Chr column and then take the corresponding number from the Dec column. For example, we can see “A” has the value of 65.

The conversion can also happen the other way, if we take the ASCII code 65 and look it up in the ASCII table we can see it matches up with the “A” character.

Task

Define a function *string_to_ascii* that takes one parameter:

Name	Type	Example Input
input_str	str	"Programming puzzles!"

When called, the function should return a list containing the ASCII numeric codes of each character of the string.

Starter Code

```
def string_to_ascii(input_str: str) -> list[int]:
    # Your implementation here

print(string_to_ascii("Programming puzzles!"))
```

Examples

```
Input: "Programming puzzles!"
Output: [80, 114, 111, 103, 114, 97, 109, 109, 105, 110, 103, 32,
112, 117, 122, 122, 108, 101, 115, 33]

Input: ""
Output: []

Input: "aA"
Output: [97, 65]
```

PUZZLE 12.1 - BONUS

Task

Define a function *ascii_to_string* that takes one parameter:

Name	Type	Example Input
input_ascii_codes	list of int	[80, 114, 111, 103, 114, 97, 109, 109, 105, 110, 103, 32, 112, 117, 122, 122, 108, 101, 115, 33]

When called, the function should return a string consisting of the converted ASCII codes back to their char equivalents.

Starter Code

```
def ascii_to_string(input_ascii_codes: list[int]) -> str:
    # Your implementation here

print(ascii_to_string([80, 114, 111, 103, 114, 97, 109, 109, 105,
110, 103, 32, 112, 117, 122, 122, 108, 101, 115, 33]))
```

Examples

```
Input: [80, 114, 111, 103, 114, 97, 109, 109, 105, 110, 103, 32,
112, 117, 122, 122, 108, 101, 115, 33]
Output: "Programming puzzles!"

Input: []
Output: ""

Input: [97, 65]
Output: "aA"
```

PUZZLE 13

Task

Define a function *filter_strings_with_vowels* that takes one parameter:

Name	Type	Example Input
input_strs	list of str	["apple", "banana", "zyxvb"]

When called, the function should return a new list with all the strings that have at least one vowel.

Starter Code

```
def filter_strings_with_vowels(input_strs: list[str])
    -> list[str]:
    # Your implementation here

print(filter_strings_with_vowels(["apple", "banana", "zyxvb"]))
```

Examples

```
Input: ["apple", "banana", "zyxvb"]
Output: ["apple", "banana"]

Input: []
Output: []

Input: ["q", "w", "e", "r", "t", "y"]
Output: ["e"]
```

PUZZLE 14

Task

Define a function *reverse_first_five_positions* that takes one parameter:

Name	Type	Example Input	Constraint
input_nums	list of int	[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]	len(input_nums) == 10

When called, the function should return a new list with the first five elements of the original list reversed. The solution should make use of python slicing and should not use loops.

Starter Code

```
def reverse_first_five_positions(input_nums: list[int])
    -> list[int]:
    # Your implementation here

print(reverse_first_five_positions(
    [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
))
```

Examples

```
Input: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
Output: [5, 4, 3, 2, 1, 6, 7, 8, 9, 10]

Input: [100, 90, 80, 70, 60, 50, 40, 30, 20, 10]
Output: [60, 70, 80, 90, 100, 50, 40, 30, 20, 10]

Input: [-1, -2, -3, -4, -5, -6, -7, -8, -9, -10]
Output: [-5, -4, -3, -2, -1, -6, -7, -8, -9, -10]
```

PUZZLE 15

Task

Define a function *filter_palindromes* that takes one parameter:

Name	Type	Example Input
input_strs	list of str	["cat", "dog", "racecar", "deified", "giraffe"]

When called, the function should return a new list that contains only the strings that are palindromes.

Starter Code

```
def filter_palindromes(input_strs: list[str]) -> list[str]:
    # Your implementation here

print(filter_palindromes(
    ["cat", "dog", "racecar", "deified", "giraffe"]
))
```

Examples

```
Input: ["cat", "dog", "racecar", "deified", "giraffe"]
Output: ["racecar", "deified"]

Input: ["kayak", "deified", "rotator", "repaper", "deed", "a"]
Output: ["kayak", "deified", "rotator", "repaper", "deed", "a"]

Input: ["ab", "ba", "cd", "ef", "pt"]
Output: []
```

PUZZLE 16

Task

Define a function *censor_python* that takes one parameter:

Name	Type	Example Input
input_strs	list of str	["python", "hello", "HELLO"]

When called, the function should return a new list of strings with the letters “P”, “Y”, “T”, “H”, “O”, “N” replaced with “X”, the solution should be case insensitive.

Starter Code

```
def censor_python(input_strs: list[str]) -> list[str]:
    # Your implementation here

print(censor_python(["python", "hello", "HELLO"]))
```

Examples

```
Input: ["python", "hello", "HELLO"]
Output: ["XXXXXX", "XellX", "XELLX"]

Input: ["abcdefg"]
Output: ["abcdefg"]

Input: []
Output: []
```

PUZZLE 17

Background

A string is happy if every three consecutive characters are distinct.

Example happy strings:

- “abcdefg”
- “qwerty”
- “abcabcabcabc”

Example unhappy strings:

- “aaaaaaaa”
- “cbc”
- “hello”

Task

Define a function *check_if_string_is_happy* that takes one parameter:

Name	Type	Example Input
input_str	str	"abcdefg"

When called, the function should return a bool indicating if the input string is happy or not.

Starter Code

```
def check_if_string_is_happy(input_str: str) -> bool:
    # Your implementation here

print(check_if_string_is_happy("abcdefg"))
```

Examples

```
Input: "abcdefg"
Output: True

Input: "abcabcabcabc"
Output: True

Input: "hello"
Output: False
```

PUZZLE 18

Task

Define a function *get_number_of_digits* that takes one parameter:

Name	Type	Example Input	Constraint
input_num	int	1234	input_num >= 0

When called, the function should return the number of digits in the *input_num*.

Caveats:

- The function should be recursive
- The function should not convert the integer to a string

Starter Code

```
def get_number_of_digits(input_num: int) -> int:
    # Your implementation here

print(get_number_of_digits(1234))
```

Examples

```
Input: 1234
Output: 4

Input: 0
Output: 1

Input: 123456789
Output: 9
```

PUZZLE 19

Background

Tic-Tac-Toe (also known as naughts and crosses) is a two-player game played on a 3x3 grid. The objective of the game is to be the first player to get three of their symbols in a row, either horizontally, vertically, or diagonally. The game is played by alternating turns, with each player placing their symbol (usually an X or an O) on an empty space on the board until one player achieves a winning configuration or the board is filled without a winner, resulting in a tie.

A Tic-Tac-Toe board can be represented in python by a 2-dimensional list, with each of the inner lists representing a row. Let's look at an example:

```
input_board = [["X", "X", "X"],
               ["O", "X", "O"],
               ["X", "O", "O"]]

Represents the following game configuration:
X X X
O X O
X O O
```

Task

Define a function *get_tic_tac_toe_winner* that takes one parameter:

Name	Type	Example Input
input_board	list of list of str	[["X", "X", "X"], ["O", "X", "O"], ["X", "O", "O"]]

When called, the function should determine if X or O has won. If there is a draw the function should return *None*.

Starter Code

```
def get_tic_tac_toe_winner(input_board: list[list[str]])
    -> str | None:
    # Your implementation here

print(get_tic_tac_toe_winner(
    [["X", "X", "X"], ["O", "X", "O"], ["X", "O", "O"]]
))
```

Examples

```
Input: [["X", "X", "X"], ["O", "X", "O"], ["X", "O", "O"]]
Output: "X"

Input: [["X", "O", "O"], ["O", "O", ""], ["X", "O", "O"]]
Output: "O"

Input: [["X", "O", "O"], ["O", "X", ""], ["X", "O", "O"]]
Output: None
```

PUZZLE 20

Task

Define a function *print_triangle* that takes two parameters:

Name	Type	Example Input
number_of_levels	int	4
symbol	str	“*”

When called, the function should output a centred triangle shape made up of the desired symbol. The number of symbols in each row should increase by two with each level, starting with one symbol on the first level, then three on the second level and so on until the final level is reached. For example, for *number_of_levels = 4* and *symbol = ”*”* the following triangle should be produced:

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

Starter Code

```
def print_triangle(number_of_levels: int, symbol: str) -> None:
    # Your implementation here

print_triangle(4, "*")
```

Examples

```
Inputs:
 - number_of_levels = 3
 - symbol = "*"

Output:
  *
 ***
*****

Inputs:
 - number_of_levels = 1
 - symbol = "|"

Output:
|
```

PUZZLE 21

Background

The Fibonacci sequence is a well-known mathematical series of numbers that has fascinated mathematicians and scientists for centuries. It is a sequence of numbers where each number in the sequence is the sum of the two preceding numbers. The sequence starts with 0, 1, and each subsequent number is the sum of the previous two numbers.

The sequence goes 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, and so on.

Task

Define a function *fibonacci* that takes one parameter:

Name	Type	Example Input
sequence_number	int	4

When called, the function should return the corresponding fibonacci sequence number (starting from 0).

Caveats:

- The function should be recursive.

Starter Code

```
def fibonacci(sequence_number: int) -> int:
    # Your implementation here

print(fibonacci(4))
```

Examples

```
Input: 4
Output: 3

Input: 0
Output: 0

Input: 6
Output: 8
```

PUZZLE 22

Background

A harmonic sum is a mathematical concept that is used to calculate the sum of the reciprocals of a set of numbers. The reciprocal of a number is defined as 1 divided by the number. For example, the reciprocal of 2 is 0.5, because 1 divided by 2 is 0.5.

In the case of a harmonic sum, the set of numbers is usually represented by the natural numbers from 1 to n. The formula for the harmonic sum of n is: $H_n = 1/1 + 1/2 + 1/3 + ... + 1/n$.

Task

Define a function *harmonic_sum* that takes one parameter:

Name	Type	Example Input
n	int	5

When called, the function should return the harmonic sum of *n*.

Caveats:

- The function should be recursive.

Starter Code

```
def harmonic_sum(n: int) -> float:
    # Your implementation here

print(harmonic_sum(5))
```

Examples

```
Input: 5
Output: 2.283

Input: 2
Output: 1.5

Input: 0
Output: 0
```

PUZZLE 23

Background

The XOR (Exclusive Or) logic gate is a digital circuit that performs a logical operation on two inputs. The output of an XOR gate is true (1) if and only if one of the inputs is true and the other is false (0). In other words, the XOR gate compares the inputs, and if they are different, the output is true. If they are the same, the output is false.

This can be represented in a truth table, which is a table that shows the output of a logic gate based on all possible combinations of inputs. Here is an example of a truth table for the XOR gate:

Input A	Input B	Output
0	0	0
0	1	1
1	0	1
1	1	0

In this example, the left column represents input A, the middle column represents input B, and the right column represents the output. As you can see, when the inputs are different, the output is 1 and when the inputs are the same, the output is 0.

Task

Define a function *xor* that takes two parameters:

Name	Type	Example Input
input_a	str	“1101”
input_b	str	“0001”

When called, the function should return a string value that is the result of XOR’ing the two input strings together. If one string is longer than the other the excess characters should be ignored from the result.

Starter Code

```
def xor(input_a: str, input_b: str) -> str:
    # Your implementation here

print(xor("1101", "0001"))
```

Examples

```
Inputs:
 - input_a: "1111"
 - input_b: "1111"

Output: "0000"

Inputs:
 - input_a: "1111"
 - input_b: "0000"

Output: "1111"

Inputs:
 - input_a: "1101"
 - input_b: "00010"

Output: "1100"
```

PUZZLE 24

Background

In Python, the zip function takes in one or more iterable objects (such as lists, tuples, or strings) and returns an iterator of tuples. Each tuple contains elements from each iterable object in the same position. The zip function stops when it reaches the end of the shortest iterable object.

Here is an example of how the zip function works:

```
a = [1, 2, 3]
b = [4, 5, 6]

zipped = zip(a, b)
print(list(zipped)) # [(1, 4), (2, 5), (3, 6)]
```

Task

Define a function *my_zip* that takes two parameters:

Name	Type	Example Input
input_list_a	list of Any	[1, 2, 3, 4]
input_list_b	list of Any	[5, 6, 7, 8]

When called, the function should return the same result as python's built-in zip function.

Starter Code

```
from typing import Any

def my_zip(input_list_a: list[Any], input_list_b: list[Any])
    -> list[tuple[Any, Any]]:
    # Your implementation here

print(my_zip([1, 2, 3, 4], [5, 6, 7, 8]))
```

Examples

```
Inputs:
 - input_list_a: [1, 2, 3, 4]
 - input_list_b: [5, 6, 7, 8]

Output: [(1, 5), (2, 6), (3, 7), (4, 8)]

Inputs:
 - input_list_a: []
 - input_list_b: []

Output: []

Inputs:
 - input_list_a: [1, 2, 3]
 - input_list_b: [5, 6, 7, 8]

Output: [(1, 5), (2, 6), (3, 7)]
```

PUZZLE 25

Task

Define a function *is_valid_equation* that takes one parameter:

Name	Type	Example Input	Constraint
input_equation	str	“2 + 3 = 5”	The string should consist of an integer, followed by a plus or a minus sign, followed by another integer, an equals sign, and the answer. The equation should be separated by spaces.

When called, the function should return a boolean value indicating whether the equation is valid or not. The equation is classed as valid if:

- It’s in the correct format as specified above
- Both sides of the equation evaluate to the same number

Starter Code

```
def is_valid_equation(input_equation: str) -> bool:
    # Your implementation here

print(is_valid_equation("2 + 3 = 5"))
```

Examples

```
Input: "2 + 3 = 5"
Output: True
Input: "-5 + -6 = -11"
Output: True
Input: "-2 + 3 = -5"
Output: False
```

PUZZLE 26

Task

Define a function *rotate_list_left* that takes two parameters:

Name	Type	Example Input
input_list	list of Any	[1, 2, 3, 4, 5]
rotate_amount	int	2

When called, the function should return a new list with the elements of the original list rotated left by the specified number of positions.

Caveats:

- The solution should not make use of loops
- The function should still work if the rotation amount is greater than the length of the list, e.g. a rotation amount of 6 on a list of length 5 will produce the same result as if the rotation amount was 1.

Starter Code

```
from typing import Any

def rotate_list_left(input_list: list[Any], rotate_amount: int)
    -> list[Any]:
    # Your implementation here

print(rotate_list_left([1, 2, 3, 4, 5], 2))
```

Examples

```
Inputs:
 - input_list: [1, 2, 3, 4, 5]
 - rotate_amount: 2
Output: [3, 4, 5, 1, 2]

Inputs:
 - input_list: [1, 2, 3, 4, 5]
 - rotate_amount: 5
Output: [1, 2, 3, 4, 5]

Inputs:
 - input_list: [1, 2, 3, 4, 5]
 - rotate_amount: 7
Output: [3, 4, 5, 1, 2]
```

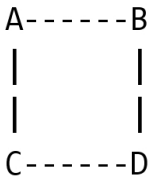
PUZZLE 27

Background

An undirected graph is a mathematical representation of a set of objects in which some pairs of the objects are connected by links. The objects are represented by vertices (or nodes) and the links are represented by edges. In an undirected graph, the edges have no direction, meaning they can be traversed in both directions. It means if there is an edge between vertex A and B, then it is also true that there is an edge between vertex B and A.

A simple example of an undirected graph would be a group of cities and the roads connecting them. Each city would be represented by a vertex, and the roads connecting the cities would be represented by edges. It would not matter whether you are travelling from city A to city B or from city B to city A, the edge between them would still be the same.

It could look something like this:



Each letter represents a vertex (city) and the lines represent edges (roads). Here, A and B are connected by an edge and A and C are also connected by an edge.

Graphs are commonly represented in python using an adjacency matrix. The concept is simple, we have the list of nodes across the top and a list of nodes down the side. If two nodes are connected we place a “1” in the connection cell. If they’re not connected we have a 0.

Example: Node A is connected to itself, Node B and Node C. Node B is only connected to node A. Node C is only connected to Node A.

	Node A	Node B	Node C
Node A	1	1	1
Node B	1	0	0
Node C	1	0	0

In python, this can be represented as a list of lists where each node is represented by an index. For example Nodes A, B, C would be represented by indexes 0, 1, 2 respectively:

```
graph = [
    [1, 1, 1],
    [1, 0, 0],
    [1, 0, 0]
]

graph[0][0] == 1 # Is Node A connected to itself?
graph[0][1] == 1 # Is Node A connected to Node B?
graph[2][1] == 1 # Is Node C connected to Node B?
```

Task

Define a function *find_adjacent_nodes* that takes two parameters:

Name	Type	Example Input
adj_matrix	list of list of int	[[1, 1, 1], [1, 0, 0], [1, 0, 0]]
start_node	int	0

When called, the function should return a list of all nodes that are adjacent to *start_node*.

Challenge: this challenge can be achieved with the function body being only one line of code, are you able to find the solution?

Starter Code

```
def find_adjacent_nodes(
    adj_matrix: list[list[int]],
    start_node: int
) -> list[int]:

    # Your implementation here

print(
    find_adjacent_nodes([[1, 1, 1], [1, 0, 0], [1, 0, 0]], 0)
)
```

Examples

```
Inputs:
 - adj_matrix: [[1, 1, 1], [1, 0, 0], [1, 0, 0]]
 - start_node: 0
Output: [0, 1, 2]

Inputs:
 - adj_matrix: [[1, 1, 1], [1, 0, 0], [1, 0, 0]]
 - start_node: 1
Output: [0]

Inputs:
 - adj_matrix: [[0, 1, 1, 0], [1, 0, 0, 1], [1, 0, 0, 1], [0, 1, 1,
0]]
 - start_node: 1
Output: [0, 3]
```

PUZZLE 28

Background

In technical analysis, a peak refers to a high point or local maximum in the price of an asset. A valley, on the other hand, refers to a low point or local minimum in the price of an asset. When looking at a chart of the price of an asset, peaks and valleys can help traders identify potential turning points and potential changes in the overall trend of the asset.

For our sake, peaks and valleys are defined as the following:

- Peak:
    - A peak must have one or more numbers in ascending order leading up to it.
    - A peak must have one or more numbers in descending order after it.
    - A peak can not occur at the start or end of price action.
- Valley
    - A valley must have one or more numbers in descending order leading up to it.
    - A valley must have one or more numbers in ascending order after it.
    - A valley can not occur at the start or end of price action.

Task

Define a function *count_peaks_valleys* that takes one parameter:

Name	Type	Example Input
price_action	list of int	[1, 2, 3, 2, 1]

When called, the function should return a tuple representing how many peaks and valleys are in the given price action. The tuple should contain two integers, the first representing the number of peaks and the second representing the number of valleys.

Starter Code

```
def count_peaks_valleys(price_action: list[int])
    -> tuple[int, int]:
    # Your implementation here

print(count_peaks_valleys([1, 2, 3, 2, 1]))
```

Examples

```
Input: [1, 2, 3, 2, 1]
Output: (1, 0)

Input: [1, 2, 3, 2, 1, 2]
Output: (1, 1)

Input: [7, 6, 5, 10, 11, 12, 10, 9, 10]
Output: (1, 2)
```

PUZZLE 29

Background

Tap code is a simple method for transmitting messages through a series of taps or knocks. It was originally developed for prisoners of war to communicate with each other in secret, but it has also been used in other situations where communication is difficult, such as by hikers lost in the wilderness or by people trapped in a collapsed building. Tap code uses the following 5x5 grid to represent each letter:

	1	2	3	4	5
1	A	B	C / K	D	E
2	F	G	H	I	J
3	L	M	N	O	P
4	Q	R	S	T	U
5	V	W	X	Y	Z

Note: C and K are both represented by (1, 3) - when coding you can choose to either return C or K.

To transmit a message, the sender taps out the row and column of each letter in the message, with a pause between letters and a longer pause between words. The receiver then uses tap code to decode the message.

As an example, the word “water” would be represented by:

- 5, 2
- 1, 1
- 4, 4
- 1, 5
- 4, 2

Or when using in real life a series of taps: ••••• •• • • •••• •••• • ••••• •••• ••

Task

Define a function *tap_code_to_english* that takes one parameter:

Name	Type	Example Input
input_code	str	".. ... ."

When called, the function should return the converted sentence. Items within the string should be represented as the following:

- Letters: 1-5 dots followed by a space, followed by 1-5 dots e.g. “. ..” = “b”
- End of letter / start of new letter: two spaces e.g. “. .. . ..” = “bb”
- End of word / start of new word: three spaces “.. … .. …. .. … .. ….” = “hi hi”

Starter Code

```
tap_code_map = {
    "a": ". .", "b": ". ..", "c": ". ...", "d": ". ....",
    "e": ". .....", "f": ".. .", "g": ".. ..", "h": ".. ...",
    "i": ".. ....", "j": ".. .....", "l": "... .",
    "m": "... ..", "n": "... ...", "o": "... ....",
    "p": "... .....", "q": ".... .", "r": ".... ..",
    "s": ".... ...", "t": ".... ....", "u": ".... .....",
    "v": "..... .", "w": "..... ..", "x": "..... ...",
    "y": "..... ....", "z": "..... ....."
}

def tap_code_to_english(input_code: str) -> str:
    # Your implementation here

print(tap_code_to_english(".. ...  .. ....   .. ...  .. ...."))
```

Examples

```
Input: ".. ...  .. ....   .. ...  .. ...."
Output: "hi hi"

Input: ". ...  ... ....  ... ....  ... ."
Output: "cool"

Input: ""
Output: ""
```

PUZZLE 30

Task

Define a function *find_zero_sum_triplets* that takes one parameter:

Name	Type	Example Input
input_nums	list of int	[1, 2, 3, 4, 5, -9]

When called, the function should return all possible combinations of the indexes of three numbers that add up to 0. The function should return a list of tuples, each tuple containing the three indexes of the numbers that add up to 0, or an empty list if no such combination exists. The function should be able to deal with duplicate numbers in the input list.

Starter Code

```
def find_zero_sum_triplets(input_nums: list[int])
    -> list[tuple[int, int, int]]:
    # Your implementation here

print(find_zero_sum_triplets([1, 2, 3, 4, 5, -9]))
```

Examples

```
Input: [1, 2, 3, 4, 5]
Output: []

Input: [1, 2, 3, 4, 5, -9]
Output: [(3, 4, 5)]

Input: [1, 2, 3, 4, 5, -9, -9]
Output: [(3, 4, 5), (3, 4, 6)]
```

PUZZLE 31

Task

Define a function *param_count* that takes an arbitrary number of arguments.

When called, the function should return the number of arguments it was called with.

Starter Code

```
from typing import Any

def param_count(*args: Any) -> int:
    # Your implementation here

print(param_count(1, 2, 3, 4, 5))
```

Examples

```
Inputs: 1, 2, 3, 4, 5
Output: 5

Inputs: "hello"
Output: 1

Inputs:
Output: 0
```

PUZZLE 31.1 - BONUS

Task

Using the *my_zip* function from puzzle 24 as context, improve the function so that it can take an arbitrary number of arguments.

When called, the function should return the same result as python's built-in zip function. Your function can either return a list, or if you want to follow python's interpretation more closely it can return an iterator.

Starter Code

Return a list:

```
from typing import Any

def my_zip(*input_lists: list[Any]) -> list[tuple[Any, ...]]:
    # Your implementation here

print(my_zip_one([1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]))
```

Return an iterator:

```
from typing import Any
from collections.abc import Iterator

def my_zip(*input_lists: list[Any])
    -> Iterator[tuple[Any, ...]]:
    # Your implementation here

print(list(
    my_zip_two([1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12])
))
```

Examples

```
Inputs: [1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]
Output: [(1, 5, 9), (2, 6, 10), (3, 7, 11), (4, 8, 12)]

Inputs: [1, 2, 3], [5, 6, 7, 8]
Output: [(1, 5), (2, 6), (3, 7)]

Input: [], []
Output: []
```

PUZZLE 32

Task

Define a function *contains_python_chars* that takes one parameter:

Name	Type	Example Input
input_str	str	"Nohtyp"

When called, the function should return a boolean value indicating whether the string contains any combination of the word “python”. The letters of “python” can be in any order e.g. “nohtyp” but must not be interrupted by any other characters. The function should be case-insensitive.

Starter Code

```
def contains_python_chars(input_str: str) -> bool:
    # Your implementation here

print(contains_python_chars("Nohtyp"))
```

Examples

```
Input: "pYThon"
Output: True

Input: "Nohtyp"
Output: True

Input: "pythZon"
Output: False
```

PUZZLE 33

Background

A prime number is a positive integer that is divisible by only 1 and itself. It has no other positive divisors. Prime numbers are also often referred to as "primes" or "irreducible numbers". Some examples of prime numbers are 2, 3, 5, 7, 11, 13, 17, 19. They play an important role in number theory and have many applications in fields like cryptography, coding theory, and construction of pseudorandom number generators.

Task

Define a function *find_primes* that takes one parameter:

Name	Type	Example Input
input_nums	list of int	[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

When called, the function should return a new list containing only the prime numbers from *input_nums*.

Starter Code

```python
def find_primes(input_nums: list[int]) -> list[int]:
    # Your implementation here

print(find_primes([1, 2, 3, 4, 5, 6, 7, 8, 9, 10]))
```

Examples

```
Input: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
Output: [2, 3, 5, 7]

Input: [-1, -2, -3, -4, -5, -6, -7, -8, -9, -10]
Output: []

Input: [2, 3, 5, 7, 11, 13, 17]
Output: [2, 3, 5, 7, 11, 13, 17]
```

PUZZLE 34

Background

ROT13 is a simple encryption technique that replaces each letter in a message with the letter 13 positions ahead of it in the alphabet. For example, "A" becomes "N", "B" becomes "O", and so on.

To calculate ROT13 for a given message, you simply need to shift each letter 13 positions ahead in the alphabet. If a letter is near the end of the alphabet and there are not enough letters left to shift it 13 positions ahead, you would wrap around to the beginning of the alphabet and continue counting.

Task

Define a function *rot13* that takes one parameter:

Name	Type	Example Input
input_str	str	"Hello world!"

When called, the function should return a new string which has been encrypted using ROT13. Numbers and symbols should not be rotated, they can be kept the same.

Starter Code

```
def rot13(input_str: str) -> str:
    # Your implementation here

print(rot13("Hello world!"))
```

Examples

```
Input: "Hello world!"
Output: "Uryyb jbeyq!"

Input: "Cool puzzles!"
Output: "Pbby chmmyrf!"

Input: "12345!@£$%"
Output: "12345!@£$%"
```

PUZZLE 35

Task

Define a function *get_parentheses_groups* that takes one parameter:

Name	Type	Example Input	Constraint
input_str	str	"(( )) (( )) ((( ))"	The parentheses in the string will always match up e.g. N opening parentheses should be followed by N closing parentheses.

When called, the function should return a list containing groups of fully matched parentheses without any spaces. Each parenthesis group should be one item in the list.

Starter Code

```
def get_parentheses_groups(input_str: str) -> list[str]:
    # Your implementation here

print(get_parentheses_groups("(( ))  ((  ) ) (   ((  )))"))
```

Examples

```
Input: "(( ))  ((  ) ) (   ((  )))"
Output: ["(())", "(())", "((()))"]

Input: "( ( ( ( ( ( ) ) ) ) ) )"
Output: ["((((((()))))))"]

Input: ""
Output: []
```

PUZZLE 36

Background

In mathematics, a matrix is a rectangular array of numbers, symbols, or expressions, arranged in rows and columns. Matrices are often labelled by capital letters, such as A, B, C, etc. Each element in a matrix is referred to by its row and column indices.

For example, consider the following matrix A:

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

This matrix has 3 rows and 3 columns, and its elements can be referred to as A_ij, where i is the row index and j is the column index. So, for example, A_12 = 2 and A_31 = 7.

Matrices can be used to represent large sets of data and operations can be performed on them such as matrix multiplication.

To perform matrix multiplication we need to ensure the number of columns of the left matrix equals the number of rows in the right matrix. Once that's confirmed, we can use the following steps:

1. Determine the size of the resulting matrix: the resulting matrix will have the same number of rows as the first matrix and the same number of columns as the second matrix.
2. Create a matrix of the correct size with all elements set to zero.
3. For each element in the resulting matrix, multiply the elements of the corresponding row in the first matrix and the corresponding column in the second matrix.
4. Sum up the products obtained in the previous step and assign the result to the corresponding element in the resulting matrix.
5. Repeat steps 3 to 4 for each element in the resulting matrix.
6. Return the resulting matrix.

Working example:

```
A = [[2, 3], [4, 5]]
B = [[10, 15], [5, 1]]
C = [[0, 0], [0, 0]]

C[0][0] = (A[0][0] * B[0][0]) + (A[0][1] * B[1][0])
C[0][1] = (A[0][0] * B[0][1]) + (A[0][1] * B[1][1])
C[1][0] = (A[1][0] * B[0][0]) + (A[1][1] * B[1][0])
C[1][1] = (A[1][0] * B[0][1]) + (A[1][1] * B[1][1])

Output: [[35, 33], [65, 65]]
```

Task

Define a function *matrix_multiply* that takes two parameters:

Name	Type	Example Input
left_matrix	list of list of int	[[2, 3], [4, 5]]
right_matrix	list of list of int	[[10, 15], [5, 1]]

When called, the function should return the result of the left and right matrix being multiplied together.

Starter Code

```
A = [[2, 3], [4, 5]]
B = [[10, 15], [5, 1]]

def matrix_multiply(
    left_matrix: list[list[int]], right_matrix: list[list[int]]
) -> list[list[int]]:

    # Your implementation here

print(matrix_multiply(A, B))
```

Examples

```
Inputs:
 - left_matrix: [[2, 3], [4, 5]]
 - right_matrix: [[10, 15], [5, 1]]

Output: [[35, 33], [65, 65]]

Inputs:
 - left_matrix: [[1, 2, 3, 4, 5, 6], [1, 2, 3, 4, 5, 6]]
 - right_matrix: [[1, 2], [3, 4], [5, 6], [7, 8], [9, 10], [11,
12]]

Output: [[161, 182], [161, 182]]

Inputs:
 - left_matrix: [[1, 2, 3, 4, 5, 6], [1, 2, 3, 4, 5, 6]]
 - right_matrix: [[1, 2, 3]]

Output: None
```

PUZZLE 37

Background

The greatest common divisor of two or more integers is the largest positive integer that divides each of the integers without a remainder. For example, the GCD of 8 and 12 is 4, because 4 is the largest number that divides both 8 and 12 without leaving a remainder.

Task

Define a function *gcd* that takes two parameters:

Name	Type	Example Input
num_one	int	36
num_two	int	8

When called, the function should return the greatest common divisor of two integers.

Starter Code

```
def gcd(num_one: int, num_two: int) -> int:
    # Your implementation here

print(gcd(36, 8))
```

Examples

```
Inputs:
 - num_one: 36
 - num_two: 8

Output: 4

Inputs:
 - num_one: 5
 - num_two: 25

Output: 5

Inputs:
 - num_one: 5
 - num_two: 26

Output: 1
```

PUZZLE 38

Task

Define a function *find_pairs_summing_to_target* that takes two parameters:

Name	Type	Example Input
input_nums	list of int	[1, 2, 3, 4, 5, 6, 7, 8, 9]
target	int	int

When called, the function should return a list of all pairs in *input_nums* whose sum is equal to *target*.

Caveat:

- The resulting list should not contain duplicate pairs e.g. if *input_nums=[5, 5, 5, 5]* and *target=10* then the result should be [(5, 5)] instead of [(5, 5), (5, 5), (5, 5), (5, 5), (5, 5), (5, 5)].

Starter Code

```
def find_pairs_summing_to_target(
    input_nums: list[int], target: int
) -> list[tuple[int, int]]:

    # Your implementation here

print(find_pairs_summing_to_target(
    [1, 9, 2, 8, 3, 7, 4, 6, 5, 5], 10
))
```

Examples

```
Inputs:
 - input_nums: [5, 5, 5, 5]
 - target: 10
Output: [(5, 5)]

Inputs:
 - input_nums: [1, 2, 3, 4, 5, 6, 7, 8, 9]
 - target: 10
Output: [(1, 9), (2, 8), (3, 7), (4, 6)]

Inputs:
 - input_nums: [11, 12, 13, 14, 15]
 - target: 5
Output: []
```

PUZZLE 38.1 - BONUS

Task

Improve the *find_pairs_summing_to_target* function from puzzle 38 so that it doesn't use nested loops. The inputs and output to the function will be the same. If your original solution didn't make use of nested loops then feel free to skip this bonus task.

Starter Code

```
def find_pairs_summing_to_target_bonus(
    input_nums: list[int], target: int
) -> list[tuple[int, int]]:

    # Your implementation here

print(find_pairs_summing_to_target_bonus(
    [1, 9, 2, 8, 3, 7, 4, 6, 5, 5], 10
))
```

Examples

```
Inputs:
 - input_nums: [5, 5, 5, 5]
 - target: 10

Output: [(5, 5)]

Inputs:
 - input_nums: [1, 2, 3, 4, 5, 6, 7, 8, 9]
 - target: 10

Output: [(1, 9), (2, 8), (3, 7), (4, 6)]

Inputs:
 - input_nums: [11, 12, 13, 14, 15]
 - target: 5

Output: []
```

PUZZLE 39

Background

The Tower of Hanoi is a mathematical puzzle that consists of three rods and a number of disks of different sizes which can slide onto any rod. The puzzle starts with the disks in a neat stack in ascending order of size on one rod, the smallest at the top, thus making a conical shape.

The objective of the puzzle is to move the entire stack to another rod, obeying the following rules:

1. Only one disk can be moved at a time.
2. Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack or on an empty rod.
3. No disk may be placed on top of a smaller disk.

Task

Define a function *tower_of_hanoi* that takes four parameters:

Name	Type	Example Input	Description
num_disks	int	4	Number of disks starting on the source peg
source	str	“Source”	A string representing the source peg
aux	str	“Auxiliary”	A string representing the auxiliary peg
target	str	“Target”	A string representing the target peg

When called, the function should solve the Tower of Hanoi problem. The output should be printed to the console in the following format: “Move disk {n} from {peg} to {another_peg}” where n is the size of the disk. For example, the smallest disk will have size 1, the size up will be size 2, etc.

Starter Code

```
def tower_of_hanoi(num_disks: int, source: str, aux: str, target:
str) -> None:
    # Your implementation here

tower_of_hanoi(4, "Source", "Auxiliary", "Target")
```

Examples

```
Inputs:
 - num_disks: 2
 - source: "Source"
 - aux: "Auxiliary"
 - target: "Target"
Output:
Move disk 1 from Source to Auxiliary
Move disk 2 from Source to Target
Move disk 1 from Auxiliary to Target
```

```
Inputs:
 - num_disks: 2
 - source: "Source"
 - aux: "Auxiliary"
 - target: "Target"
Output:
Move disk 1 from Source to Auxiliary
Move disk 2 from Source to Target
Move disk 1 from Auxiliary to Target
Move disk 3 from Source to Auxiliary
Move disk 1 from Target to Source
Move disk 2 from Target to Auxiliary
Move disk 1 from Source to Auxiliary
Move disk 4 from Source to Target
Move disk 1 from Auxiliary to Target
Move disk 2 from Auxiliary to Source
Move disk 1 from Target to Source
Move disk 3 from Auxiliary to Target
Move disk 1 from Source to Auxiliary
Move disk 2 from Source to Target
Move disk 1 from Auxiliary to Target
```

PUZZLE 40

Background

Insertion sort is a simple sorting algorithm that builds the final sorted list one item at a time. It iterates through the input list, and for each element, it compares it to the elements that come before it and then inserts the element in the correct position. This process continues until all elements have been compared and placed in the correct position in the final sorted list.

It is called insertion sort because it can be thought of as inserting each element from the input list into its correct position in the final sorted list.

An example of insertion sort is sorting a deck of cards: starting with an empty left hand, we remove one card at a time from the deck with the right hand and place it in the correct position in the left hand.

Task

Define a function *insertion_sort* that takes one parameter:

Name	Type	Example Input
input_nums	list of int	[5, 10, 9, 11, 4]

The function should implement the insertion sort algorithm and return a sorted version of *input_nums* when called.

Starter Code

```
def insertion_sort(input_nums: list[int]) -> list[int]:
    # Your implementation here

print(insertion_sort([5, 10, 9, 11, 4]))
```

Examples

```
Input: [5, 10, 9, 11, 4]
Output: [4, 5, 9, 10, 11]

Input: [1, 2, 3, 4, 5]
Output: [1, 2, 3, 4, 5]

Input: [-1, -2, -3, -4, -5]
Output: [-5, -4, -3, -2, -1]
```

PUZZLE 41

Background

Roman numerals are a numerical system that originated in ancient Rome and were widely used throughout the Roman Empire. They are a way of representing numbers using a combination of letters from the Latin alphabet. The system uses seven letters, each of which has a corresponding numerical value: I (1), V (5), X (10), L (50), C (100), D (500), and M (1000).

You can convert from integers to roman numerals using the following steps:

1. Create a lookup table for Roman numerals (see starter code).
2. Find the largest Roman numeral that is less than or equal to the integer, and subtract it from the integer.
3. Append the corresponding Roman numeral to the result string.
4. Subtract the corresponding integer value from the original integer.
5. Repeat steps 2, 3 and 4 with the remaining integer until the integer is 0.
6. Return the result string.

Task

Define a function *int_to_roman* that takes one parameter:

Name	Type	Example Input	Constraint
input_num	int	3000	The integer should be between 1 and 4999 inclusive.

When called, the function should return the input number converted to a roman numeral.
Remember to meet the following constraints of roman numerals:

1. You can only use the Roman numerals I, V, X, L, C, D and M, no other characters or notation is allowed.

2. You must follow the traditional rule of Roman numeral formation where smaller numerals are placed before larger numerals to indicate subtraction. For example, 4 is represented as IV, not IIII.

Starter Code

```
roman_map = {
    1000: "M", 900: "CM", 500: "D", 400: "CD", 100: "C",
    90: "XC", 50: "L", 40: "XL", 10: "X", 9: "IX", 5: "V",
    4: "IV",1: "I"
}


def int_to_roman(input_num: int) -> str:
    # Your implementation here

print(int_to_roman(4))
```

Examples

```
Input: 4
Output: "IV"

Input: 27
Output: "XXVII"

Input: 4999
Output: "MMMMCMXCIX"
```

PUZZLE 41.1 - BONUS

Task

Define two extra functions: *roman_to_int* and *int_roman_converter*.

The *roman_to_int* function will carry out the reverse of the *int_to_roman* defined in puzzle 41. The function will take a string containing roman numerals and will return the corresponding integer.

Name	Type	Example Input	Constraint
input_str	str	"XXVII"	The representative integer should be between 1 and 4999 inclusive.

The *int_roman_converter* function will take either an integer or a string containing roman numerals. Depending on the input the function will either:

- Return the input integer converted to a roman numeral.
- Return the input string converted to an integer.

The function can determine whether it's been provided an integer or string by making use of pythons built in *isinstance* function.

Name	Type	Example Input	Constraint
to_convert	str or int	"XXVII"	The representative integer should be between 1 and 4999 inclusive.

Starter Code

```
# ...your solution to puzzle 41

def roman_to_int(input_str: str) -> int:
    # Your implementation here

def int_roman_converter(to_convert: str | int) -> int | str:
    # Your implementation here

for i in range(1, 5000):
    is_equal = int_roman_converter(int_roman_converter(i)) == i

    if not is_equal:
        print(f"{i} is incorrect!")
```

Examples

```
# roman_to_int examples
Input: "IV"
Output: 4

Input: "XXVII"
Output: 27

Input: "MMMMCMXCIX"
Output: 4999


# int_roman_converter examples
Input: "IV"
Output: 4

Input: 27
Output: "XXVII"

Input: "MMMMCMXCIX"
Output: 4999
```

PUZZLE 42

Task

Define a function *bitwise_add* that takes two parameters:

Name	Type	Example Input
num_one	int	3
num_two	int	4

When called, the function should return the sum of the two input parameters.

Caveat:

- The function should not make use of any arithmetic operators.

Note: if you're not familiar with binary numbers you'll need to learn the basics before attempting this question. Google something along the lines of "introduction to binary numbers". You should also learn about logic gates.

Starter Code

```
def bitwise_add(num_one: int, num_two: int) -> int:
    # Your implementation here

print(bitwise_add(3, 4))
```

Examples

```
Inputs:
 - num_one: 3
 - num_two: 4
Output: 7

Inputs:
 - num_one: 255
 - num_two: 256
Output: 511

Inputs:
 - num_one: -1
 - num_two: -2
Output: -3
```

PUZZLE 43

Background

Binary search is an efficient algorithm for finding an item from a sorted list of items. The basic idea behind the algorithm is to repeatedly divide the search interval in half until the value is found, or the search interval is empty.

To carry out a binary search the following steps can be executed:

1. Set the search interval equal to the whole list.

2. Calculate the middle element. If the length of the list is even, we take the lower middle element e.g. for [1, 2, 3, 4] we would say the middle element is 2.

3. The middle element of the search interval is compared to the value we are searching for, then:
    a. If the middle element is equal to the value we are searching for, the algorithm stops and returns the index of the element.

    b. If the middle element is greater than the value we are searching for, we know that the element we are searching for must be in the left half of the list. Therefore, the search interval is updated to be the left half of the current interval.

    c. If the middle element is less than the value we are searching for, we know that the element we are searching for must be in the right half of the list. Therefore, the search interval is updated to be the right half of the current interval.

4. The algorithm repeats step 3 with the updated search interval, until the value is found or the search interval is empty.

The functions steps can be represented using a simple walkthrough:

```
Sorted list = [2, 3, 4, 10, 40]
Value to find = 10

# Step 1
Search interval = sorted list
Middle element = 4

# Step 2
Is 10 equal to 4? No
Is 10 greater than 4? Yes
Is 10 less than 4? No

Search interval = [10, 40]
Middle element = 10

Is 10 equal to 10? Yes - value found
```

Task

Define a function *binary_search* that takes two parameters:

Name	Type	Example Input
sorted_list	list of int	[2, 3, 4, 10, 40]
value_to_find	int	10

When called, the function should perform a binary search on the *sorted_list* and return the index of *value_to_find* if present. If *value_to_find* isn't present in *sorted_list* the function should return -1.

Caveat:

- The function should have a time complexity of *O(log n)*

Starter Code

```
def binary_search(sorted_list: list[int], value_to_find: int)
    -> int:
    # Your implementation here

searchable_list = [2, 3, 4, 10, 40]
print(binary_search(searchable_list, 10))
```

Examples

```
Inputs:
 - sorted_list: [2, 3, 4, 10, 40]
 - value_to_find: 10

Output: 3

Inputs:
 - sorted_list: [2, 3, 4, 10, 40]
 - value_to_find: 0

Output: -1

Inputs:
 - sorted_list: []
 - value_to_find: 0

Output: -1
```

PUZZLE 44

Background

Quicksort is a divide-and-conquer algorithm that is used for sorting lists of items. It works by selecting a "pivot" element from the list and partitioning the other elements into two sub-lists according to whether they are less than or greater than the pivot. The sub-lists are then sorted recursively, which eventually results in the complete list being sorted.

The steps of the algorithm can be described as follows:

1. Choose a pivot element from the list. This element will be used to partition the list into two sub-lists.
2. Partition the list around the pivot element by moving all elements less than the pivot to the left of the pivot and all elements greater than the pivot to the right of the pivot.
3. Recursively sort the left sub-list.
4. Recursively sort the right sub-list.
5. Combine the sub-lists and the pivot to obtain the sorted list.

The choice of pivot element affects the performance of the algorithm. If a good pivot is chosen, the algorithm will be efficient, but if a poor pivot is chosen, the algorithm may become slow. A common strategy for choosing a pivot is to take the first, middle, or last element of the list.

Quicksort has an average time complexity of $O(n \log n)$, making it a very efficient sorting algorithm, especially for large lists. It is also an "in-place" sorting algorithm, which means that it does not require additional memory to sort the list as it sorts the list in place by exchanging elements within the list.

Task

Define a function *quicksort* that takes three parameters:

Name	Type	Example Input
input_list	list of int	[5, 7, 8, 1, 2, 4, 99, 77, 56, 43, 12, 98]
low	int	0
high	int	11

When called, the function should sort the *input_list* using the quicksort algorithm.

Starter Code

```
def quicksort(input_list: list[int], low: int, high: int)
    -> list[int]:
    # Your implementation here

unsorted_list = [5, 7, 8, 1, 2, 4, 99, 77, 56, 43, 12, 98]
print(quicksort(unsorted_list, 0, len(unsorted_list) - 1))
```

Examples

```
Inputs:
 - input_list: [5, 7, 8, 1, 2, 4, 99, 77, 56, 43, 12, 98]
 - low: 0
 - high: len(input_list) - 1

Output: [1, 2, 4, 5, 7, 8, 12, 43, 56, 77, 98, 99]

Inputs:
 - input_list: [10, 5, -10, -5, 0]
 - low: 0
 - high: len(input_list) - 1

Output: [-10, -5, 0, 5, 10]

Inputs:
 - input_list: []
 - low: 0
 - high: 0

Output: []
```

PUZZLE 45

Background

Given a set of items, each with a weight and a value, and a knapsack with a limited capacity, determine the combination of items that maximises the total value while not exceeding the capacity of the knapsack.

The items are represented as a list of tuples where each tuple contains the weight and value of an item. The knapsack capacity is given as an input to the program.

Each item has a quantity of one.

Task

Define a function *solve_knapsack_problem* that takes two parameters:

Name	Type	Example Input	Description
items	list of tuples	[(5, 2), (1, 1000), (100, 1), (25, 25), (2, 1000)]	[(weight, value), ...]
knapsack_capacity	int	5	

When called, the function should return the maximum value possible without going over the weight restriction.

Starter Code

```
def solve_knapsack_problem(
    items: list[tuple[int, int]], knapsack_capacity: int
) -> int:

    # Your implementation here

items = [(5, 2), (1, 1000), (100, 1), (25, 25), (2, 1000)]
max_weight = 5
print(solve_knapsack_problem(items, max_weight))
```

Examples

```
Inputs:
 - items: [(5, 2), (1, 1000), (100, 1), (25, 25), (2, 1000)]
 - knapsack_capacity: 5

Output: 2000

Inputs:
 - items: [(5, 2), (1, 1000), (100, 1), (25, 25), (2, 1000)]
 - knapsack_capacity: 0

Output: 0

Inputs:
 - items: []
 - knapsack_capacity: 5

Output: 0
```

PUZZLE 45.1 - BONUS

Task

Improve your implementation of puzzle 45 to solve the knapsack problem in *O(nW)* time complexity where n is the number of items and W is the capacity of the knapsack.

Starter Code

```
def solve_knapsack_problem(
    items: list[tuple[int, int]], knapsack_capacity: int
) -> int:

    # Your implementation here

items = [(5, 2), (1, 1000), (100, 1), (25, 25), (2, 1000)]
max_weight = 5
print(solve_knapsack_problem(items, max_weight))
```

Examples

```
Inputs:
 - items: [(5, 2), (1, 1000), (100, 1), (25, 25), (2, 1000)]
 - knapsack_capacity: 5

Output: 2000

Inputs:
 - items: [(5, 2), (1, 1000), (100, 1), (25, 25), (2, 1000)]
 - knapsack_capacity: 0

Output: 0

Inputs:
 - items: []
 - knapsack_capacity: 5

Output: 0
```

PUZZLE 46

Task

Define a function *ip_range_to_list* that takes one parameter:

Name	Type	Example Input	Description
input_ip_range	str	"192.255.255.0-192.255.255.255"	The string should be formatted as the start IP followed by a dash followed by the end IP

When called, the function should return a list of all the unique IP addresses within the range, including the start and end IPs. Each IP address should be represented as a string in the format "x.x.x.x". The IP addresses should be in numerical order.

To make the solution as time efficient as possible the solution should make use of bitwise operations. You should make use of two libraries: struct and socket.

Starter Code

```
import socket
import struct

def ip_range_to_list(input_ip_range: str) -> list[str]:
    # Your implementation here

print(ip_range_to_list("192.168.1.1-192.168.1.5"))
```

Examples

```
Input: "192.168.1.1-192.168.1.5"
Output: ["192.168.1.1",
         "192.168.1.2",
         "192.168.1.3",
         "192.168.1.4",
         "192.168.1.5"]

Input: "1.1.1.0-1.1.1.1"
Output: ["1.1.1.0", "1.1.1.1"]

Input: "192.255.255.0-192.255.255.0"
Output: ["192.255.255.0"]
```

PUZZLE 47

Task

Define a function *solve_maze* that takes three parameters:

Name	Type	Example Input	Description
maze	list of list of int	[[0, 1, 1], [0, 0, 1], [1, 0, 1]]	A 2-dimensional list consisting of walls (1) and pathways (0). Each inner list represents a row of the maze with each n’th element representing the n’th column.
start_pos	tuple of (int, int)	(0, 0)	Starting position coordinates in the format (x, y).
end_pos	tuple of (int, int)	(1, 2)	Ending position coordinates in the format (x, y).

When called, the function should return a boolean indicating if the maze is solvable. The maze is only solvable if there is a pathway (all 0’s) from the *start_pos* to the *end_pos*. The only valid moves in the maze are up, down, left and right - diagonal moves are not allowed.

Starter Code

```
def solve_maze(
    maze: list[list[int]],
    start_pos: tuple[int, int],
    end_pos: tuple[int, int]
) -> bool:
    # Your implementation here

start = (0, 0)
end = (1, 2)

solvable_maze = [
    [0, 1, 1],
    [0, 0, 1],
    [1, 0, 1]
]

print(solve_maze(solvable_maze, start, end))
```

Examples

```
# --- solvable maze example --- #
maze = [
    [1, 0, 1, 1, 1, 1, 1, 1, 1, 1],
    [1, 0, 0, 0, 0, 0, 0, 0, 0, 1],
    [1, 0, 1, 1, 1, 1, 1, 1, 0, 1],
    [1, 0, 1, 0, 0, 0, 0, 1, 0, 1],
    [1, 0, 1, 0, 1, 0, 0, 1, 0, 1],
    [1, 0, 1, 0, 1, 0, 0, 1, 0, 1],
    [1, 1, 0, 0, 1, 0, 1, 1, 0, 1],
    [1, 0, 0, 1, 0, 0, 0, 0, 0, 1],
    [1, 0, 1, 1, 1, 1, 1, 1, 1, 1],
    [1, 0, 1, 1, 1, 1, 1, 1, 1, 1]
]
start = (1, 0)
end = (1, 9)
def solve_maze(maze, start, end):
    ...

print(solve_maze(maze, start, end)) # Returns True
```

```
# --- unsolvable maze example --- #
maze = [
    [1, 0, 1, 1, 1, 1, 1, 1, 1, 1],
    [1, 0, 0, 0, 0, 0, 0, 0, 0, 1],
    [1, 0, 1, 1, 1, 1, 1, 1, 0, 1],
    [1, 0, 1, 0, 0, 0, 0, 1, 0, 1],
    [1, 0, 1, 0, 1, 0, 0, 1, 0, 1],
    [1, 0, 1, 0, 1, 0, 0, 1, 0, 1],
    [1, 1, 0, 1, 1, 0, 1, 1, 0, 1],
    [1, 0, 0, 1, 0, 0, 0, 0, 0, 1],
    [1, 0, 1, 1, 1, 1, 1, 1, 1, 1],
    [1, 0, 1, 1, 1, 1, 1, 1, 1, 1]
]

start = (1, 0)
end = (1, 9)

def solve_maze(maze, start, end):
    ...

print(solve_maze(maze, start, end)) # Returns False
```

PUZZLE 48

Background

In computer science a binary tree is a tree in which every node has zero, one or two children. If a node has children they are denoted by “left” or “right” children. A common action is to traverse the binary tree which is where we visit each node in a certain order and output the nodes values.

There are three main methods of traversal: pre-order, in-order and post-order. In this puzzle we’re going to focus on in-order traversal, which works as follows:

1. Traverse the left subtree recursively by calling in-order traversal on the left child.
2. Visit the root node.
3. Traverse the right subtree recursively by calling in-order traversal on the right child.

Task

Define a function *traverse_inorder* that takes one parameter:

Name	Type	Example Input
root_node	TreeNode	TreeNode(4, two, six)

A TreeNode is a simple class we've created that has three properties:

Name	Type	Example
val	int	1
left_node	TreeNode	TreeNode(2, None, None)
right_node	TreeNode	TreeNode(3, None, None)

When called, the function should return an iterable containing the result of the in-order traversal.

Starter Code

```
from __future__ import annotations
from typing import Iterator

class TreeNode:
    def __init__(
        self, val: int = 0,
        left: TreeNode | None = None,
        right: TreeNode | None = None,
    ) -> None:
        self.val = val
        self.left = left
        self.right = right

def traverse_inorder(root_node: TreeNode | None)
    -> Iterator[int]:
    # Your implementation here

one = TreeNode(1, None, None)
three = TreeNode(3, None, None)
two = TreeNode(2, one, three)
five= TreeNode(6, None, None)
four = TreeNode(4, two, five)

for node in traverse_inorder(four):
    print(node)
```

Examples

```
Input:
     4
   /   \
  2     6
 / \   / \
1   3 5   7
Output: [1, 2, 3, 4, 5, 6, 7]

Input:
  2
 / \
1   3
     \
      4
Output: [1, 2, 3, 4]
```

PUZZLE 48.1 - BONUS

Background

The in-order traversal algorithm visits each node exactly once, so the time complexity of the algorithm is *O(n)* and the space complexity is *O(h)* where h is the height of the tree.

It is possible to perform an in-order traversal of a binary tree with constant space complexity *O(1)* using the morris traversal algorithm. The algorithm works as follows:

1. Initialise the current node to the root of the tree.
2. If the current node has no left child, yield its value and move to its right child.
3. If the current node has a left child, find its in-order predecessor (i.e. the rightmost node in its left subtree):
    a. If the predecessor has no right child, set its right child to the current node and move to the left child of the current node.
    b. Else if the predecessor has a right child, reset its right child to None, yield the value of the current node, and move to the current node's right child.

Task

Write a python solution to implement the morris traversal algorithm. The inputs and outputs should be exactly the same as puzzle 48.

Starter Code

```
from __future__ import annotations
from typing import Iterator

class TreeNode:
    def __init__(
        self,
        val: int = 0,
        left: TreeNode | None = None,
        right: TreeNode | None = None,
    ) -> None:
        self.val = val
        self.left = left
        self.right = right

def morris_traverse_inorder(root_node: TreeNode | None)
    -> Iterator[int]:
    # Your implementation here

one = TreeNode(1, None, None)
three = TreeNode(3, None, None)
two = TreeNode(2, one, three)
five = TreeNode(5, None, None)
seven = TreeNode(7, None, None)
six = TreeNode(6, five, seven)
four = TreeNode(4, two, six)

for node in morris_traverse_inorder(four):
    print(node)
```

PUZZLE 49

Background

The climbing stairs problem is a common problem in computer science and mathematics. The problem involves finding the number of different ways someone can climb a set of stairs, where each step can be taken one at a time or two at a time.

The only input to the problem is the number of stairs in the staircase. For example, if the number of stairs in the staircase is 3, we have the following ways to climb the stairs:

- 1 step, 1 step, 1 step
- 1 step, 2 steps
- 2 steps, 1 step

Therefore, the answer to the problem would be 3.

Task

Define a function *solve_climbing_stairs_problem* that takes one parameter:

Name	Type	Example Input
total_stairs	int	4

When called, the function should return the number of possible combinations there are to climb the stairs.

Caveat:

- The solution for this problem should make use of dynamic programming instead of simple recursion.

Starter Code

```
def solve_climbing_stairs_problem(total_stairs: int) -> int:
    # Your implementation here

print(solve_climbing_stairs_problem(4))
```

Examples

```
Input: 4
Output: 5

Input: 10
Output: 89

Input: 0
Output: 0
```

PUZZLE 49.1 - BONUS

Task

Your solution from puzzle 49 should output the total number of combinations possible, however it doesn't say what these combinations are.

Improve your solution so that the different combinations the staircase can be climbed in are returned.

Caveat:

- The solution for this problem should make use of dynamic programming instead of simple recursion.

Starter Code

```
def solve_climbing_stairs_problem_with_output(
    total_stairs: int
) -> list[list[int]]:
    # Your implementation here

print(solve_climbing_stairs_problem_with_output(4))
```

Examples

```
Input: 4
Output: [[1, 1, 1, 1], [2, 1, 1], [1, 2, 1], [1, 1, 2], [2, 2]]

Input: 10
Output: [[1, 1, 1, 1, 1, 1, 1, 1, 1, 1], [2, 1, 1, 1, 1, 1, 1, 1,
1], ...]

Input: 0
Output: []
```

PUZZLE 49.2 - BONUS

Task

Modify your solution from puzzle 49 to allow for 1, 2 or 3 stairs to be climbed at once instead of just 1 or 2.

Caveat:

- The solution for this problem should make use of dynamic programming instead of simple recursion.

Starter Code

```
def solve_climbing_stairs_problem_with_three_steps_allowed(
    total_stairs: int
) -> int:
    # Your implementation here

print(
    solve_climbing_stairs_problem_with_three_steps_allowed(4)
)
```

Examples

```
Input: 4
Output: 7

Input: 10
Output: 274

Input: 0
Output: []
```

PUZZLE 50

Background

Similar to puzzle 49, the coin change problem is a classic problem in computer science. The problem is defined as follows: given a set of coin denominations and a target amount, find the minimum number of coins needed to make the target amount.

Task

Define a function *solve_coin_change_problem* that takes two parameters:

Name	Type	Example Input
coin_values	list of int	[1, 6]
target_amount	int	6

When called, the function should return the minimum number of coins needed to make the target amount. The function should return -1 if it is not possible to reach the target.

We can assume there are an infinite number of each coin.

Caveat:

- Your solution should utilise dynamic programming to be as efficient as possible.

Starter Code

```
def solve_coin_change_problem(coin_values: list[int],
target_amount: int) -> int:
    # Your implementation here

print(solve_coin_change_problem([1, 6], 6))
```

Examples

```
Inputs:
 - coin_values: [1, 6]
 - target_amount: 6

Output: 1

Inputs:
 - coin_values: [1, 2]
 - target_amount: 6

Output: 3

Inputs:
 - coin_values: [2, 1]
 - target_amount: 13

Output: 7
```

FUN PUZZLES

Welcome to the fun puzzles section! Puzzles in this section are designed to be a bit different, usually making use of external libraries that you may or may not have experience in. If you don’t have experience using a specific library, don’t skip the puzzle! It’s a great chance to learn something new.

REVERSE DNS LOOKUP

Background

DNS (Domain Name System) is a system that translates human-readable domain names, such as example.com, into the IP addresses that computers use to identify each other on the internet. It works as a directory service, allowing users to access websites and other online resources by entering a domain name in their web browser. The domain name is then resolved into an IP address by DNS servers.

Task

Define a function *reverse_dns_lookup* that takes one parameter:

Name	Type	Example Input
ip_address	str	“8.8.8.8”

When called, the function should return the domain name that maps to that IP address using PTR DNS records. The function should return *None* if the IP address does not have a PTR record.

You should make use of the *gethostbyaddr* function found in the socket library.

Starter Code

```
import socket

def reverse_dns_lookup(ip_address: str) -> str:
    # Your implementation here

print(reverse_dns_lookup("8.8.8.8"))
```

Examples

Note: there’s a small chance the outputs could change, if you get a slightly different response for these inputs the response is still likely to be correct.

```
Input: "8.8.8.8"
Output: "dns.google"

Input: "208.67.222.222"
Output: "dns.umbrella.com"

Input: "1.1.1.1"
Output: "one.one.one.one"
```

RUSSIAN DOLLS

Background

Russian dolls, also known as Matryoshka dolls, are a set of wooden dolls of decreasing sizes that are placed one inside the other. The largest doll in the set typically opens up to reveal a smaller doll inside, which in turn contains an even smaller doll, and so on, until there are no dolls remaining.

Task

Define a function *unpack_dolls* that takes one parameter:

Name	Type	Example Input
doll	RussianDoll	RussianDoll(4, 3, "purple")

The parameter is of type *RussianDoll*, this is a simple class that contains three properties:

- Size - the size of the doll.
- Colour - the colour of the doll.
- Child doll - another *RussianDoll* object that this doll has inside of it, or *None*.

The *RussianDoll* class is given in the starter code below. When the *unpack_dolls* function is called it should have the same output as the example below.

Example

Given the following russian dolls:

Doll Size	Colour	Child Size
5	Grey	4
4	Purple	3
3	Green	2
2	Blue	1
1	Red	None

The program should have the following output to the console:

```
Unpacking a grey doll of size: 5 with 4 nested dolls inside.
Unpacking a purple doll of size: 4 with 3 nested dolls inside.
Unpacking a green doll of size: 3 with 2 nested dolls inside.
Unpacking a blue doll of size: 2 with 1 nested dolls inside.
Unpacking a red doll of size: 1 with 0 nested dolls inside.
Total number of dolls in the set: 5
```

Starter Code

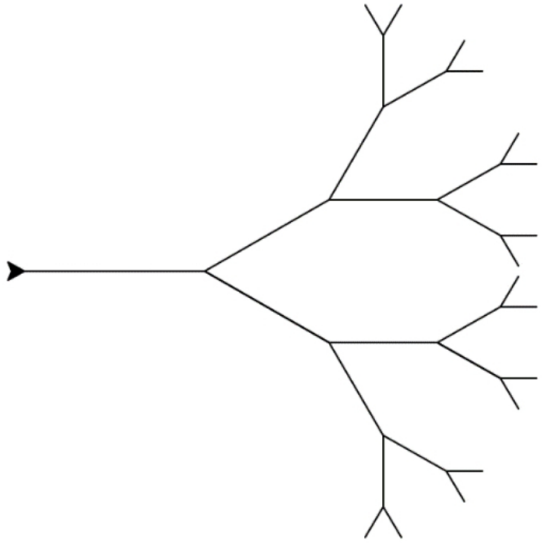
```
from __future__ import annotations

class RussianDoll:
    def __init__(
            self,
            size: int,
            colour: str,
            child_doll: RussianDoll | None = None
    ) -> None:

        self.size = size
        self.colour = colour
        self.child_doll = child_doll

def unpack_dolls(doll: RussianDoll) -> int:
    # Your implementation here

doll_size_one = RussianDoll(1, "red", None)
doll_size_two = RussianDoll(2, "blue", doll_size_one)
doll_size_three = RussianDoll(3, "green", doll_size_two)
doll_size_four = RussianDoll(4, "purple", doll_size_three)
doll_size_five = RussianDoll(5, "grey", doll_size_four)
unpack_dolls(doll_size_five)
```

FRACTAL TREE

Background

A fractal tree is a tree-like structure that exhibits self-similar patterns at different scales, created by recursively applying a set of rules or algorithms to generate a branching pattern that looks similar to the structure of a real tree.

An example of a fractal tree can be seen to the below:



Task

Define a function *draw_tree* that takes one parameter:

Name	Type	Example Input
depth	int	5

When called, the function should display a fractal tree.

You should make use of the turtle graphics library that is built into python. If you’re not familiar with the turtle graphics library, it provides a simple and user-friendly way to create graphics and visualisations using a virtual "turtle." The turtle is a cursor that can be controlled by a set of instructions to draw shapes and lines on a canvas.

Starter Code

```
import turtle

def draw_tree(depth: int) -> None:
    # Your implementation here

draw_tree(5)
turtle.exitonclick()
```

Example

The tree depth of the tree above was depth 5. Your depth 5 tree should look exactly the same.

PING PONG

Task

Create a basic two player ping pong game using the pygame library. The game should have two paddles, one ball and implement a scoring system.

Note: challenges like this are most fun when letting the imagination run wild, so I've left the description purposely vague. There are no right or wrong answers here, as long as the final program represents some form of ping pong!

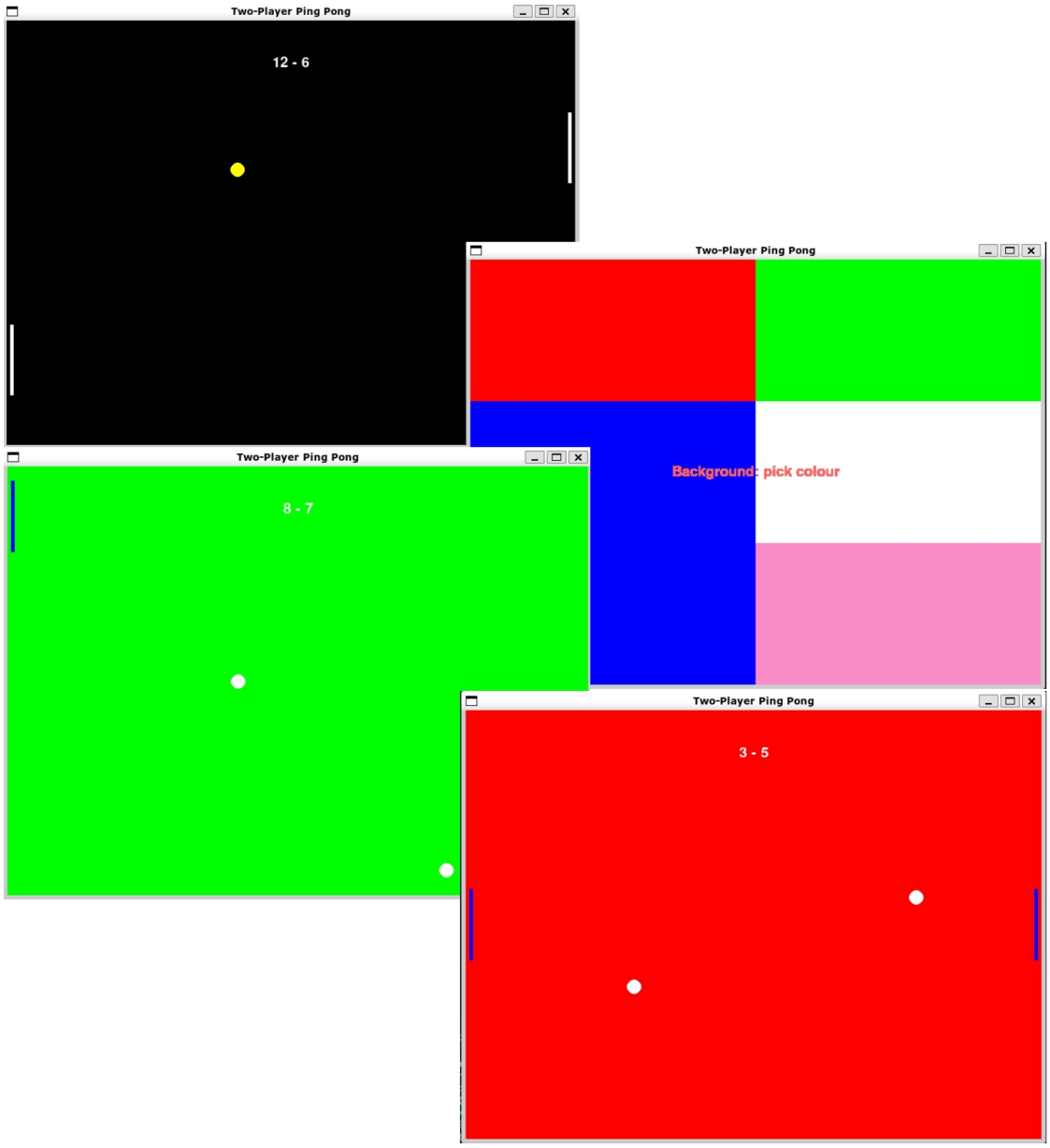
Bonus challenges:

- Allow the players to choose what colour they want the background, ball and paddles to be (*bonus-bonus: implement a visual colour picker*).
- Make the ball change colour every second.
- Allow an arbitrary number of balls to be in play at once.

Hint: the bonus challenges are easier to implement using object oriented programming (...think about having multiple balls at once).

Examples

As this is a creative puzzle each solution may well look completely different, and that's fine! Below are some examples of what a potential solution could look like.



DRAWING TOOL

Task

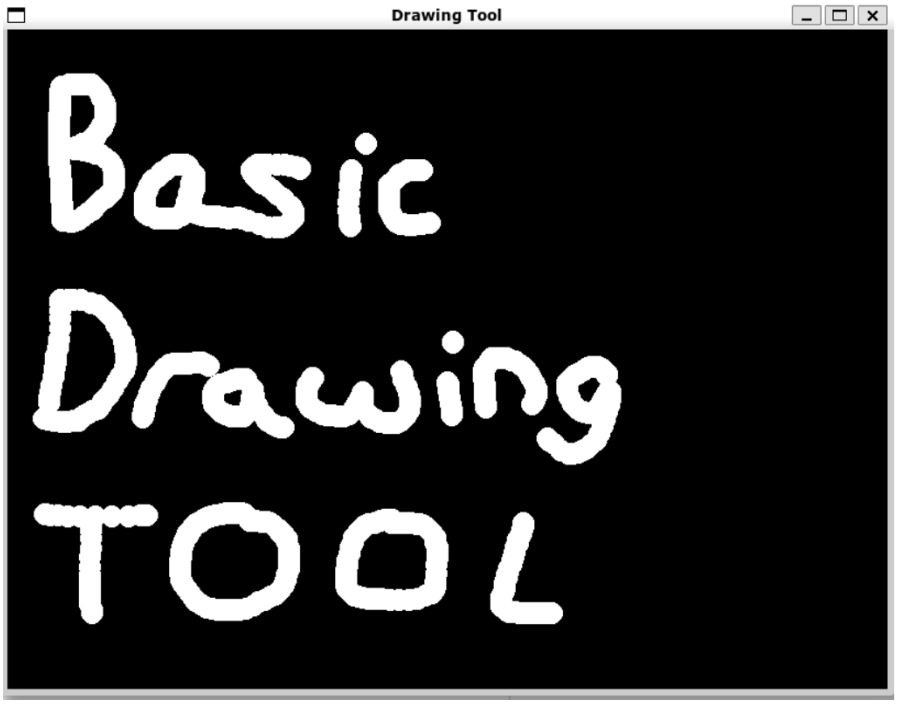
Create a basic drawing tool using the pygame library where a black canvas is displayed and the user can draw white coloured paint by clicking on the canvas.

Once the above application is complete we have a good base to add additional functionality to, here are some ideas as bonus challenges:

- Implement a sidebar that allows the user to:
    - Select a different drawing colour
    - Clear the canvas
    - Save their drawing to disk
    - Change the brush thickness
- Creating a line drawing tool so the user can choose between the "paint brush" and "line drawing tool”

…but don’t stop there! These are just starter ideas and you can take your application in any direction you like!

Examples


Drawing Tool
Basic
Drawing
TOOL


Drawing Tool
Bonus
Drawing
TOOL
Paint Brush
Line Tool
Clear Canvas
Save Image
- 10 +


AMAZING
SAVED
IMAGE

CHALLENGE PUZZLES: HINTS

Puzzle 1

You will need to make use of a for loop to loop through each string in *input_strs*. Inside of the loop you will need to check if the current word contains the letter “a”, this can be done using an if-statement.

To check if a string contains a letter you can use the “in” syntax e.g.

- *"a" in “banana” = True*
- *“a” in “pop“ = False*

Puzzle 2

The first step to this puzzle is to sum together the two numbers.

The second step is to check if the resulting sum of the two numbers is less than 50. We can check this using an if statement, if the if statement is true (the sum is less than 50) then we return *None*. Else, we can return the result of the sum.

Puzzle 3

This puzzle can be broken down into a few steps…

The first step is to filter out odd integers, to do this we should make use of the modulo (%) operator. This operator works by giving us the remainder of a division e.g.

- *5 % 2 = 1*
- *4 % 2 = 0*

We can pair this operator with list comprehension to create a new list of only even integers. If you're not familiar with list comprehension it has the following syntax:

- *[<variable name> for <variable name> in <list> if <condition>]*
- The list comprehension: *[my_number for my_number in [1, 2, 3] if my_number % 3 == 0]* would result in *[3]* as *1 % 3 = 1* and *2 % 3 = 2* which makes our condition False.

We can then make use of Python's built in sum function. This function does what it says on the tin, sum's together all the elements of a list.

So if we create a new list with only even elements, then sum that list together and return the result… We're there!

Puzzle 4

This puzzle is a combination of the techniques learnt in puzzles 1 and 3.

We need to use a conditional list comprehension to loop through each character of our input string. The condition of the list comprehension should check if the current character is a vowel (hint: you can make use of the “in” syntax you used in puzzle 1). Once we have our resulting list, we can convert it to a string using the built in *join* function.

Puzzle 5

Let’s start off by asking ourselves what’s the longest string that we know of, before looking at the input strings? An empty string! The idea of this solution will be as follows:

- Define an empty string as our “longest string”
- Loop through each of the input strings and if the current string is longer than our “longest string” variable, then replace it.
- At the end, the string that’s left in our “longest string” variable will be the longest!

You can use the built in *len* function to get the length of a string.

Puzzle 6

I’m purposely going to leave the hints for this one short, the only hint I will give is to look at the techniques used in puzzles 3, 4 and 5.

Puzzle 7

Your solution here should make use of Python slicing. Python slicing is where we have square brackets after a list with the following syntax inside of the brackets: *start:stop:step*.

For example, if we take a list of [1,2,3,4,5,6] and a slice condition of [0:4:2] we would get a result of [1, 3]. This is because our slice condition is saying start at index 0 of the list (inclusive), stop at index 4 of the list (exclusive) with a step size of 2.

The number at index 0 of the list is 1, we then step forward 2 positions and take the number at index 2 of the list which is 3, then we step forward 2 more positions and that takes us past our stop condition so we're finished.

So what numbers would we need in our slicing condition to reverse the list? Hint: the step value can be negative!

Puzzle 8

For this puzzle you will need to make use of list comprehension and the built in *isinstance* function. This function works as follows:

- *isinstance(variable, type)* - if the type of the variable is equal to type then *True* is returned e.g.
    - *isinstance(“a string”, str) = True*
    - *isinstance(“a string”, int) = False*

Puzzle 9

Our solution should start off by initialising an empty list where we're going to store our result.

Then we need to loop through each character in the input string:

1. If the character is not a space, use the *morse_dict* to convert it to morse code and append it to our resulting list.

2. If the character is a space then we can append a forward slash to our resulting list.

Once we have our resulting list, we can join it together into a string using a similar technique to what we used in puzzle 4.

Puzzle 10

Let’s break down our puzzle into steps:

1. Does the list contain at least two numbers? If not, return *None*.
2. Find the second largest number:
    a. Find and remove the maximum number from the list (hint: use the built in *max* function)
    b. We now know that in the remaining list the largest number is the second largest number of the original list. That means we can make use of the max function to find the largest number again, and return the result.

Puzzle 11

There are multiple ways to solve this solution however the most straightforward would be to make use of Python's f-strings. In Python, f-strings (formatted string literals) provide a concise and convenient way to embed expressions inside string literals, allowing you to create strings with dynamic content. They were introduced in Python 3.6 as a way to simplify string formatting compared to older methods like using the % operator or the *.format()* method.

Take a look at the Python f-string documentation and see what you can find!