Introduction to Scientific Programming in Python

Pablo Peñas (pablo.penas@virtuscollege.es)

June 14, 2024

Contents

Chapter 1: Variables and logic

1.1 Variable types

1.1.1 Strings (`str`)

Strings are sequences of characters enclosed in quotes.

```
greeting = "Hello, world!"
print(greeting)

# Print variable type
print(type(greeting))
```

Output:

```
Hello, world!
<class 'str'>
```

```
greeting = "Welcome,"
school = "Virtus"
print(greeting, "you are at", school)
```

Output:

```
Welcome, you are at Virtus
```

Methods on strings

```
text = "Hello, World!"
print("Uppercase:", text.upper())
print("Lowercase:", text.lower())
print("Replace:", text.replace("World", "Potato"))
```

Output:

```
Uppercase: HELLO, WORLD!
Lowercase: hello, world!
Replace: Hello, Potato!
```

Operations on strings

```
quantity =  "Acceleration"
value = "9.81"
units = "m/s^2"
equation = quantity + " = " + value + " " + units
print(equation)
```

Output:

```
Acceleration = 9.81 m/s^2
```

```
lesson = "I will not waste chalk. \n"
print(6*lesson)
```

Output:

```
I will not waste chalk.
I will not waste chalk.
I will not waste chalk.
I will not waste chalk.
I will not waste chalk.
I will not waste chalk.
```

Substrings and string slicing

```
text = "Hello, world!"
print("First letter:", text[0])
print("Second letter:", text[1])
print("Last letter:", text[-1])
print("Substring:", text[7:12])

length = len(text)
print("Length:", length)
print("Last letter:", text[length-1])
```

Output:

```
First letter: H
Second letter: e
Last letter: !
Substring: world
Length: 13
Last letter: !
```

1.1.2 Integers (`int`)

Integers are whole numbers without a decimal point.

```
score = -3
year = 2024
print("Integer examples:", score, year)

# Print variable type
print(type(score))
print(type(year))
```

Output:

```
Integer examples: -3 2024
<class 'int'>
<class 'int'>
```

Operations on integers

```
# Basic arithmetic operations
a = 10
b = 3
print("Addition:", a + b)
print("Subtraction:", a - b)
print("Multiplication:", a * b)
print("Division:", a / b, "is of type", type(a/b))
print("Floor (integer) Division:", a // b, "is of type", type(a//b))
print("Modulus:", a % b)
print("Exponentiation:", a ** b)
```

Output:

```
Addition: 13
Subtraction: 7
Multiplication: 30
Division: 3.3333333333333335 is of type <class 'float'>
Floor (integer) Division: 3 is of type <class 'int'>
Modulus: 1
Exponentiation: 1000
```

1.1.3 Floats (`float`)

Floats are numbers that have a decimal point.

```
temperature = 16.6
mass = 77.2
velocity = -4.0
print("Float examples:", temperature, mass, velocity)
```

Output:

```
Float examples: 16.6 77.2 -4.0
```

```
# Scientific notation
mega = 1e6
milli = 1e-3
print(mega, type(mega))
print(milli, type(milli))

force = 1.85e5
mass = 1.4e-3
print("force (N)", force)
print("mass (kg)", mass)
```

Output:

```
1000000.0 <class 'float'>
0.001 <class 'float'>
force (N) 185000.0
mass (kg) 0.0014
```

Operations on floats

```
# Basic arithmetic operations
a = 10.0
b = 3.0
print("Addition:", a + b)
print("Subtraction:", a - b)
print("Multiplication:", a * b)
print("Division:", a / b, "is of type", type(a/b))
print("Floor (integer) Division:", a // b, "is of type", type(a//b))
print("Modulus:", a % b)
print("Exponentiation:", a ** b)
```

Output:

```
Addition: 13.0
Subtraction: 7.0
Multiplication: 30.0
Division: 3.3333333333333335 is of type <class 'float'>
Floor (integer) Division: 3.0 is of type <class 'float'>
Modulus: 1.0
Exponentiation: 1000.0
```

Evaluation of formulas eg:

$$x = \left(\frac{a+b}{a-b} + b^2\right)^{-1/3}$$

```
a = 10.0
b = 1.5
x = ((a+b)/(a-b)+b**2)**(-1/3)
print(x)
```

Output:

```
0.6523003468435792
```

Operations can combine floats and integers

```
# Basic arithmetic operations
a = 7.5
b = 3
print("Addition:", a + b)
print("Subtraction:", a - b)
print("Multiplication:", a * b)
print("Division:", a / b, "is of type", type(a/b))
print("Floor (integer) Division:", a // b, "is of type", type(a//b))
print("Modulus:", a % b)
print("Exponentiation:", a ** b)
```

Output:

```
Addition: 10.5
Subtraction: 4.5
Multiplication: 22.5
Division: 2.5 is of type <class 'float'>
Floor (integer) Division: 2.0 is of type <class 'float'>
Modulus: 1.5
Exponentiation: 421.875
```

1.1.4 Booleans (`bool`)

Booleans represent one of two values: True or False.

```
is_sunny = True
is_raining = False
print("Boolean examples:", is_sunny, is_raining)
print(type(is_sunny))
print(type(is_raining))
```

Output:

```
Boolean examples: True False
<class 'bool'>
<class 'bool'>
```

```
# The not function
is_sunny = True
is_raining = False
print(not(is_sunny), not(is_raining))
```

Output:

```
False True
```

Comparison operators

```
# Booleans are often the result of comparisons and logical operations.
x = 5

condition = x > 3
print("Is x greater than 3?", condition)

print("Is x equal to 3?", x == 3)
print("Is x not equal to 3?", x != 3)
print("Is x less than or equal to 5?", x <= 5)
print("Is x between 1 and 6?", 1<x <6)
```

Output:

```
Is x greater than 3? True
Is x equal to 3? False
Is x not equal to 3? True
Is x less than or equal to 5? True
Is x between 1 and 6? True
```

Logical operators

```python
a = True
b = False

print("a and b:", a and b) # False
print("a or b:", a or b)   # True
print("not a:", not a)     # False
```

Output:

```
a and b: False
a or b: True
not a: False
```

1.2 Casting

Converting to Integer: `int()`

```python
# Converting a float to an integer
pi = 3.14159
pi_int = int(pi)
print(f"Float {pi} cast to Integer: {pi_int}")

# Converting a string to an integer
number_str = "42"
number_int = int(number_str)
print(f"String '{number_str}' cast to Integer: {number_int}")
```

Output:

```
Float 3.14159 cast to Integer: 3
String '42' cast to Integer: 42
```

Converting to Float: `float()`

```python
# Converting an integer to a float
age = 30
age_float = float(age)
print(f"Integer {age} cast to Float: {age_float}")

# Converting a string to a float
height_str = "1.75"
height_float = float(height_str)
print(f"String '{height_str}' cast to Float: {height_float}")
```

Output:

```
Integer 30 cast to Float: 30.0
String '1.75' cast to Float: 1.75
```

Converting to String: `str()`

```python
# Converting an integer to a string
score = 100
score_str = str(score)
```

```python
# Converting a float to a string
scale = 3.6e-5
scale_str = str(scale)

# Converting a bool to a string
condition = True
condition_str = str(condition)

print(score_str, scale_str, condition_str)
```

Output:

```
100 3.6e-05 True
```

Converting to Bool: `bool()`

```python
# Converting an integer to a boolean
zero = 0
one = 1
neg3 = -3
print(f"Integer {zero} cast to Boolean: {bool(zero)}")
print(f"Integer {one} cast to Boolean: {bool(one)}")
print(f"Integer {neg3} cast to Boolean: {bool(neg3)}")
```

Output:

```
Integer 0 cast to Boolean: False
Integer 1 cast to Boolean: True
Integer -3 cast to Boolean: True
```

```python
# Converting a string to a boolean
empty_str = ""
non_empty_str = "Hello"
print(f"String '{empty_str}' cast to Boolean: {bool(empty_str)}")
print(f"String '{non_empty_str}' cast to Boolean: {bool(non_empty_str)}")
```

Output:

```
String '' cast to Boolean: False
String 'Hello' cast to Boolean: True
```

```python
# Converting a float to a boolean
zerof = 0.0
small_neg = -0.00001
print(f"Flot {0.0} cast to Boolean: {bool(zerof)}")
print(f"Float {small_neg} cast to Boolean: {bool(small_neg)}")
```

Output:

```
Flot 0.0 cast to Boolean: False
Float -1e-05 cast to Boolean: True
```

1.3 Updating variables

Updating integers

```python
# Integer
x = 5
print(x)
x = x + 10
print(x)
x += 5
print(x)
```

```
x -= 2
print(x)
```

Output:

```
5
15
20
18
```

Updating floats

```
# Float
y = 3.14
print(y)
y = y * 2
print(y)
y *= 2
print(y)
y /= 10
print(y)
```

Output:

```
3.14
6.28
12.56
1.256
```

Updating strings

```
# String
s = "Hello"
print(s)
s = s + " World"
print(s)
s += "!!"
print(s)
```

Output:

```
Hello
Hello World
Hello World!!
```

1.4 Logic statements

1.4.1 `if` and `if else` statements

```
x = 10
y = 20
z = 15

# If statement

if x < y:
  print("x is less than y")


# If-else statement
if x > z:
  print("x is greater than z")
```

```
else:
  print("x is less than z")
```

Output:

```
x is less than y
x is less than z
```

```
# Negating a condition
if not (x > y):
    print("x is not greater than y")

# Combining with and operator
if x < y and y == 20:
    print("x is less than y and y equals 20")

# Combining with or operator
if x > y or x < 11:
    print("x is greater than y or x < 11")
```

Output:

```
x is not greater than y
x is less than y and y equals 20
x is greater than y or x < 11
```

1.4.2 `if elif else` statements

```
# Grading system based on marks
marks = 67

if marks >= 80:
    print("Grade: A")
elif marks >= 70:
    print("Grade: B")
elif marks >= 60:
    print("Grade: C")
elif marks >= 50:
    print("Grade: D")
else:
    print("Grade: U")
```

Output:

```
Grade: C
```

1.5 The `input()` function

```
name = input("Enter your name: ")
print(f"Hello, {name}!")
print(f"Input is always of type {type(name)}")
```

Output:

```
Hello, Pablo!
Input is always of type <class 'str'>
```

Converting to other types

```
height = float(input("Enter your height in meters: "))
age = int(input("Enter your age in years: "))
print(f"You are {height} m, age {age}")
```

```
current_year=2024
target_year = 2050
print(f"In {target_year} you will be age {target_year-current_year+age}")
```

Output:

```
You are 1.79 m, age 34
In 2050 you will be age 60
```

1.6 f-Strings

f-Strings (Formatted String Literals) provide a concise and convenient way to embed expressions inside string literals using curly braces {}.

```
name = "Pablo"
age = 34
greeting = f"Hello, {name}! You are {age} years old."
print(greeting)
```

Output:

```
Hello, Pablo! You are 34 years old.
```

Concatenating f-strings

```
# Concatenating f-strings
name = "Pablo"
age = 34
occupation = "teacher"

profile = f"Name: {name}\n" f"Age: {age}\n" f"Occupation: {occupation}"
print(profile)
```

Output:

```
Name: Pablo
Age: 34
Occupation: teacher
```

Numerical evaluation and formatting

```
# Evaluation inside f-strings
length = 5.41
width = 3.27
result = f"The area of the rectangle is {length * width} square units."
print(result)
```

Output:

```
The area of the rectangle is 17.6907 square units.
```

```
# Formatting numbers
length = 5.413
width = 3.17
area = length * width
print(f"Area = {area} sq.")
print(f"Area = {area:.0f} sq.")
print(f"Area = {area:.1f} sq.")
print(f"Area =  {area:.2f} sq.")
```

Output:

```
Area = 17.15921 sq.
Area = 17 sq.
Area = 17.2 sq.
```

```
Area =  17.16 sq.
```

1.7 Exercise 1.1 The state of water

Write a program that takes the temperature of water (in Celsius) as `input` and prints whether the state of water at that temperature is solid, liquid, or gas.

Solution

```python
# Temperature of water in Celsius
temperature = float(input("Enter the temperature of water in Celsius: "))

# Determine the state of water
if temperature <= 0:
    print("Solid (Ice)")
elif temperature < 100:
    print("Liquid (Water)")
else:
    print("Gas (Steam)")
```

Output:

```
Liquid (Water)
```

1.8 Exercise 1.2 BMI classification

Write a program that calculates the and prints the BMI category.

The Body Mass Index (BMI) of the person with weight W (in kg) and height H (in meters) is defined as

$$\text{BMI} = \frac{W}{H^2}$$

Write a program that ouputs the BMI value to 1 decimal place and the BMI category:

- "Underweight" if BMI < 18.5,
- "Normal weight" if BMI is between 18.5 and 24.9,
- "Overweight" if BMI is between 25 and 29.9, and
- "Morbidly obese" if BMI is 30 or above.

Solution

```python
# Input weight and height
weight = float(input("Enter your weight in kg: "))
height = float(input("Enter your height in meters: "))

# Calculate BMI
bmi = weight / (height ** 2)

# Determine the BMI category
if bmi < 18.5:
    category = "Underweight"
elif 18.5 <= bmi < 25:
    category = "Normal weight"
elif 25 <= bmi < 30:
    category = "Overweight"
else:
    category = "Morbidly obese"

# Print the result
```

```
print(f"Your BMI is {bmi:.2f}")
print(f"Category: {category}")
```

Output:

```
Your BMI is 23.41
Category: Normal weight
```

1.9 Exercise 1.3 The Mach number

The **Mach number** M is a dimensionless quantity in fluid dynamics that represents the ratio of the speed of an object moving relative to a fluid (typically air), v, to the speed of sound in that fluid, c.

$$M = \frac{v}{c}$$

Flight regimes

- **Subsonic**: $M < 0.95$. The object's speed is less than the speed of sound.
- **Transonic**: $0.95 \leq M < 1.0$. The object's speed is approximately equal to the speed of sound.
- **Supersonic**: $1.0 \leq M < 5$. The object's speed is greater than the speed of sound.
- **Hypersonic**: $M \geq 5$. The object's speed is much greater than the speed of sound.

The speed of sound in air varies with temperature according to

$$c = \sqrt{\gamma RT/M}$$

where

- $\gamma = 1.4$ (ratio of specific heats for air)
- $R = 8.314$ J/(mol·K) (universal gas constant)
- $M = 0.029$ kg/mol (molar mass of air)
- T is the temperature in Kelvin

Write a program that takes the speed of an object (in kilometers per hour) and the temperature of the air (in Celsius) as inputs, calculates the speed of sound in air at that temperature, computes the Mach number, and prints the Mach number (to 2 decimal places) and whether the object is moving at subsonic, transonic, supersonic, or hypersonic speed.

Test it for

(i) A bullet travelling at 3000 km/h at 20 °C

(ii) A commercial airliner cruising at 900 km/h at -50 °C.

Solution

```
import math

# Constants
gamma = 1.4
R = 8.314  # J/(mol·K)
M = 0.029  # kg/mol

# Input speed and temperature
v = float(input("Enter the speed of object in km/h: "))
v = v / 3.6
temp = float(input("Enter the temperature of the air in Celsius: "))
temp = temp+273.15

# Speed of sound in air at the given temperature
c = (gamma * R * temp / M)**0.5
```

```python
# Calculate Mach number
mach = v / c

# Determine the speed regime
if mach < 0.95:
    regime = "Subsonic"
elif 0.95 <= mach < 1.0:
    regime = "Transonic"
elif 1.0 <= mach < 5:
    regime = "Supersonic"
else:
    regime = "Hypersonic"

# Print the result
print(f"Mach number: {mach:.2f}")
print(f"Speed classification: {regime}")
```

Output:

```
Mach number: 2.43
Speed classification: Supersonic
```

Chapter 2: Python as a scientific calculator

2.1 Basic arithmetic operations

```
# Define operands
a = 15
b = 4

# Addition
addition = a + b
print("Addition:", addition)

# Subtraction
subtraction = a - b
print("Subtraction:", subtraction)

# Multiplication
multiplication = a * b
print("Multiplication:", multiplication)

# Division
division = a / b
print("Division:", division)

# Integer Division
integer_division = a // b
print("Integer Division:", integer_division, "is", type(a//b))

# Modulus
modulus = a % b
print("Modulus:", modulus)  # Output: 3

# Exponentiation
exponentiation = a ** b
print("Exponentiation:", exponentiation)
```

Output:

```
Addition: 19
Subtraction: 11
Multiplication: 60
Division: 3.75
Integer Division: 3 is <class 'int'>
Modulus: 3
Exponentiation: 50625
```

```
# Define operands
a = -15.2
b = 4.5

# Division
print("Division:", a / b)

# Integer Division
print("Integer division:", a // b, "is", type(a//b))

# Modulus
```

```
print("Modulus:", a % b)
```

Output:

```
Division: -3.3777777777777778
Integer Division: -4.0 is <class 'float'>
Modulus: 2.8000000000000007
```

2.2 The `math` library

```
import math

# Basic arithmetic
print("Absolute value of -7:", math.fabs(-7))
print("Square root of 16:", math.sqrt(16))
print("2 raised to the power 3:", math.pow(2, 3))
```

Output:

```
Absolute value of -7: 7.0
Square root of 16: 4.0
2 raised to the power 3: 8.0
```

```
# Trigonometry
angle = math.pi / 4  # 45 degrees in radians
print("Sine of 45 degrees:", math.sin(angle))
print("Cosine of 45 degrees:", math.cos(angle))
print("Tangent of 45 degrees:", math.tan(angle))
```

Output:

```
Sine of 45 degrees: 0.7071067811865475
Cosine of 45 degrees: 0.7071067811865476
Tangent of 45 degrees: 0.9999999999999999
```

```
# Logarithms and exponentials
print("Natural logarithm of 2:", math.log(2))
print("Base-10 logarithm of 100:", math.log10(100))
print("e raised to the power 2:", math.exp(2))
```

Output:

```
Natural logarithm of 2: 0.6931471805599453
Base-10 logarithm of 100: 2.0
e raised to the power 2: 7.38905609893065
```

```
# Constants
print("Value of pi:", math.pi)
print("Value of e:", math.e)
```

Output:

```
Value of pi: 3.141592653589793
Value of e: 2.718281828459045
```

```
# Special functions
print("Factorial of 5:", math.factorial(5))
print("GCD of 48 and 18:", math.gcd(48, 18))
```

Output:

```
Factorial of 5: 120
GCD of 48 and 18: 6
```

2.3 Exercise 2.1 Compound interest rate

Evaluate

$$A = P(1 + \frac{r}{100})^n$$

where $P = 100, r = 5.0, n = 7$.

Print your answer to two decimal places.

Solution

```
P = 100
r = 5.0
n = 7
A = P * (1 + r / 100) ** n
print(f"Amount is {A:.2f}")
```

Output:

```
Amount is 140.71
```

2.4 Exercise 2.2 Simple Harmonic Motion

A particle under SHM has position

$$x(t) = A\sin(\omega t + \phi)$$

with $A = 1, \omega = 2$ and $\phi = \pi/6$. Evaluate x at $t = 0.2$.

Print your answer to four decimal places.

Solution

```
import math
A = 1
omega = 2
phi = math.pi/6
t = 0.2
x = A * math.sin(omega*t + phi)
print(f"x = {x:.4f}")
```

Output:

```
x = 0.7978
```

2.5 Exercise 2.3 Projectile Motion

Evaluate the horizontal (x) and vertical (y) positions of a ball at different time intervals using the following equations for projectile motion:

$$x(t) = v_0\cos(\theta)t$$

$$y(t) = v_0\sin(\theta)t - \frac{1}{2}gt^2$$

where

- $v_0 = 20$ m/s (initial velocity)
- $\theta = 45^\circ$ (angle of projection)
- $g = 9.81$ m/s^2 (acceleration due to gravity)
- t is time in seconds

Calculate the positions at $t = 5$ seconds.

Print your answers to two decimal places.

Solution

```
import math

# Parameters
v0 = 20  # initial velocity in m/s
theta = 45  # angle of projection in degrees
g = 9.81  # acceleration due to gravity in m/s^2

# Convert angle to radians
theta_rad = math.radians(theta)

# Time
t = 5

# Calculate and print x and y positions
x = v0 * math.cos(theta_rad) * t
y = v0 * math.sin(theta_rad) * t - 0.5 * g * t**2
print(f"At t = {t} s: x = {x:.2f} m, y = {y:.2f} m")
```

Output:

```
At t = 5 s: x = 70.71 m, y = -51.91 m
```

2.6 Exercise 2.4 The bell curve

Evaluate the Gaussian function (bell curve):

$$f(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

where $\mu = 0, \sigma = 1$, and $x = 1$.

Print your answer to four decimal places.

Solution

```
import math

# Parameters
mu = 0
sigma = 1
x = 1

# Evaluates the Gaussian function
f =  (1 / (math.sqrt(2 * math.pi * sigma**2))) \
        * math.exp(-((x - mu)**2) / (2 * sigma**2))

print(f"f({x}) = {f:.4f}")
```

Output:

```
f(1) = 0.2420
```

Chapter 3: Data structures

3.1 Lists

3.1.1 Creating and accessing lists

A list is an ordered collection of items. A list is

- Mutable: Can be changed after creation.
- Heterogeneous: Can contain different types.
- Indexed: Elements have specific positions starting from 0.

```
# List of integers
integers = [1, 2, 3, 4, 5]

# List of numbers
floats = [1.9, 2.2, -3.5, 1.0, -0.4]

# List of strings
fruits = ["apple", "banana", "cherry"]

# List of booleans
states = [False, True, True]

# Mixed list
mixed = [1, "hello", 3.14, True]

# Length of list
length_mixed = len(mixed)
print(length_mixed)

# Repeated list
seven_elevens = 7*[11]
print(seven_elevens)

# Create empty list
empty_list = []
print(empty_list)
```

Output:

```
4
[11, 11, 11, 11, 11, 11, 11]
[]
```

Accesing and modifying elements

```
# List of strings
fruits = ["apple", "banana", "cherry", "mango", "blueberry"]

# Accessing elements
first_element = fruits[0]
third_element = fruits[2]
last_element = fruits[-1]
second_to_last = fruits[-2]

print(first_element, third_element, last_element, second_to_last)
```

Output:

```
apple cherry blueberry mango
```

3.1.2 List methods

```
mylist = [1, 20, 3, 9, 17]

# Append an element to the end of list
mylist.append(6)
print(mylist)

# Extend with multiple elements
mylist.extend([10, 13, 9])
print(mylist)

# Insert at a specific position
mylist.insert(1, 15)
print(mylist)

mylist.insert(-3, 15)
print(mylist)

# Remove the first occurrence of an element
mylist.remove(20)
print(mylist)

# Pop last element
mylist.pop()
print(mylist)

# Pop element at a specific position
mylist.pop(2)
print(mylist)
```

Output:

```
[1, 20, 3, 9, 17, 6]
[1, 20, 3, 9, 17, 6, 10, 13, 9]
[1, 15, 20, 3, 9, 17, 6, 10, 13, 9]
[1, 15, 20, 3, 9, 17, 6, 15, 10, 13, 9]
[1, 15, 3, 9, 17, 6, 15, 10, 13, 9]
[1, 15, 3, 9, 17, 6, 15, 10, 13]
[1, 15, 9, 17, 6, 15, 10, 13]
```

```
mylist = [1, 20, 3, 9, 17, 20, 3, 10, 3, 4, 6, 11, 18, 15, 9]

# Find the index of first occurrance of an element
index = mylist.index(20)
print(index)

# Count occurrences of an element
counts = mylist.count(3)
print(counts)

# Sort the list
mylist.sort()
print(mylist)

# Reverse the list
mylist.reverse()
print(mylist)
```

```
# Copy the list
copied_list = mylist.copy()
print(copied_list)

# Clear the list
mylist.clear()
print(list)
```

Output:

```
1
3
[1, 3, 3, 3, 4, 6, 9, 9, 10, 11, 15, 17, 18, 20, 20]
[20, 20, 18, 17, 15, 11, 10, 9, 9, 6, 4, 3, 3, 3, 1]
[20, 20, 18, 17, 15, 11, 10, 9, 9, 6, 4, 3, 3, 3, 1]
<class 'list'>
```

3.1.3 List slicing

```
numbers = [1, 3, 7, 5, 12, 14]
# Slicing elements
first3 = numbers[:3]
last3 = numbers[-3:]
subset = numbers[2:4]
print(first3, last3, subset)
```

Output:

```
[1, 3, 7] [5, 12, 14] [7, 5]
```

3.1.4 Concatenating Lists

```
list1 = [1, 2, 3]
list2 = [4, 5, 6]

# Repeating lists
periodic_list = 5*list1
print(periodic_list)

# Concatenating lists
combined_list = list1 + list2
print(combined_list)
```

Output:

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

3.1.5 Summing numerical lists with `sum()`

```
numbers = [1, 2, 3, 4, 5]
total = sum(numbers)
print(total)
```

Output:

```
15
```

3.2 Tuples

A tuple is a collection of ordered, immutable (unchangeable) elements.

```python
# Creating a tuple of integers
integers_tuple = (4, 3, 2, 1)

# Tuples can contain different data types
mixed_tuple = (1, "hello", 3.14)

# Creating a tuple without parentheses (tuple packing)
another_tuple = 5, 6, 7, 8

# Creating a tuple with one item (note the comma)
single_item_tuple = (10,)

# Creating an empty tuple
empty_tuple = ()
```

3.2.1 Accessing elements

```python
primes = (2, 3, 5, 7, 11)

# Number of elements
print("Length:", len(primes))

# Accessing the first element
print("First element:", primes[0])

# Accessing the last element
print("Last element:", primes[-1])

# Slicing tupled
print("Sliced tuple:", primes[1:3])
```

Output:

```
Length: 5
First element: 2
Last element: 11
Sliced tuple: (3, 5)
```

3.2.2 Unpacking tuples

```python
potassium = ("K", 19, 39.0983)

# Unpacking tuple elements into variables
symbol, number, weight = potassium

print(symbol)
print(number)
print(weight)
```

Output:

```
K
19
39.0983
```

3.2.3 Tuple operations

```python
tuple1 = (1, 2, 3)
tuple2 = (4, 5, 6)

# Concatenation
```

```
concatenated_tuple = tuple1 + tuple2
print(concatenated_tuple)

# Repetition
repeated_tuple = tuple1 * 3
print(repeated_tuple)

# Membership
check = 3 in tuple1
print(check)
```

Output:

```
(1, 2, 3, 4, 5, 6)
(1, 2, 3, 1, 2, 3, 1, 2, 3)
True
```

3.3 Dictionaries

A dictionary is a collection of key-value pairs. Each key is unique and maps to a corresponding value. Dictionaries are mutable.

3.3.1 Creating dictionaries

```
# Dictionary representing Carbon in the periodic table
carbon = {
    'symbol': 'C',
    'name': 'Carbon',
    'atomic_number': 6,
    'atomic_mass': 12.0,
    'electron_config': ['1s2', '2s2', '2p2']
}

isotope= dict(symbol='C-12', mass_number=12, abundance=98.93)
```

3.3.2 Adding and modifying entries

```
# Dictionary representing Carbon in the periodic table
carbon = {
    'name': 'Carbon',
    'atomic_mass': 12.0
}

# Adding a new key-value pair
carbon['group'] = 14

# Modifying an existing value
carbon['atomic_mass'] = 12.011

# Adding and modifying using the update() method
carbon.update({'name': 'CARBON', 'period': 2})

print(carbon)
```

Output:

```
{'name': 'CARBON', 'atomic_mass': 12.011, 'group': 14, 'period': 2}
```

3.3.3 Deleting entries

```
carbon = {
    'symbol': 'C',
    'name': 'Carbon',
    'atomic_number': 6,
    'atomic_mass': 12.0
}

# Using del statement
del carbon['name']

# Using the pop() method
atomic_mass = carbon.pop('atomic_mass')
print(atomic_mass)

print("Carbon dict:", carbon)
```

Output:

```
12.0
Carbon dict: {'symbol': 'C', 'atomic_number': 6}
```

3.3.4 Dictionary methods

```
carbon = {
    'symbol': 'C',
    'atomic_mass': 12.0,
    'atomic_number': 6
}

# Getting all keys
keys = carbon.keys()
print("Keys:", keys)
print("Keys as a list:", list(keys))

# Getting all values
values = carbon.values()
print("Values:", values)
print("Values as a list:", list(values))


# Getting all key-value pairs
items = carbon.items()
print("Items:", items)
print("Items as a list:", list(items))

# Using get() method
config = carbon.get('electron_config', 'No config. found')
print("Electron config:", config)
```

Output:

```
Keys: dict_keys(['symbol', 'atomic_mass', 'atomic_number'])
Keys as a list: ['symbol', 'atomic_mass', 'atomic_number']
Values: dict_values(['C', 12.0, 6])
Values as a list: ['C', 12.0, 6]
Items: dict_items([('symbol', 'C'), ('atomic_mass', 12.0), ('atomic_number', 6)])
Items as a list: [('symbol', 'C'), ('atomic_mass', 12.0), ('atomic_number', 6)]
Electron config: No config. found
```

3.3.5 Nested dictionaries

```
C12 = {'mass_number': 12, 'abundance': 98.93}
C14 = {'mass_number': 13, 'abundance': 1.07}

carbon = {
    'symbol': 'C',
    'atomic_number': 6,
    'atomic_mass': 12.011,
    'isotopes': {'C-12': C12, 'C-13': C14}
}

print("C-12 isotope:", carbon["isotopes"]["C-12"])

print("C-12 abundance:", carbon["isotopes"]["C-12"]["abundance"])
```

Output:

```
C-12 isotope: {'mass_number': 12, 'abundance': 98.93}
C-12 abundance: 98.93
```

3.4 Exercise 1: Grade statistics

The numerical grades for a cohort of girls and boys at a school are given below:

Girls: 68, 92, 78, 80, 65, 51, 44, 65

Boys: 76, 85, 89, 65, 84, 77, 99, 67, 81

Calculate the median grade and the range of the grades for the **whole cohort**. It is known that the highest and lowest grades are outliers and should be removed.

Print a filtered list of grades, the median grade, and the range of grades.

Hint: The median is the value at the middle of the list. For a list of odd length `n`, the median is the value of the element with index `n//2`. For a list of even length `n`, the median is the average of the values with indices `n//2` and `n//2-1`.

Solution

```
# Lists of numerical grades for girls and boys (unsorted)
girls_grades = [68, 92, 78, 80, 65, 51, 44, 65]
boys_grades = [76, 85, 89, 65, 84, 77, 99, 67, 81]

# Combine the two lists
grades = girls_grades + boys_grades
grades.sort()

# Remove first and last element
grades.pop(0)
grades.pop()

n = len(grades)
median = grades[n // 2]

# Calculate the range
grade_range = max(grades) - min(grades)

# Output the results
print(f"Filtered Grades: {grades}")
print(f"Median Grade: {median}")
print(f"Range of Grades: {grade_range}")
```

Output:

```
Filtered Grades: [51, 65, 65, 65, 67, 68, 76, 77, 78, 80, 81, 84, 85, 89, 92]
Median Grade: 77
Range of Grades: 41
```

3.5 Exercise 3.2 Luminance of RGB colors

The luminance (grayscale brightness) of an RGB color can be calculated using a weighted sum of its red (R), green (G), and blue (B) components. The RGB components each have a value from 0 (min) to 255 (max). The luminance Y is given by

$$Y = 0.2126R + 0.7152G + 0.0722B$$

where

- $0 \leq R \leq 255$ is the Red component of the color
- $0 \leq G \leq 255$ is the Green component of the color
- $0 \leq B \leq 255$ is the Blue component of the color

These coefficients (0.2126, 0.7152, 0.0722) reflect the relative sensitivity of the human eye to red, green, and blue light, respectively. The green component has the highest coefficient because the human eye is most sensitive to green light.

By using the luminance formula, a color image can be converted to a grayscale image that maintains the perceived brightness of the original colors.

For example, "slate blue" has RGB values of (106, 90, 205). The resulting luminance is $Y = 101.7$, so the grayscale value is 102 (after rounding). The RGB values of the grayscale color are thus (102, 102, 102).

(i) Store the following colors in a dictionary where the keys are the color names and the values are their corresponding RGB tuples:

- Crimson (220, 20, 60)
- DodgerBlue (30, 144, 255)
- ForestGreen (34, 139, 34)
- Gold (255, 215, 0)
- HotPink (255, 105, 180)
- SlateBlue (106, 90, 205)

Extract the RGB tuple values of any given color from the dictionary and store it as tuple containing (R, G, and B) named `rgb`.

(ii) Compute the grayscale tuple `gray` for the equivalent grayscale color. Each component (R, G, and B) of `gray` should be equal to the calculated luminance (rounded).

(iii) Copy and paste the code below to plot a patch of `rgb` and `gray` side by side.

```
# Plots rgb and gray side by side
import matplotlib.pyplot as plt
fig, ax = plt.subplots(figsize=(3, 2))
# Plots the original color rectangle
ax.add_patch(plt.Rectangle((0, 0), 0.5, 1, color=[c/255 for c in rgb]))
# Plots the grayscale rectangle
ax.add_patch(plt.Rectangle((0.5, 0), 0.5, 1, color=[c/255 for c in gray]))
# Styles plot
ax.set_xlim(0, 1)
ax.set_ylim(0, 1)
ax.axis('off')
plt.show()
```

Solution

```
# Define a dictionary of RGB colors
rgb_colors = {
    'Crimson': (220, 20, 60),
    'DodgerBlue': (30, 144, 255),
    'ForestGreen': (34, 139, 34),
    'Gold': (255, 215, 0),
    'HotPink': (255, 105, 180),
    'SlateBlue': (106, 90, 205)
}

# Extract a single color from the dictionary
color_name = 'HotPink'
rgb = rgb_colors[color_name]

# Calculate luminance
r, g, b = rgb
Y = round(0.2126 * r + 0.7152 * g + 0.0722 * b)
gray = (Y , Y , Y )

# Plots rgb and gray side by side
import matplotlib.pyplot as plt
fig, ax = plt.subplots(figsize=(3, 2))
# Plots the original color rectangle
ax.add_patch(plt.Rectangle((0, 0), 0.5, 1, color=[c/255 for c in rgb]))
# Plots the grayscale rectangle
ax.add_patch(plt.Rectangle((0.5, 0), 0.5, 1, color=[c/255 for c in gray]))
# Styles plot
ax.set_xlim(0, 1)
ax.set_ylim(0, 1)
ax.axis('off')
plt.show()
```

```
<Figure size 300x200 with 1 Axes>
```

Chapter 4: Loops

4.1 `for` Loops

A `for` loop in Python is used to iterate over a sequence (such as a list, tuple, string, or range) and execute a block of code multiple times. It is commonly used for iterating over elements in a list or any other iterable.

```
for element in sequence:
    # Code to execute
```

```
# Iterating Over a List
fruits = ["apple", "banana", "cherry"]
for fruit in fruits:
    print(fruit)
```

Output:

```
apple
banana
cherry
```

```
# Iterating Over a Tuple
fruits = "apple", "banana", "cherry"
for fruit in fruits:
    print(fruit)
```

Output:

```
apple
banana
cherry
```

```
# Iterating Over a String
fruit = "apple"
for letter in fruit:
    print(letter)
```

Output:

```
a
p
p
l
e
```

4.1.1 `for` Loops with `range()`

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

Output:

```
0
1
2
3
4
```

```
for j in range(2,6):
    print(j)
```

Output:

```
2
3
4
5
```

```
for k in range(15, 3, -3):
    print(k)
```

Output:

```
15
12
9
6
```

4.1.2 `for` loops with `enumerate()`

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

Output:

```
0 apple
1 banana
2 cherry
```

```
fruits = ["apple", "banana", "cherry"]
for index, fruit in enumerate(fruits, start=4):
    print(index, fruit)
```

Output:

```
4 apple
5 banana
6 cherry
```

4.1.3 Iterating over dictionaries

```
carbon = {
    'symbol': 'C',
    'atomic_mass': 12.0,
    'atomic_number': 6
}
# Iterating over keys
for key in carbon:
    print(key, ":", carbon[key])

# Iterating over key-value pairs
for key, value in carbon.items():
    print(key, ":", value)
```

Output:

```
symbol : C
atomic_mass : 12.0
atomic_number : 6
symbol : C
atomic_mass : 12.0
atomic_number : 6
```

4.1.4 Nested `for` loops

```
for i in range(3):
  print("---")
  for j in range(4):
    print("i =", i, " j =", j)
```

Output:

```
---
i = 0  j = 0
i = 0  j = 1
i = 0  j = 2
i = 0  j = 3
---
i = 1  j = 0
i = 1  j = 1
i = 1  j = 2
i = 1  j = 3
---
i = 2  j = 0
i = 2  j = 1
i = 2  j = 2
i = 2  j = 3
```

4.2 List comprehension

List comprehensions provide a concise way to create lists.

```
new_list = [expression for item in iterable]
```

```
# List of Squares
squares = [x**2 for x in range(10)]
print(squares)
```

Output:

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

```
# Using filtering: List of even numbers
evens = [x for x in range(10) if x % 2 == 0]
print(evens)
```

Output:

```
[0, 2, 4, 6, 8]
```

```
# Using enumerate: List of indexed fruits (List of tupples)
fruits = ["apple", "banana", "cherry"]
indexed_fruits = [(index, fruit) for index, fruit in enumerate(fruits)]
print(indexed_fruits)
```

Output:

```
[(0, 'apple'), (1, 'banana'), (2, 'cherry')]
```

4.3 `while` loops

A `while` loop in Python repeatedly executes a block of code as long as a given condition is true. It is commonly used when the number of iterations is not known beforehand and depends on a condition.

```
while condition:
    # Code to execute
```

```
count = 0
while count < 5:
```

```
    print(count)
    count += 1
```

Output:

```
0
1
2
3
4
```

```
user_input = ''
while user_input.lower() != 'exit':
    user_input = input("Type 'exit' to stop: ")
    print(f"You typed: {user_input}")
```

Output:

```
You typed: hello
You typed:
You typed: Exit
```

4.3.1 `break` and `continue` statements

- `break`: Exits the loop immediately.
- `continue`: Skips the rest of the code inside the loop for the current iteration and moves to the next iteration.

```
count = 0
while True:
    print(count)
    count += 1
    if count == 3:
        break
```

Output:

```
0
1
2
```

```
count = 0
while count < 5:
    count += 1
    if count == 3:
        continue
    print(count)
```

Output:

```
1
2
4
5
```

4.4 Exercise 4.1 Compound interest revisited

The formula for compound interest is given by:

$$A = P\left(1 + \frac{r}{100}\right)^n$$

where:

- A is the amount of money accumulated after n years, including interest.

- P is the principal amount (the initial amount of money).
- r is the annual interest rate (in percentage).
- n is the number of years the money is invested or borrowed for.

Given $P = 100, r = 5.0$, calculate the value of A for each year from $n = 0$ to $n = 10$. Store all values in a list.

Solution

```
# Parameters
P = 100
r = 5.0

# Calculate A for each year from n = 0 to n = 10 and store in a list
A_values = []
for n in range(11):
    A = P * (1 + r / 100) ** n
    A_values.append(round(A, 2))

# Print the list of values
for i, val in enumerate(A_values):
  print(i, val)
```

Output:

```
0 100.0
1 105.0
2 110.25
3 115.76
4 121.55
5 127.63
6 134.01
7 140.71
8 147.75
9 155.13
10 162.89
```

4.5 Exercise 4.2 Sum of the cubes

The sum of the cubes of the first n natural numbers $ \sum_{r=1}^{n} r^3 $ can be obtained by the standard formula:

$$\sum_{r=1}^{n} r^3 = \frac{1}{4}n^2(n+1)^2$$

i. Calculate the sum of the cubes of the first $n = 20$ natural numbers using a for loop and then the standard formula.

ii. Find the sum: $13^3 + 14^3 + ... + 29^3$.

iii. Find the sum of the cubes of the first n natural numbers for $n = 1, 2, ...10$. Store the results in a list.

Solution

```
n = 20

# Part i: Calculate the sum of the cubes of the first n natural numbers
# Using a for loop
sum_cubes_loop = sum(r**3 for r in range(1, n+1))
print(f"Sum of cubes using loop: {sum_cubes_loop}")

# Using the standard formula
sum_cubes_formula = (n * (n + 1) // 2) ** 2
```

```
print(f"Sum of cubes using formula: {sum_cubes_formula}")

# Part ii: Find the sum 13^3 + 14^3 + ... + 29^3
sum_cubes_range = sum(r**3 for r in range(13, 30))
print(f"Sum of cubes from 13 to 30: {sum_cubes_range}")

# Part iii: Find the sum of the cubes of the first n natural numbers for n=1,2,...10
cubes_list = [(i, sum(r**3 for r in range(1, i + 1))) for i in range(1, 11)]
print("Sum of cubes for n=1 to 10:")
for i, s in cubes_list:
    print(f"n={i}: {s}")
```

Output:

```
Sum of cubes using loop: 44100
Sum of cubes using formula: 44100
Sum of cubes from 13 to 30: 183141
Sum of cubes for n=1 to 10:
n=1: 1
n=2: 9
n=3: 36
n=4: 100
n=5: 225
n=6: 441
n=7: 784
n=8: 1296
n=9: 2025
n=10: 3025
```

4.6 Exercise 4.3 Logistic growth

The logistic growth model is given by the equation:

$$P_{t+1} = P_t + rP_t\left(1 - \frac{P_t}{K}\right)$$

where:

- P_t is the population size t years after the start of growth.
- $r = 0.3$ is the intrinsic growth rate.
- $K = 100$ is the carrying capacity.
- P_{t+1} is the population size at time $t + 1$.

Given the initial population size $P_0 = 10$, calculate the population size for each year from $t = 0$ to $t = 40$ and store all values in a list called `population_sizes`. Print the population size after every 10 years.

Then, visualize the growth dynamics by copying the following code:

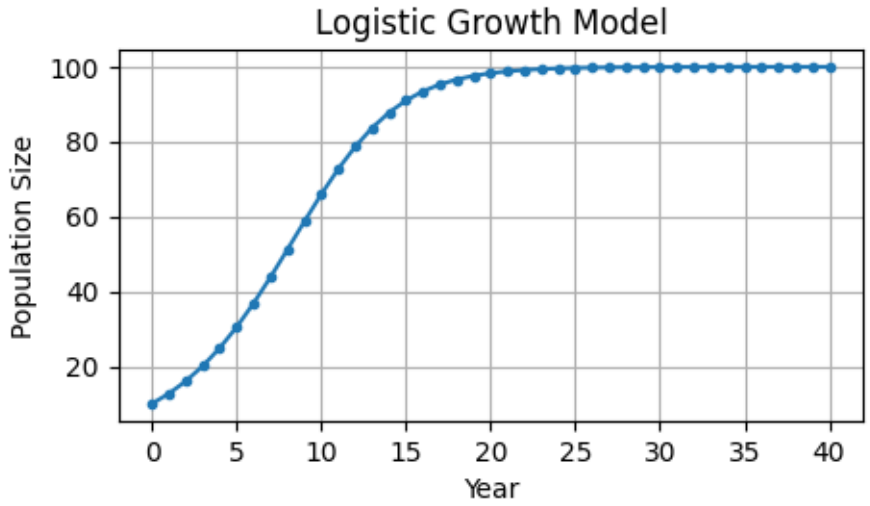
```
import matplotlib.pyplot as plt

# Plot the results
plt.plot(population_sizes, marker='o')
plt.xlabel('Year')
plt.ylabel('Population Size')
plt.title('Logistic Growth Model')
plt.grid(True)
plt.show()
```

Solution

```python
import matplotlib.pyplot as plt

# Parameters
P0 = 10  # Initial population size
r = 0.3  # Intrinsic growth rate
K = 100  # Carrying capacity
n = 40  # Number of time steps

# Initialize the list to store population sizes
population_sizes = [P0]

# Calculate population sizes for each time step
for t in range(1, n + 1):
  Pt = population_sizes[-1]
  Pt_next = Pt + r * Pt * (1 - Pt / K)
  population_sizes.append(Pt_next)

# Print the results
for t, P in enumerate(population_sizes):
  if t%10==0:
    print(f"Year {t}: Population size = {P:.2f}")

# Plot the results
plt.figure(figsize=(5,2.5))
plt.plot(population_sizes, marker='.')
plt.xlabel('Year')
plt.ylabel('Population Size')
plt.title('Logistic Growth Model')
plt.grid(True)
plt.show()
```

Output:

```
Year 0: Population size = 10.00
Year 10: Population size = 65.97
Year 20: Population size = 98.31
Year 30: Population size = 99.95
Year 40: Population size = 100.00
```




```
<Figure size 500x250 with 1 Axes>
```

4.7 Exercise 4.4 Estimating π Using the Leibniz Formula

The Leibniz formula for π is an infinite series that converges to $\pi/4$:

$$\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \frac{1}{11} + \cdots$$

To estimate π, we can use the formula:

$$\pi \approx 4\left(1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \cdots + \frac{(-1)^n}{2n+1}\right)$$

(i) Estimate π using the first 150 terms of the series.

(ii) Determine how many terms are required to estimate π accurately to 3 decimal places. This can be done by iterating in a while loop until the difference between the estimate an the "true" value of π not greater than 4.99×10^{-4}.

Solution

```python
import math

# Estimate pi using the first 150 terms
n_terms = 150
pi_over_4 = 0
for n in range(n_terms):
    pi_over_4 += ((-1) ** n) / (2 * n + 1)
pi_est_20_terms = 4 * pi_over_4
print(f"Estimate of pi using {n_terms} terms: {pi_est_20_terms:.6f}")

# Determine the number of terms required for 3 decimal place accuracy
difference = 4.99e-4
n_terms = 0
pi_estimate = 0
pi_previous = -1
while abs(pi_estimate - math.pi) > difference:
    pi_previous = pi_estimate
    n_terms += 1
    pi_over_4 = 0
    for n in range(n_terms):
        pi_over_4 += ((-1) ** n) / (2 * n + 1)
    pi_estimate = 4 * pi_over_4

print(f"Number of terms required for 3 decimal place accuracy: {n_terms}")
print(f"Estimate of pi using {n_terms} terms: {pi_estimate:.3f}")
```

Output:

```
Estimate of pi using 150 terms: 3.134926
Number of terms required for 3 decimal place accuracy: 2005
Estimate of pi using 2005 terms: 3.142
```

Chapter 5: Functions

5.1 Function creation with `def`

Defining a Function You define a function using the `def` keyword, followed by the function name, parentheses `()`, and a colon `:`. The code block within the function is indented.

```
def function_name(parameters):
    """
    Docstring (optional): Description of the function.
    """
    # Function body
    # Perform operations
    return result
```

Example: area of a trapezium

```
def trapezium(a: float, b: float, h: float) -> float:
    """
    Calculates the area of a trapezium.

    Parameters:
    a (float): Length of the first parallel side
    b (float): Length of the second parallel side
    h (float): Height (perpendicular distance between the parallel sides)

    Returns:
    float: Area of the trapezium
    """
    area = 0.5 * (a + b) * h
    return area

a = 5.0  # Length of the first parallel side
b = 7.0  # Length of the second parallel side
h = 4.0  # Height

print(trapezium.__doc__) # Print the function documentation
area = trapezium(a, b, h) # Call the function
print("Area of the trapezium:", area)
```

Output:

```
Calculates the area of a trapezium.

    Parameters:
    a (float): Length of the first parallel side
    b (float): Length of the second parallel side
    h (float): Height (perpendicular distance between the parallel sides)

    Returns:
    float: Area of the trapezium

Area of the trapezium: 24.0
```

Example: Lift and Drag forces on an airfoil

```
def lift_drag(V, A, rho=1.225, Cl=1.2, Cd=0.05):
    """
```

```
    Calculates the lift and drag forces on an airfoil.

    Parameters:
    V (float): Airspeed (m/s).
    A (float): Wing area (m^2).
    rho (float): Air density (kg/m^3). Default: 1.225 kg/m^3 (sea-level standard)
    Cl (float): Lift coefficient (dimensionless). Default:  1.2
    Cd (float): Drag coefficient (dimensionless). Default:  0.05

    Returns:
    tuple: (L, D)
        L (float): Lift force (Newtons).
        D (float): Drag force (Newtons).
    """
    # Calculate lift force
    L = 0.5 * rho * V**2 * A * Cl
    # Calculate drag force
    D = 0.5 * rho * V**2 * A * Cd

    return L, D

V = 50.0  # m/s
A = 20.0  # m^2

# Overwrite default values
rho = 1.12 # kg/m^3
Cl = 1.3
Cd = 0.07
L, D = lift_drag(V, A, rho, Cl, Cd)
print(f"Lift Force: {L:.2f} N")
print(f"Drag Force: {D:.2f} N")

# Use default values
L, D = lift_drag(V, A)
print(f"Default Lift Force: {L:.2f} N")
print(f"Default Drag Force: {D:.2f} N")
```

Output:

```
Lift Force: 36400.00 N
Drag Force: 1960.00 N
Default Lift Force: 36750.00 N
Default Drag Force: 1531.25 N
```

Using type hints:

```
def lift_drag(V: float, A: float, rho: float = 1.225,
              Cl: float = 1.2, Cd: float = 0.05) -> tuple[float, float]:
    """
    Returns Lift, Drag forces
    """
    L = 0.5 * rho * V**2 * A * Cl
    D = 0.5 * rho * V**2 * A * Cd
    return L, D

L, D = lift_drag(50.0, 20.0)
print(f"Lift Force: {L:.2f} N")
print(f"Drag Force: {D:.2f} N")
```

Output:

```
Lift Force: 36750.00 N
Drag Force: 1531.25 N
```

5.2 Exercise 5.1 Coulomb's law

Coulomb's law gives the electric force between two point charges:

$$F = k_e \frac{q_1 q_2}{r^2}$$

where:

- F is the force between the charges.
- $k_e = 8.99 \times 10^9$ Nm² /C² is Coulomb's constant.
- q_1 and q_2 are the magnitudes of the charges.
- r is the distance between the charges.

Write a function that computes the electric force given q_1, q_2, r.

Test it for $q_1 = 1$ microcoulombs, $q_2 = -2$ microcoulombs and $r = 5$ cm.

Solution

```python
def electric_force(q1: float, q2: float, r: float) -> float:
    """
    Calculates the electric force between two point charges.

    Parameters:
    q1 (float): Charge of the first particle (Coulombs)
    q2 (float): Charge of the second particle (Coulombs)
    r (float): Distance between the charges (meters)

    Returns:
    float: Electric force (Newtons)
    """
    k_e = 8.99e9  # Coulomb's constant in Nm^2/C^2
    force = k_e * q1 * q2 / r**2
    return force

# Charges in Coulombs
q1 = 1e-6  # 1 microCoulomb
q2 = -2e-6  # -2 microCoulombs
r = 0.05  # 5 centimeters converted to meters

# Calculate electric force
force = electric_force(q1, q2, r)
print(f"Electric Force: {force:.2f} N")
```

Output:

```
Electric Force: -7.19 N
```

5.3 Exercise 5.2 Turbulent pipe flow

The Reynolds number (Re) is given by:

$$Re = \frac{\rho V D}{\mu}$$

where ρ is the fluid density, V is the speed, D is a characteristic length and μ the dynamic viscosity of the fluid.

It represents as the ratio of inertial forces to viscous forces within a fluid that is moving.

Flow in a pipe of diameter D is considered turbulent if $Re > 2900$.

Write a function with inputs ρ, V, D, μ that returns `True` if the flow in a pipe of diameter D is turbulent and `False` otherwise.

Test it for - $\rho = 1000$ kg/m³ - $V = 2.0$ m/s - $D = 0.2$ m - $\mu = 0.001$ Pa·s

Solution

```python
def is_turbulent(rho: float, V: float, D: float, mu: float) -> bool:
    """
    Checks if the flow is turbulet based on the Reynolds number.

    Parameters:
    rho (float): Density of the fluid (kg/m³)
    V (float): Velocity of the fluid (m/s)
    D (float): Characteristic length (m)
    mu (float): Dynamic viscosity of the fluid (Pa·s)

    Returns:
    bool: True if the flow is turbulent, False if laminar
    """
    Re = rho*v*D/mu
    return Re > 2900

# Example usage
rho = 1000.0  # Density of water in kg/m³
V = 0.01       # Velocity of body in m/s
D = 0.2      # Characteristic Length of body in m
mu = 0.001    # Dynamic viscosity of water in Pa·s

if is_turbulent(rho, V, D, mu):
    print("The flow is turbulent.")
else:
    print("The flow is laminar.")
```

5.4 Exercise 5.3 Quadratic solver

Write a function that solves the quadratic equation

$$ax^2 + bx + c = 0$$

The function takes a, b and c, and returns a tuple containing the two real solutions. The function returns a tuple `(None, None)` if there are no real solutions.

Use it to solve:

(i) $x^2 + -3x + 2 = 0$.

(ii) $3x^2 + x + 4 = 0$.

Solution

```python
import math

def quadratic_solver(a: float, b: float, c: float) -> tuple[float, float]:
    """
    Solves the quadratic equation ax^2 + bx + c = 0.
    Returns a tuple containing the two real solutions.
    The tuple is (None, None) if there are no real solutions.
    """
```

```
    discriminant= b**2 - 4*a*c
    if discriminant < 0:
      return None, None
    x1 = (-b + math.sqrt(discriminant)) / (2*a)
    x2 = (-b - math.sqrt(discriminant)) / (2*a)
    return x1, x2


print(quadratic_solver.__doc__)
x1, x2 = quadratic_solver(3, 1, 4)
print("Solutions:", x1, x2)

a, b, c = 1, -3, 2
x1, x2 = quadratic_solver(a, b, c)
print("Solutions:", x1, x2)
```

Output:

```
Solves the quadratic equation ax^2 + bx + c = 0.
    Returns a tuple containing the two real solutions.
    The tuple is (None, None) if there are no real solutions.

Solutions: None None
Solutions: 2.0 1.0
```

5.5 Exercise 5.4 Prime checker

Write a function that checks if a number is prime. The function should return `True` if the number is prime and `False` otherwise. To check if integer n is prime, you need to check all divisors from 2 up to $\sqrt{n}$.

(i) Test the function for the numbers 179 (prime) and 777 (not prime).

(ii) Print all prime numbers from 20,000 to 20,100.

Solution

```
def is_prime(n: int) -> bool:
    """
    Checks if a number is prime.

    Parameters:
    n (int): The number to check

    Returns:
    bool: True if the number is prime, False otherwise
    """
    if n <= 1:
        return False
    for i in range(2, int(n ** 0.5) + 1):
        if n % i == 0:
            return False
    return True
# Test the function
n1 = 179
print(f"Is {n1} a prime number? {is_prime(n1)}")
n2 = 777
print(f"Is {n2} a prime number? {is_prime(n2)}")

# Print all prime numbers from 10000 to 10100:
for n in range(20000, 20100):
```

```
  if is_prime(n):
    print(n, end=" ")
```

Output:

```
Is 179 a prime number? True
Is 777 a prime number? False
20011 20021 20023 20029 20047 20051 20063 20071 20089
```

5.6 Exercise 5.5 The International Standard Atmosphere

The International Standard Atmosphere (ISA) is a model used to represent the standard variation of atmospheric properties with altitude. It is widely used in the fields of aviation, aerospace, and meteorology. The ISA defines values for atmospheric pressure, temperature and density over a range of altitudes for standard conditions.

1. Troposphere (0 to 11 km)

In the troposphere, the temperature T decreases linearly with altitude h:

$$T(h) = T_0 + Lh$$

where

- $T_0 = 288.15$ K is the sea level standard temperature
- $L = -0.0065$ K/m is the standard temperature lapse rate

The pressure p at altitude h is given by

$$p(h) = p_0 \left(\frac{T(h)}{T_0} \right)^{\frac{-g}{LR}}$$

where

- $p_0 = 101325$ Pa is the sea level standard pressure
- $g = 9.81$ m/s² is the acceleration due to gravity (9.80665)
- $R = 287.05$ J/(kg·K) is the specific gas constant for dry air

2. Tropopause (11 to 20 km)

In the tropopause, the temperature remains constant,

$$T(h) = T_{11}$$

where T_{11} is the temperature at 11 km, calculated from the troposphere equations.

The pressure can be found using:

$$p(h) = p_{11} \cdot \exp\left(\frac{-g(h - 11000)}{R \cdot T_{11}} \right)$$

where p_{11} is the pressure at 11 km, calculated from the troposphere equations.

The air density in both the Tropopause and Troposphere can be calculated using the ideal gas law:

$$\rho(h) = \frac{p(h)}{R \cdot T(h)}$$

Task 1

Create a Python function that takes height h (in meters) as input and returns the atmospheric pressure, temperature, density, and the layer name ("Tropopause" or "Troposphere") as a 4-element tuple:

```
p, T, rho, layer = ISA_model(h)
```

Test it for the following locations and corresponding altitudes:

1. Burj Khalifa: 828 m

2. Mount Everest: 8,848 m
3. Cruise altitude of commercial airliners: 12 km

Task 2

Create a function that formats the location, altitude and atmospheric conditions into a single, neatly structured string using f-string. The formatted output should include units and display values to two decimal places where applicable.

For example,

```
def get_conditions(location: str, h: float)->str:
    p, T, rho, layer = ISA_model(h)
    formatted_conditions =  f"..."
    return formatted_conditions

print(get_conditions("Mount Everest", 8848))
```

Solution

```
import math

def ISA_model(h):
    # Constants
    T0 = 288.15  # Sea level standard temperature in K
    p0 = 101325  # Sea level standard pressure in Pa
    L = -0.0065  # Standard temperature lapse rate in K/m
    g = 9.81  # Acceleration due to gravity in m/s²
    R = 287.05  # Specific gas constant for dry air in J/(kg·K)

    # Troposphere calculations
    if h <= 11000:
        T = T0 + L * h
        p = p0 * (T / T0) ** (-g / (L * R))
        layer = "Troposphere"
    # Tropopause calculations
    else:
        T11 = T0 + L * 11000
        p11 = p0 * (T11 / T0) ** (-g / (L * R))
        T = T11
        p = p11 * math.exp(-g * (h - 11000) / (R * T11))
        layer = "Tropopause"

    # Density calculation
    rho = p / (R * T)

    return (p, T, rho, layer)

def get_conditions(location: str, h: float)->str:
    p, T, rho, layer = ISA_model(h)
    formatted_conditions =  f"{location} (Height: {h} m)\n" \
                        f"Layer: {layer}\n" \
                        f"p: {p:.2f} Pa\n" \
                        f"T: {T:.2f} K\n" \
                        f"density: {rho:.2f} kg/m³\n"
    return formatted_conditions

# Test cases
print(get_conditions("Burj Khalifa", 828))
print(get_conditions("Mount Everest", 8848))
```

```
print(get_conditions("Cruise altitude", 12000))
```

Output:

```
Burj Khalifa (Height: 828 m)
Layer: Troposphere
p: 91762.33 Pa
T: 282.77 K
density: 1.13 kg/m³

Mount Everest (Height: 8848 m)
Layer: Troposphere
p: 31431.04 Pa
T: 230.64 K
density: 0.47 kg/m³

Cruise altitude (Height: 12000 m)
Layer: Tropopause
p: 19319.13 Pa
T: 216.65 K
density: 0.31 kg/m³
```

5.7 Exercise 5.6 Pascal's triangle

Pascal's Triangle is a triangular array of the binomial coefficients. Each number is the sum of the two directly above it.

```
n=0:                    1
n=1:                  1   1
n=2:                1   2   1
n=3:              1   3   3   1
n=4:            1   4   6   4   1
n=5:          1   5  10  10   5   1
n=6:        1   6  15  21  15   6   1
```

The k-th number in row n represent the coefficient $\binom{n}{k}$ of the binomial expansion:

$$(1+x)^n = \sum_{k=0}^{n} \binom{n}{k} x^k$$

Note that both k and n start from 0. For example, $\binom{4}{2} = 6$.

Write a function that takes the row number n as input and returns the numbers in that row of Pascal's Triangle as a list.

Challenge: Return a nested list `triangle = [row_1, row_2, ..., row_n]`, where each element is a list that contains a row in Pascal's Triangle. For example, `row_1 = [1]`, `row_2 = [1, 1]`, etc.

Solution

```
def pascals_triangle(n: int)->list[list]:
    """
    Generates Pascal's Triangle up to the nth row and returns it as a nested list.
    Parameters:
    n (int): The number of rows of Pascal's Triangle to generate.
    Returns:
    list: A nested list representing Pascal's Triangle.
    """
    triangle = []

    for row_num in range(n):
```

```
        row = [1] * (row_num + 1)
        for j in range(1, row_num):
            row[j] = triangle[row_num - 1][j - 1] + triangle[row_num - 1][j]
        triangle.append(row)

    return triangle

# Example usage
rows = 5
triangle = pascals_triangle(rows)
print("Pascals:", triangle)
```

Output:

```
[1]
[1, 1]
[1, 2, 1]
[1, 3, 3, 1]
[1, 4, 6, 4, 1]
Pascals: [[1], [1, 1], [1, 2, 1], [1, 3, 3, 1], [1, 4, 6, 4, 1]]
```

Chapter 6: Numerical arrays and plotting

6.1 Numerical arrays in `numpy`

NumPy is a powerful library for numerical computing in Python. It provides support for arrays, matrices, and many mathematical functions to operate on these data structures. NumPy is widely used in data science, machine learning, scientific computing, and engineering.

```
import numpy as np
```

6.1.1 One-dimensional arrays

```
# Creating a 1D array
a = np.array([1, 2, 3, 4, 5])
print(a)
print(type(a))
b = np.array([.1, -2, 3.9, 0, 0.5])
print(b)
```

Output:

```
[1 2 3 4 5]
<class 'numpy.ndarray'>
[ 0.1 -2.   3.9  0.   0.5]
```

6.1.2 Using `zeros`, `ones`, `arange`, `linspace`, and `logspace`

```
# Array of zeros: np.zeros(size)
zeros_array = np.zeros(5)
print(zeros_array)

# Array of ones: np.ones(size)
ones_array = np.ones(5)
print(ones_array)

# Array with a range of values: np.arange(start, stop, step)
range_array = np.arange(0, 10, 2)
print(range_array)

# Array with linearly spaced values:
# np.linspace(a, b, number) outputs an array of n equally-spaced values from a to b
linspace_array = np.linspace(0, 1, 5)
print(linspace_array)

# Array with logarithmically spaced values
# np.logspace(a, b, n) outputs an array n log-spaced values from 10^a to 10^b
logspace_array = np.logspace(0, 2, 5)
print(logspace_array)
```

Output:

```
[0. 0. 0. 0. 0.]
[1. 1. 1. 1. 1.]
[0 2 4 6 8]
[0.   0.25 0.5  0.75 1.  ]
[  1.           3.16227766  10.          31.6227766  100.        ]
```

6.1.3 Checking and casting variable types

```
# Creating an integer array
int_array = np.array([1, 2, 3, 4, 5])
print(f"{int_array}: {int_array.dtype}")

# Creating a float array
float_array = np.array([0.6, 2.1, 3.8, -4.0, -5.5])
print(f"{float_array}: {float_array.dtype}")

# Casting a float array into an int array
casted_array = float_array.astype(int)
print(f"{casted_array}: {casted_array.dtype}")
```

Output:

```
[1 2 3 4 5]: int64
[ 0.6  2.1  3.8 -4.  -5.5]: float64
[ 0  2  3 -4 -5]: int64
```

6.1.4 Array operations

```
a = np.array([1.0, 2.0, 3.0, 4.0])

# Scalar multiplication
print("Scalar addition:", a + 10)

# Scalar multiplication
print("Scalar multiplication:", 2*a)

b = np.array([-3.0, 1.0, 0.5, -1.5])

# Element-wise addition
print("Element-wise addition:", a + b)

# Element-wise subtraction
print("Element-wise subtraction:", a - b)

# Element-wise multiplication
print("Element-wise multiplication:", a * b)

# Element-wise division
print("Element-wise division:", a / b)

# Element-wise division
print("Element-wise integer division:", a // b)
```

Output:

```
Scalar addition: [11. 12. 13. 14.]
Scalar multiplication: [2. 4. 6. 8.]
Element-wise addition: [-2.   3.   3.5  2.5]
Element-wise subtraction: [4.  1.  2.5 5.5]
Element-wise multiplication: [-3.   2.   1.5 -6. ]
Element-wise division: [-0.33333333  2.          6.         -2.66666667]
Element-wise integer division: [-1.  2.  6. -3.]
```

```
print(np.floor(2.7))
x = np.array([3.2, 0.3, -1, 0.8])
print(np.sum(x))
print(np.mean([2,3]))
```

Output:

```
2.0
3.3
2.5
```

6.1.5 Array slicing

```
# Creating a 1D array
a = np.array([0.3, 1.5, 2.3, 3.2, 4.5, 5.1, 5.9, 7.3])
print(a)

# Slicing from index 2 to 5
sliced_a = a[2:6]
print(sliced_a)

# Slicing from the beginning to index 5
sliced_a = a[:6]
print(sliced_a)

# Slicing from index 5 to the end
sliced_a = a[5:]
print(sliced_a)

# Slicing with a step of 2
sliced_a = a[::2]
print(sliced_a)

# Slicing with a negative step (reversing the array)
sliced_a = a[::-1]
print(sliced_a)
```

Output:

```
[0.3 1.5 2.3 3.2 4.5 5.1 5.9 7.3]
[2.3 3.2 4.5 5.1]
[0.3 1.5 2.3 3.2 4.5 5.1]
[5.1 5.9 7.3]
[0.3 2.3 4.5 5.9]
[7.3 5.9 5.1 4.5 3.2 2.3 1.5 0.3]
```

6.1.6 Conditional array slicing

```
# Creating an array
x = np.array([-1, 0.5, -1.5, 2.5, 3])

# Create a copy to avoid modifying the original array
r = x.copy()

# Apply first condition: set all negative values to 0.0
r = np.where(x < 0.0, 0.0, r)
print(r)

# Apply third condition: transform values between 1 (inclusive) and 2 (exclusive) to 10 - x
r = np.where(np.logical_and(1 <= x, x < 2), 10 - x, r)
print(r)

# Apply fourth condition: set all values greater than or equal to 2 to 0.0
r = np.where(x >= 2, 0.0, r)
print(r)  # Output: array([0. , 0.5, 0.5, 0. , 0. ])
```

```
# Apply fifth condition: set values less than 0.5 or greater than 2 to -1.0
r = np.where(np.logical_or(x < 0.5, x > 2), -1.0, r)
print(r)  # Output: array([-1. ,  0.5,  0.5, -1. , -1. ])
```

Output:

```
[0.  0.5 0.  2.5 3. ]
[0.  0.5 0.  2.5 3. ]
[0.  0.5 0.  0.  0. ]
[-1.   0.5 -1.  -1.  -1. ]
```

```
# Creating an array
x = np.array([-1.33, 0.65, -1.56, 2.53, 5.58, 0.12, 2.44])

# Create a copy to avoid modifying the original array
r = x.copy()
print(r)

# Set all negative values to -0.5
print(x < 0.0)
r[x < 0.0] = -0.5
print(r)

# Set values between 0 and 1 to 0.5
flag = np.logical_and(x < 1, x > 0)
print(flag)
r[flag] = 0.5
print(r)

# Remove all values greater than 3
r = r[r <= 3]
print(r)

# Flip sign of values greater less than 0 or greater than 2.
flag = np.logical_or(r < 0.0, r > 2)
r[flag]= -r[flag]
print(flag)
print(r)
```

Output:

```
[-1.33  0.65 -1.56  2.53  5.58  0.12  2.44]
[ True False  True False False False False]
[-0.5   0.65 -0.5   2.53  5.58  0.12  2.44]
[False  True False False False  True False]
[-0.5   0.5  -0.5   2.53  5.58  0.5   2.44]
[-0.5   0.5  -0.5   2.53  0.5   2.44]
[ True False  True  True False  True]
[ 0.5   0.5   0.5  -2.53  0.5  -2.44]
```

```
# Creating an array
x = np.array([-1, 0.5, -1.5, 2.5, 5.5, 0.1, 2.4])

# where function to perform conditional operations
y = np.where(x < 0.0, 0.0, x)
print(y)
y= np.where(np.logical_and(x < 1, x > 0), 1.0, y)
print(y)
```

Output:

```
[0.  0.5 0.  2.5 5.5 0.1 2.4]
[0.  1.  0.  2.5 5.5 1.  2.4]
```

6.2 Mathematical functions

Mathematical functions can be performed on `x`, provided it is of type `numpy.ndarray`, or of type `float`, `int`, or a `list` thereof.

- Trigonometric Functions
  - `np.sin(x)`, `np.cos(x)`, `np.tan(x)`
  - `np.arcsin(x)`, `np.arccos(x)`, `np.arctan(x)`
- Logarithmic and Exponential Functions
  - `np.log(x)`, `np.log10(x)`, `np.exp(x)`
- Aggregation Functions
  - `np.sum(x)`, `np.mean(x)`, `np.median(x)`, `np.std(x)`
- Rounding Functions
  - `np.round(x)`, `np.ceil(x)` (round up), `np.floor(x)` (round down)

Example. Evaluate, for all integers x between 0 and 10:

$$y_1 = x^2 - 5x$$
$$y_2 = \sqrt{2x+1}$$
$$y_3 = \cos(2\pi x)\exp(-0.1x)$$

```
# Define x array
x = np.linspace(0,10,11)

y1 = x**2-5*x
print(y1)

y2 = np.sqrt(2*x+1)
y2 = np.round(y2, 3) # Round array values to 3 d.p.
print(y2)


y3 = np.cos(2*np.pi*x)*np.exp(-0.1*x)
y3 = np.round(y3, 3)
print(y2)
```

Output:

```
[ 0. -4. -6. -6. -4.  0.  6. 14. 24. 36. 50.]
[1.    1.732 2.236 2.646 3.    3.317 3.606 3.873 4.123 4.359 4.583]
[1.    0.905 0.819 0.741 0.67  0.607 0.549 0.497 0.449 0.407 0.368]
```

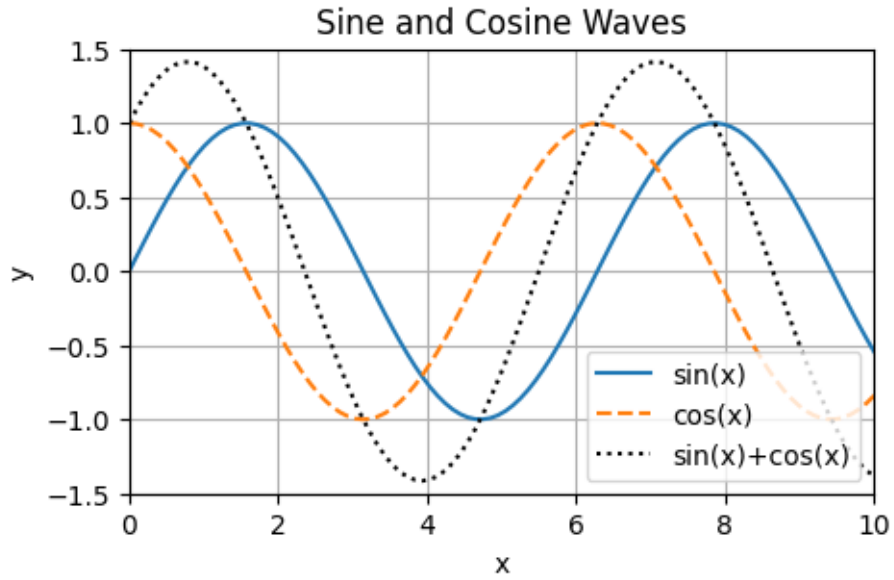
6.3 Graphs with `matplotlib`

```
import matplotlib.pyplot as plt
```

6.3.1 Basic line graph

```
# Creating data
x = np.linspace(0, 10, 100)
y1 = np.sin(x)
y2 = np.cos(x)
y3 = np.cos(x)+np.sin(x)

# Creating the plot
plt.figure(figsize=(5, 3))
```

```
plt.plot(x, y1, label="sin(x)")
plt.plot(x, y2, label="cos(x)", linestyle='--')
plt.plot(x, y3, label="sin(x)+cos(x)", linestyle=':', color="black")

# Adding title and labels
plt.title("Sine and cosine curves")
plt.xlabel("x")
plt.ylabel("y")

# Adjusting graph limits
plt.xlim([0, 10])
plt.ylim([-1.5, 1.5])

# Adding a legend
plt.legend()

# Adding grid lines
plt.grid(True)

# Display the plot
plt.show()
```




```
<Figure size 500x300 with 1 Axes>
```

6.4 Exercise 6.1 The half-wave rectifier

If an AC signal, like a sine wave for example, is sent through a diode any negative component to the signal is clipped out. The signal becomes

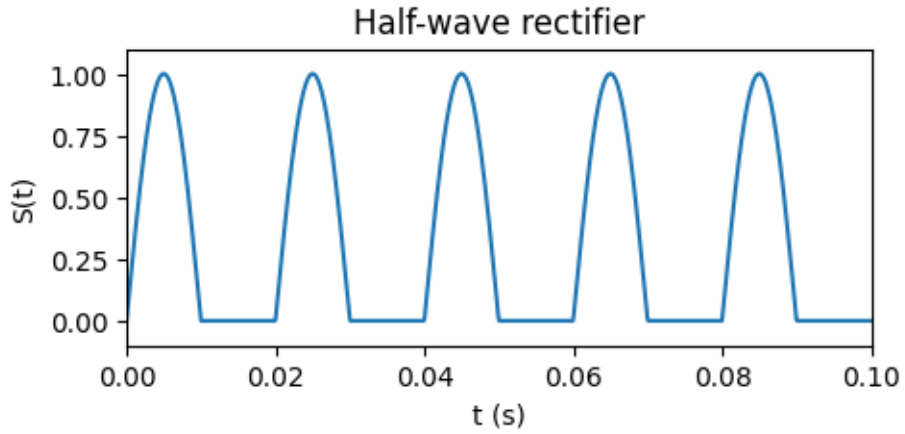
$$S(t) = \begin{cases} A\sin(2\pi f t) & \text{if } \sin(2\pi f t) > 0 \\ 0 & \text{otherwise} \end{cases}$$

Plot $S(t)$ vs time t for a total of 0.1 seconds taking $f = 50$ Hz and $A = 1$ Use 1000 points to create the plot.

Solution

```
import numpy as np
import matplotlib.pyplot as plt
```

```
# Parameters
A = 1
f = 50
t = np.linspace(0, 0.1, 1000)  # 1000 points from 0 to 0.1 seconds

# Create the signal
S = A * np.sin(2 * np.pi * f * t)

# Clip the negative part of the signal
S_clipped = np.where(S > 0, S, 0)

# Plot the S_clipped signal
plt.figure(figsize=(5, 2))
plt.plot(t, S_clipped, label="Rectified AC Signal")
# Adding title and labels
plt.title("Half-wave rectifier")
plt.xlabel("t (s)")
plt.ylabel("S(t)")
# Adjusting graph limits
plt.xlim([0, 0.1])
plt.ylim([-0.1, 1.1])
# Display the plot
plt.show()
```




```
<Figure size 500x200 with 1 Axes>
```

6.5 Exercise 6.2 Charging of a capacitor

A series circuit containing only a resistor of resistance R, a capacitor of capacitance C, a switch, and a constant DC source of voltage V is known as a charging circuit. If the capacitor is initially uncharged while the switch is open, and the switch is closed at $t = 0$, it follows that the voltage across the capacitor is zero and the charge delivered to the capacitor obeys

$$Q(t) = CV\left(1 - e^{-t/RC}\right)$$

Circuit 1: $R = 100\,\Omega, C = 1\,\mu\text{F}, V = 5\,\text{V}$

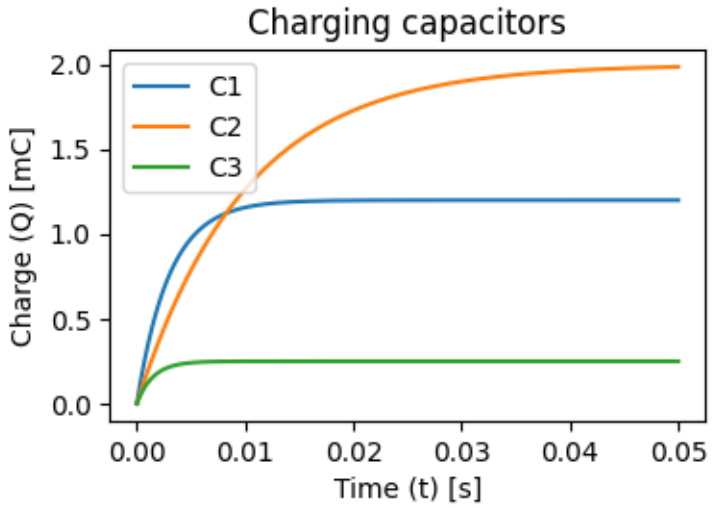
Circuit 2: $R = 50\,\Omega, C = 2\,\mu\text{F}, V = 10\,\text{V}$

Circuit 3: $R = 300\,\Omega, C = 0.5\,\mu\text{F}, V = 15\,\text{V}$

On the same graph, plot $Q(t)$ vs t for the three circuits.

Solution

```
import numpy as np
import matplotlib.pyplot as plt

# Define the function for Q(t)
def Q(t, R, C, V):
    return C * V * (1 - np.exp(-t / (R * C)))

# Time range
t = np.linspace(0, 0.05, 1000)  # 0 to 0.05 seconds

# Circuit 1 parameters
R1 = 30
C1 = 100e-6
V1 = 12
Q1 = Q(t, R1, C1, V1)

# Circuit 2 parameters
R2 = 50
C2 = 200e-6
V2 = 10
Q2 = Q(t, R2, C2, V2)

# Circuit 3 parameters
R3 = 300
C3 = 5e-6
V3 = 50
Q3 = Q(t, R3, C3, V3)

# Plotting
plt.figure(figsize=(4, 2.5))
plt.plot(t, Q1*1000, label='C1')
plt.plot(t, Q2*1000, label='C2')
plt.plot(t, Q3*1000, label='C3')

# Adding title and labels
plt.title('Charging capacitors')
plt.xlabel('Time (t) [s]')
plt.ylabel('Charge (Q) [mC]')

# Adding legend
plt.legend()

# Display the plot
plt.show()
```




```
<Figure size 400x250 with 1 Axes>
```

6.6 Exercise 6.3 Projectile motion with air resistance

In the motion of the projectile is affected by the drag force, the latter is often assumed proportional to the velocity of the projectile (this is just an approximation).

The equation of motion in x direction is

$$m\frac{\mathrm{d}v_x}{\mathrm{d}t} = mg - cv_x$$

and for the y-direction

$$m\frac{\mathrm{d}v_y}{\mathrm{d}t} = cv_y$$

where

- m is the mass of the projectile, in kg
- $g = 9.81$ m/s^2 is the acceleration due to gravity
- c is the drag constant, in kg/s
- t is the time, in s

It follows that the terminal velocty of the projectile is $V_T = mg/c$.

A projectile is projected at $t = 0$ with speed U at an angle α above the horizontal.

Integrating the equations of motion yields:

1. **Velocity in the x-direction**:

$$v_x(t) = U\cos(\alpha)e^{-gt/V_T}$$

2. **Velocity in the y-direction**:

$$v_y(t) = \left(\frac{UV_T}{g}\sin(\alpha) + V_T\right)e^{-gt/V_T} - V_T$$

3. **Position in the x-direction**:

$$x(t) = \frac{UV_T}{g}\cos(\alpha)\left(1 - e^{-gt/V_T}\right)$$

4. **Position in the y-direction**:

$$y(t) = \frac{V_T}{g}\left(U\sin(\alpha) - V_T\right)\left(1 - e^{-gt/V_T}\right) - V_T t$$

Using $U = 50$ m/s, $\alpha = 45$ degrees, $c = 0.1$ kg/s and $m = 1$ kg,

(i) Plot the projectile trajectory y vs x plot for $y > 0$.

(ii) On the same graph, plot the trajectory in the absence of drag (set $c = 0.0001$).

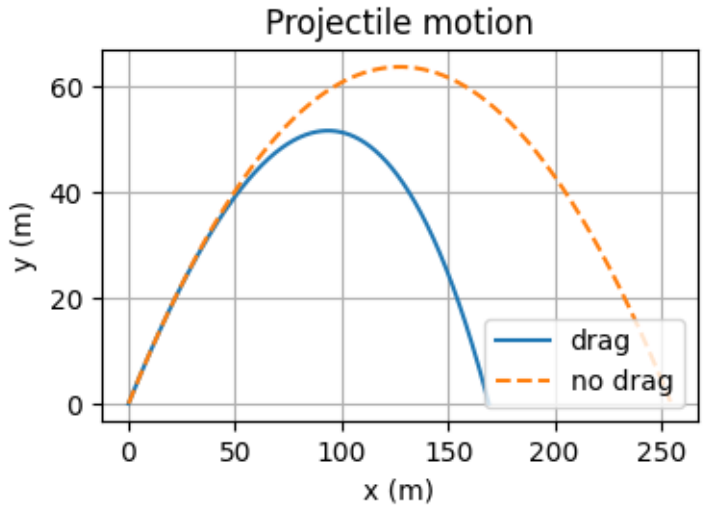
(iii) On a second graph, plot the K.E. of the projectile (both in the presence and absence of drag) as a function of time t for which $y > 0$. The K.E. can be computed as

$$E = \frac{1}{2}m\left(v_x^2 + v_y^2\right)$$

Solution

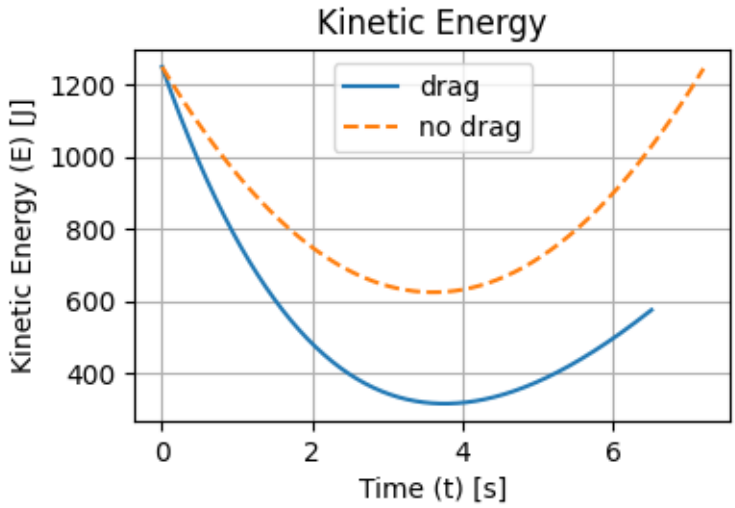
```
import numpy as np
import matplotlib.pyplot as plt

# Constants
U = 50  # initial speed in m/s
alpha = 45  # launch angle in degrees
g = 9.81  # acceleration due to gravity in m/s^2
m = 1  # mass of the projectile in kg

# Convert angle to radians
alpha_rad = np.radians(alpha)

# Time array
t = np.linspace(0, 10, 1000)  # 1000 points from 0 to 10 seconds

# Without drag (c = 0)
c = 0.0001  # small drag coefficient to approximate no drag
# Terminal velocity
V = m * g / c
# Velocity components without drag
v_x_nd = U * np.cos(alpha_rad) * np.exp(-g * t / V)
v_y_nd = (U * np.sin(alpha_rad) + V) * np.exp(-g * t / V) - V

# Position without drag
x_nd = (V / g) * U * np.cos(alpha_rad) * (1 - np.exp(-g * t / V))
y_nd = (V / g) * (U * np.sin(alpha_rad) + V) * (1 - np.exp(-g * t / V)) - V * t

# With drag
c = 0.1  # drag coefficient to approximate no drag
# Terminal velocity
V = m * g / c
# Velocity components
v_x = U * np.cos(alpha_rad) * np.exp(-g * t / V)
v_y = (U * np.sin(alpha_rad) + V) * np.exp(-g * t / V) - V

# Position components
x = (V / g) * U * np.cos(alpha_rad) * (1 - np.exp(-g * t / V))
y = (V / g) * (U * np.sin(alpha_rad) + V) * (1 - np.exp(-g * t / V)) - V * t

# Slicing to get only the valid points where y >= 0
flag = y >= 0
```

```
flag_nd = y_nd >= 0

x = x[flag]
y = y[flag]

x_nd = x_nd[flag_nd]
y_nd = y_nd[flag_nd]

# Plotting the trajectory with and without drag
plt.figure(figsize=(4, 2.5))
plt.plot(x, y, label='drag')
plt.plot(x_nd, y_nd, label='no drag', linestyle='dashed')
plt.title('Trajectories')
plt.xlabel('x (m)')
plt.ylabel('y (m)')
plt.legend()
plt.grid()
plt.show()
```




```
<Figure size 400x250 with 1 Axes>
```

```
# Kinetic Energies
E_d = 0.5 * m * (v_x[flag]**2 + v_y[flag]**2)
E_nd = 0.5 * m * (v_x_nd[flag_nd]**2 + v_y_nd[flag_nd]**2)

t_d = t[flag]
t_nd = t[flag_nd]
# Plotting the kinetic energy
plt.figure(figsize=(4, 2.5))
plt.plot(t_d, E_d, label='drag')
plt.plot(t_nd, E_nd, label='no drag', linestyle='dashed')
plt.title('Kinetic Energy')
plt.xlabel('Time (t) [s]')
plt.ylabel('Kinetic Energy (E) [J]')
plt.legend()
plt.grid()
plt.show()
```




```
<Figure size 400x250 with 1 Axes>
```

Chapter 7: Animating graphs

7.1 Context: travelling waves

The wave equation for a traveling wave can be represented as

$$u(x,t) = A\cos(kx - \omega t)$$

where:

- $u(x,t)$ is the displacement of strength of the wave at position x and time t,
- A is the amplitude (maximum displacment or strength) of the wave,
- $\omega = 2\pi f$ is the angular frequency, where the frequency is related to the wavespeed v and wavelength λ as $f = v/\lambda$.
- $k = 2\pi/\lambda = \omega/v$ is the wave number.

In the context of **radiowaves**, $u(x,t)$ typically represents the electric field strength (in Volts) of the radiowave.

The code below plots:

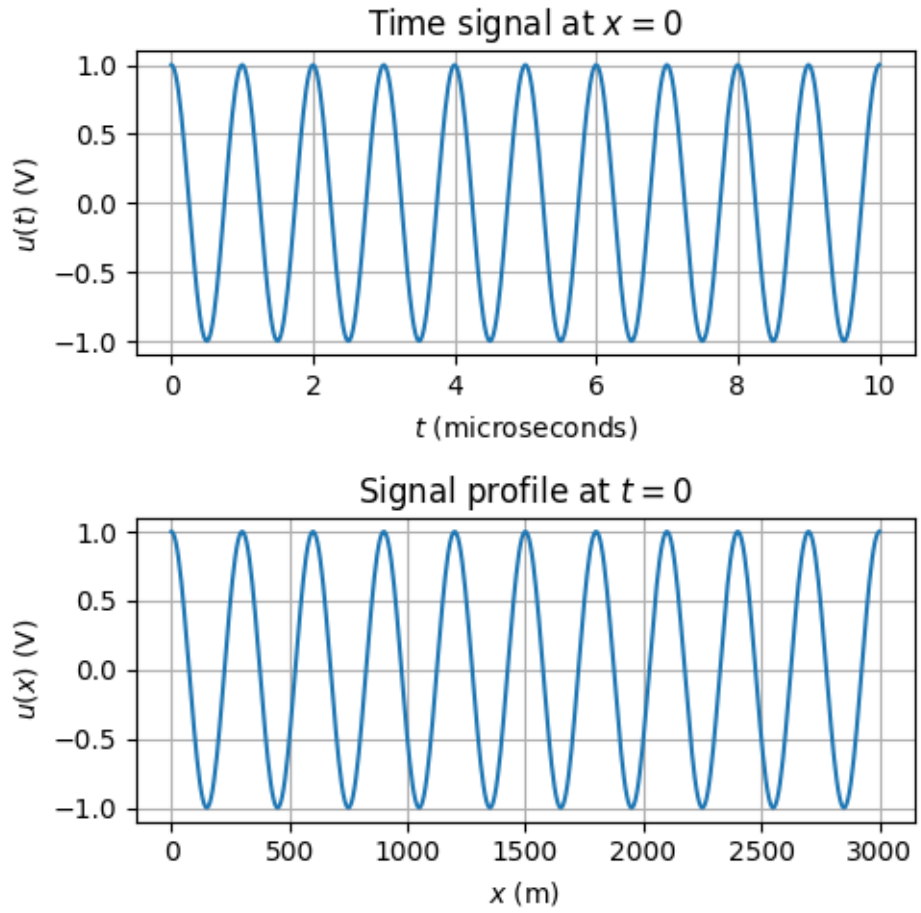
(i) u as a function of t (by setting $x = 0$). This represents the radio wave strength over time at a particular point in space,

(ii) u as a function of x (by setting $t = 0$). This represents the radio wave strength over space at a particular point in time,

for a set of typical values: $A = 1$ V, $f = 1$ MHz, $v = 3 \times 10^8$ m/s.

```
import numpy as np
import matplotlib.pyplot as plt

# Given parameters
A = 1  # amplitude in volts
f = 1e6  # frequency in Hz (1 MHz)
v = 3e8  # speed of the wave in m/s

# Derived parameters
omega = 2 * np.pi * f  # angular frequency
wavelength = v/f
k = 2 * np.pi / wavelength  # wave number

# Time plot parameters
t = np.linspace(0, 10e-6, 500)  # time from 0 to 10 microseconds

# Space plot parameters
x = np.linspace(0, 3e3, 500)  # distance from 0 to 3 km meters

# Wave equations
u_t = A * np.cos(-omega * t)  # at x = 0
u_x = A * np.cos(k * x)  # at t = 0

# Create subplots
fig, axs = plt.subplots(2, 1, figsize=(5, 5))

# Plot u as a function of t
axs[0].plot(t*1e6, u_t)
axs[0].set_title('Time signal at $x = 0$')
axs[0].set_xlabel('$t$ (microseconds)')
axs[0].set_ylabel('$u(t)$ (V)')
```

```
axs[0].grid(True)

# Plot u as a function of x
axs[1].plot(x, u_x)
axs[1].set_title('Signal profile at $t = 0$')
axs[1].set_xlabel('$x$ (m)')
axs[1].set_ylabel('$u(x)$ (V)')
axs[1].grid(True)

# Adjust layout
plt.tight_layout()
plt.show()
```




```
<Figure size 500x500 with 2 Axes>
```

7.2 Exercise 1: Travelling radiowave

For the values $A = 1$ V, $f = 1$ MHz, $v = 3 \times 10^8$ m/s, plot the wave

$$u(x,t) = A\cos(kx - \omega t)$$

as a function of x, for $0 \leq x \leq 1000$ m for three instances of time:

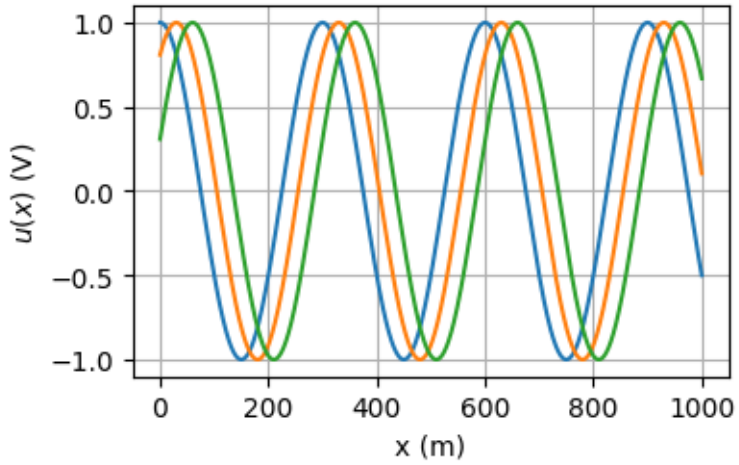
(i) $t = 0$

(ii) $t = 0.1\ \mu$s

(iii) $t = 0.2\ \mu$s

Solution

```
import numpy as np
import matplotlib.pyplot as plt

def wave(A,f,v,t, x):
  # Derived parameters
  omega = 2 * np.pi * f
  wavelength = v / f
  k = 2 * np.pi / wavelength
  return A * np.cos(k * x -omega * t)

# Given parameters
A = 1  # amplitude in volts
f = 1e6  # frequency in Hz (1 MHz)
v = 3e8  # speed of the wave in m/s

x = np.linspace(0, 1e3, 1000)  # x array from 0 to 3 km

# Plot u as a function of x
plt.figure(figsize=(4, 2.5))
plt.plot(x, wave(A,f,v,0, x))
plt.plot(x, wave(A,f,v,0.1e-6, x))
plt.plot(x, wave(A,f,v,0.2e-6, x))
plt.xlabel('$x$ (m)')
plt.ylabel('$u(x)$ (V)')
plt.grid(True)
plt.show()
```




```
<Figure size 400x250 with 1 Axes>
```

7.3 Travelling wave animation

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation

def wave(A,f,v,t,x):
  omega = 2 * np.pi * f
  wavelength = v / f
  k = 2 * np.pi / wavelength
  return A * np.cos(k * x -omega * t)
```

```
# Given parameters
A = 1  # amplitude in volts
f = 1e6  # frequency in Hz (1 MHz)
v = 3e8  # speed of the wave in m/s

# Set up the figure, the axis, and the plot element we want to animate
fig, ax = plt.subplots()
x = np.linspace(0, 1e3, 1000)
line, = ax.plot(x, wave(A,f,v,0,x))

# Initialization function: plot the background to be used by each frame
def init():
    line.set_ydata(wave(A,f,v,0,x))
    ax.set_xlabel('Position (m)')
    ax.set_xlim(0, 1000)
    ax.set_ylabel('Wave Amplitude (V)')
    ax.set_title('Travelling Radio Wave')
    ax.grid(True)
    return line,

# Animation function: this is called sequentially
def animate(i):
    t = i*2e-8
    line.set_ydata(wave(A,f,v,t,x))  # update the data
    return line,

# Call the animator. blit=True means only re-draw the parts that have changed.
ani = animation.FuncAnimation(fig, animate, init_func=init,
      frames=200, interval=20, blit=True)

# Save the animation as an MP4 file
ani.save('radiowave_animation.mp4', writer='ffmpeg')
```

7.4 Exercise 2: Amplitude Modulated (AM) wave

The amplitude-modulated wave equation can be written as

$$u(x,t) = A\cos(k_0 x - \omega_0 t)\cos(kx - \omega t)$$

where:

- A is the amplitude of the wave.
- ω_0 is the angular frequency of the modulating signal.
- k_0 is the wave number of the modulating signal.
- $\omega = 2\pi f$ is the angular frequency of the carrier signal.
- $k = 2\pi/\lambda = \omega/v$ is the wave number of the carrier signal.
- v is the speed of the wave.
- x is the position
- t is the time .

7.4.1 Task 1

Consider a modulating signal with frequency $f_0 = 50$ kHz, amplitude $A = 1$ V, carrier frequency $f = 1$ MHz, and $v = 3 \times 10^8$ m/s.

Hence,

\$f/f_0 = 20 \$, $\omega/\omega_0 = 20$ and $k/k_0 = 20$.

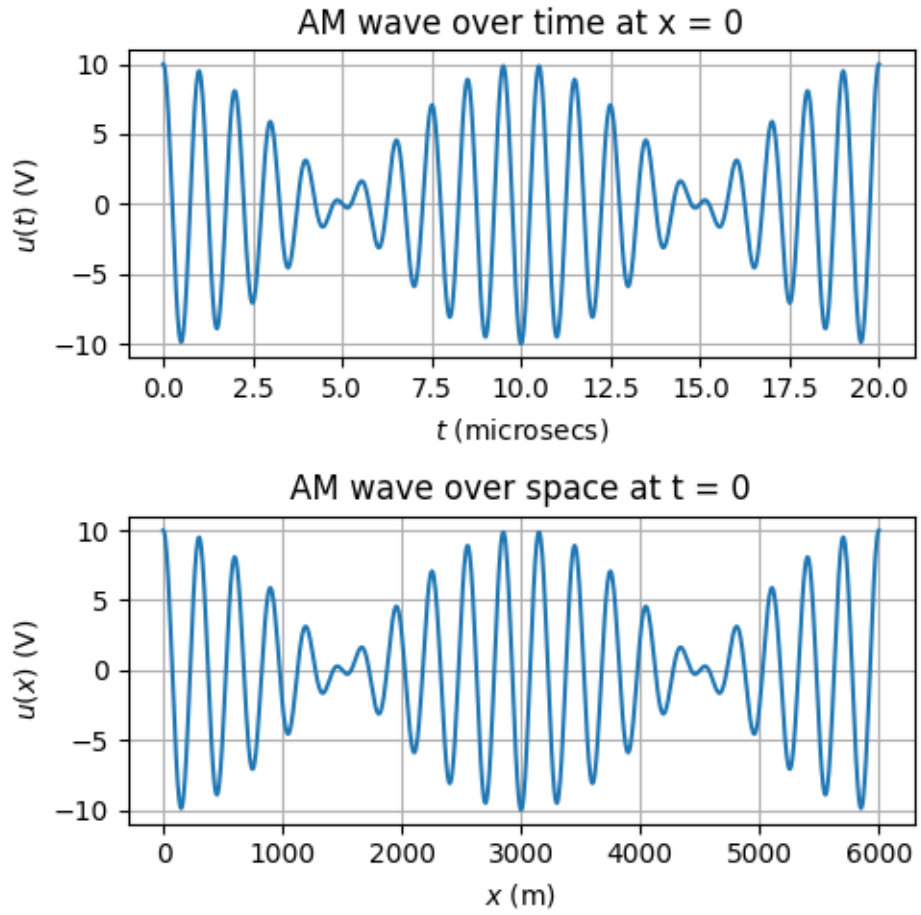
(i) Plot $u(0,t)$ as a function of t for $0 \leq t \leq 20$ μs.

(ii) Plot $u(x, 0)$ as a function of x for $0 \leq x \leq 6$ km.

Hint: Both plots should look the same.

Solution

```
import numpy as np
import matplotlib.pyplot as plt

# Given parameters
A = 10  # amplitude in volts

f = 1e6  # frequency in Hz (1 MHz)
v = 3e8  # phase velocity in m/s

# Time plot parameters
t = np.linspace(0, 20e-6, 1000)  # time from 0 to 20 microseconds

# Space plot parameters
x = np.linspace(0, 6e3, 1000)  # distance from 0 to 300 meters
t_fixed = 0  # t = 0

# Wave equations
def am_wave(A, f, v, t, x):
    omega = 2 * np.pi * f
    wavelength = v / f
    k = 2 * np.pi / wavelength
    omega_0 = omega/20
    k_0 = k/20
    return (A*np.cos(k_0 * x - omega_0 * t)) * np.cos(k * x - omega * t)

u_t = am_wave(A, f, v, t, 0)  # at x = 0
u_x = am_wave(A, f, v, 0, x)  # at t = 0

# Create subplots
fig, axs = plt.subplots(2, 1, figsize=(5, 5))

# Plot u as a function of t
axs[0].plot(1e6*t, u_t)
axs[0].set_title('AM wave over time at x = 0')
axs[0].set_xlabel('$t$ (microsecs)')
axs[0].set_ylabel('$u(t)$ (V)')
axs[0].grid(True)

# Plot u as a function of x
axs[1].plot(x, u_x)
axs[1].set_title('AM wave over space at t = 0')
axs[1].set_xlabel('$x$ (m)')
axs[1].set_ylabel('$u(x)$ (V)')
axs[1].grid(True)

# Adjust layout
plt.tight_layout()
plt.show()
```




```
<Figure size 500x500 with 2 Axes>
```

7.4.2 Task 2

Make an **animation** of the AM signal from Task 1. Copy and adapt the code from the **Travelling wave animation**.

- The animator settings can remain the same.
- A timestep of 1e-7 seconds is recommended.
- The x-axis should span 10 km.

Solution

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation

# Wave equation
def am_wave(A, f, v, t, x):
    omega = 2 * np.pi * f
    wavelength = v / f
    k = 2 * np.pi / wavelength
    omega_0 = omega/20
    k_0 = k/20
    return (A*np.cos(k_0 * x - omega_0 * t)) * np.cos(k * x - omega * t)


# Given parameters
A = 1  # amplitude in volts
f = 1e6  # frequency in Hz (1 MHz)
```

```
v = 3e8  # speed of the wave in m/s

# Set up the figure, the axis, and the plot element we want to animate
fig, ax = plt.subplots()
x = np.linspace(0, 1e4, 1000)
line, = ax.plot(x/1e3, am_wave(A,f,v,0,x))

# Initialization function: plot the background to be used by each frame
def init():
    line.set_ydata(am_wave(A,f,v,0,x))
    ax.set_xlabel('Position (km)')
    ax.set_xlim(0, 1e1)
    ax.set_title('AM Radio Wave')
    ax.grid(True)
    return line,

# Animation function: this is called sequentially
def animate(i):
    t = i*1e-7
    line.set_ydata(am_wave(A,f,v,t,x))  # update the data
    return line,

# Call the animator.
ani = animation.FuncAnimation(fig, animate, init_func=init,
      frames=200, interval=20, blit=True)

# Save the animation as an MP4 file
ani.save('am_radiowave_animation.mp4', writer='ffmpeg')

plt.show()
```

Chapter 8: Numerical integration

8.1 The Trapezium Rule

The trapezium rule is a numerical method for approximating the definite integral of a function. It is particularly useful when an exact integral is difficult or impossible to compute analytically.

Given a function $f(x)$ that is continuous on the interval $[a, b]$, the trapezium rule approximates the integral

$$\int_a^b f(x)\,dx$$

by dividing the interval into $ n $ equal subintervals, each of width $ h$:

$$\int_a^b f(x)\,dx \approx \frac{h}{2}\left[f(x_0) + 2\sum_{i=1}^{n-1} f(x_i) + f(x_n)\right]$$

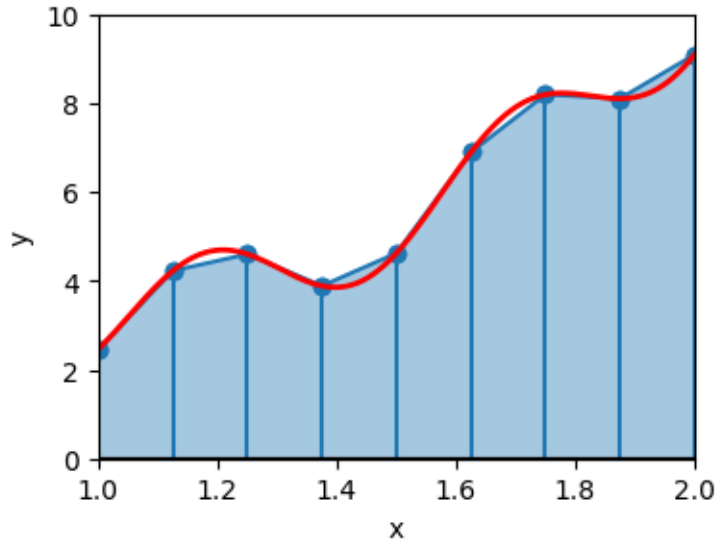
where - $x_0 = a$, $x_1 = a + h$, $x_2 = a + 2h$, ..., $x_n = b$ are the points dividing the interval, - $f(x_i)$ is the function value at each point x_i, and - $h = (b - a)/n$.

```
import numpy as np
import matplotlib.pyplot as plt

def function(x):
    """Function y = f(x) to integrate"""
    f = np.sin(12*x) + x**(3)+2
    return f

def trapezium_rule(f, a, b, n):
    """
    Approximates the integral of a function f from a to b using the trapezium rule with n subinterva

    Parameters:
    f (function): The function to integrate.
    a (float): The start point of the interval.
    b (float): The end point of the interval.
    n (int): The number of subintervals.

    Returns:
    area (float): The approximate value of the integral.
    x (np.array): Array containing x values
    y (np.array): Array containing the function (y) values
    """
    # Calculate the width of each subinterval
    h = (b - a) / n

    # Calculate the x values
    x = np.linspace(a, b, n+1)

    # Calculate the function values at the x points
    y = f(x)

    # Apply the trapezium rule formula
    area = (h / 2) * (y[0] + 2 * np.sum(y[1:n]) + y[n])
```

```
    return area, x, y

# Inputs:
a = 1
b = 2
n = 8

# Integrate y = function(x) using the Trapezium Rule
area, x, y = trapezium_rule(function, a, b, n)

# Print Area
print(f"Area = {area:.4f}")

# Plot Graph
xc = np.linspace(a, b, 1000)
yc = function(xc)

plt.figure(figsize=(4,3))
plt.fill_between(x, y, alpha=0.4)
plt.plot(x, y, 'o-')
plt.stem(x, y)
plt.plot(xc, yc, '-r', linewidth=2)
plt.plot([a, b], [0, 0], '-k')
plt.xlim([a, b])
plt.ylim([min(np.append(1.1*yc, 0)), max(np.append(1.1*yc, 0))])
plt.xlabel("x")
plt.ylabel("y")
plt.show()
```

Output:

```
Area = 5.7899
```




```
<Figure size 400x300 with 1 Axes>
```

8.2 Finite differences and the Forward Euler method

Consider the differential equation

$$u' = f(t, u)$$

Tthe equation in discretized in time. The value of the derivative the time step n is

$$u'_n = f(t_n, u_n)$$

The derivative can be approximated by the method of **finite differences**:

$$u'_n = \frac{u_{n+1} - u_n}{\Delta t}$$

Here, - u_{n+1} is the solution at the next time step. - u_n is the solution at the current time step. - t_n is the time of the current time step. - Δt is the time step size.

Rearranging the above equation to solve for u_{n+1} gives the **Forward Euler** update rule:

$$u_{n+1} = u_n + \Delta t \cdot f(t_n, u_n)$$

In the Forward Euler method, the next value of u is computed using the current value of u, the step size Δt, and the derivative at the current time step. This method is simple and straightforward but can be less accurate for stiff or highly dynamic systems.

8.2.1 Example: linear ODE

Using the Forward Euler method, solve the ODE:

$$u' = 2u - 1$$

subject to $u(0) = 2$ for the time interval $t \in [0, 2]$.

Plot the numerical solution together with the exact solution

$$u(t) = \frac{1}{2} + \frac{3}{2}e^{2t}$$

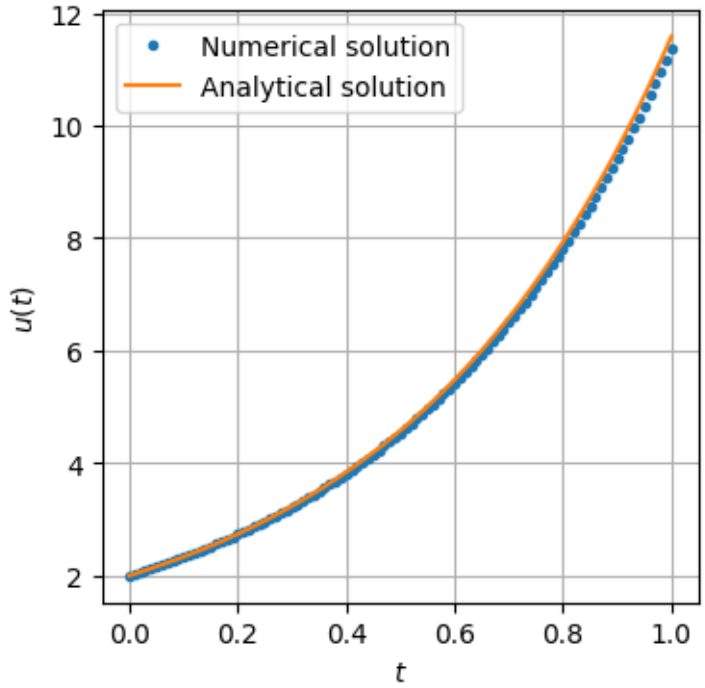
Solution The Forward Euler update equation for this ODE becomes:

$$u_{n+1} = u_n + \Delta t \cdot (2u_n - 1)$$

The time-step should not be larger than $\Delta t = 0.01$.

```
import numpy as np
import matplotlib.pyplot as plt

# Given parameters
u0 = 2            # initial condition
t_start = 0       # initial time
t_max = 1         # maximum time
dt = 0.01        # time step size
t = np.arange(t_start, t_max + dt, dt)  # time array

# Create solution array
u = np.zeros(len(t))
# Initialise solution (set first value)
u[0] = u0

# Forward Euler method for non-linear equation
for n in range(1, len(t)):
    u[n] = u[n-1] + dt * (2 * u[n-1] - 1)
```

```
# Exact solution
u_exact = 0.5 + 1.5 * np.exp(2 * t)

# Plot solutions
plt.figure(figsize=(4, 4))
plt.plot(t, u, '.', label='Numerical solution')
plt.plot(t, u_exact, '-', label='Analytical solution')
plt.xlabel('$t$')
plt.ylabel('$u(t)$')
plt.legend()
plt.grid(True)
plt.show()
```




```
<Figure size 400x400 with 1 Axes>
```

8.3 Exercise 1: Solving a first-order ODE

Using the Forward Euler method, solve the ODE:

$$y' + y/x = \cos(x)$$

subject to $y(\pi/2) = 2$ for the time interval $x \in [\pi/2, 7\pi/2]$.

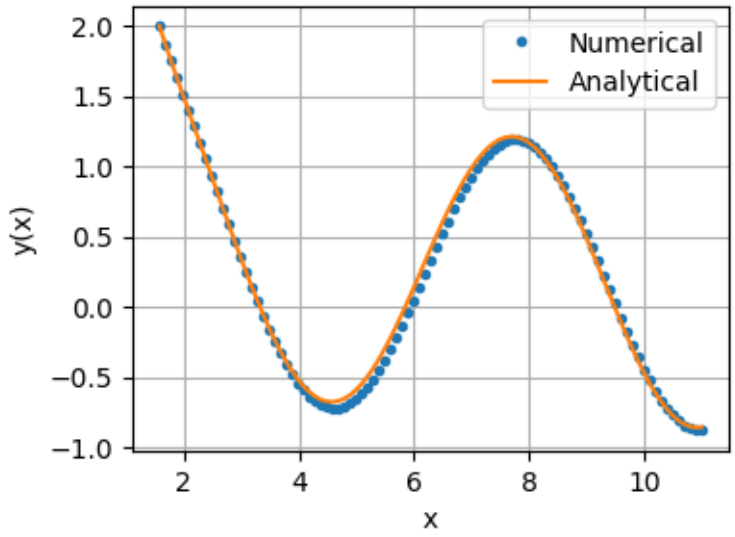
Plot the numerical solution and compare it with the exact solution

$$y(t) = \sin(x) + \frac{\cos(x) + \pi/2}{x}$$

Solution

```
import numpy as np
import matplotlib.pyplot as plt

# Define the parameters
y0 = 2
x0 = np.pi / 2
```

```
x_end = 7*np.pi / 2
dx = 0.1
N = int((x_end - x0) / dx)

# Initialize arrays for x and y
x = np.linspace(x0, x_end, N+1)
y = np.zeros(N+1)
y[0] = y0

# Forward Euler method
for n in range(N):
    y[n+1] = y[n] + dx * (np.cos(x[n]) - y[n] / x[n])

# Exact solution
def exact_solution(x):
    return np.sin(x) + (np.cos(x) + np.pi/2) / x

# Plotting the numerical and exact solutions
plt.figure(figsize=(4,3))
plt.plot(x, y, '.', label='Numerical')
plt.plot(x, exact_solution(x), label='Analytical', linestyle='-')
plt.xlabel('x')
plt.ylabel('y(x)')
plt.legend()
plt.grid(True)
plt.show()
```




```
<Figure size 400x300 with 1 Axes>
```

8.4 Exercise 2: Motion of a simple pendulum

The equation of motion for a simple pendulum:

$$\frac{d^2\theta}{dt^2} + \frac{g}{L}\sin\theta = 0$$

where

- θ is the angle that the pendulum makes with the vertical
- g is the acceleration due to gravity
- L is the length of the pendulum

To integrate numerically a second-order ODE (such as the one above) we must rewrite it as a system of first-order ODEs.

Let $\theta_1 = \theta$ and $\theta_2 = \frac{d\theta}{dt}$. Then we can write:

$$\frac{d\theta_1}{dt} = \theta_2$$

$$\frac{d\theta_2}{dt} = -\frac{g}{L}\sin(\theta_1)$$

Using the Forward Euler method, solve the ODE system above for $t \in [0, 7\pi/8]$ s.

Take

- $g = 9.81$ m/s^2
- $L = 1.0$ m

Initial conditions ($t = 0$) are:

- $\theta = \theta_0 = \frac{\pi}{4}$ (initial angle)
- $\frac{d\theta}{dt} = 0$ (no initial angular velocity)

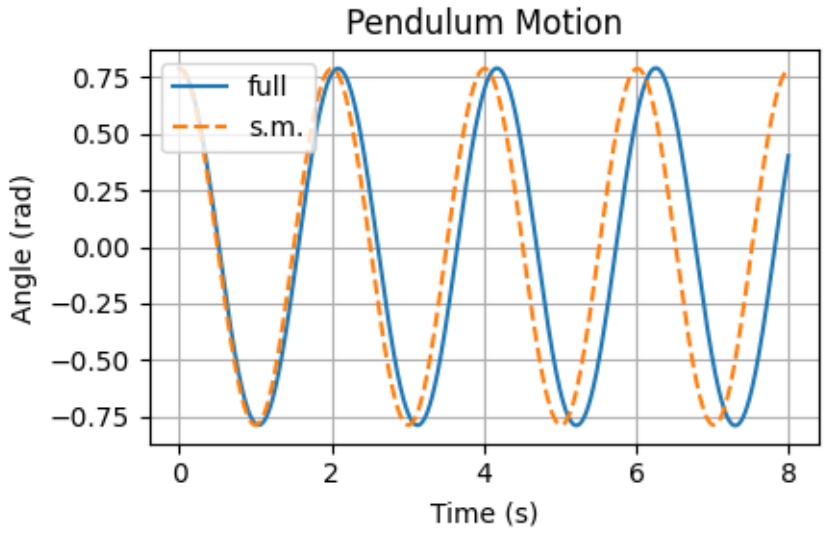
Plot θ vs t and compare it with the small-angle approximation solution:

$$\theta \approx \theta_0 \cos(\sqrt{g/L}t)$$

Solution

```
import numpy as np
import matplotlib.pyplot as plt

# Given parameters
g = 9.81  # acceleration due to gravity (m/s^2)
L = 1.0   # length of the pendulum (m)
theta0 = np.pi / 4  # initial angle (radians)
omega0 = 0.0        # initial angular velocity (rad/s)

# Time parameters
t_max = 8  # maximum time (s)
dt = 0.0001   # time step (s)
t = np.arange(0, t_max, dt)  # time array

# Forward Euler method for non-linear equation
theta1 = np.zeros(len(t))
theta2 = np.zeros(len(t))
theta1[0] = theta0
theta2[0] = omega0

for i in range(1, len(t)):
    theta2[i] = theta2[i-1] - (g / L) * np.sin(theta1[i-1]) * dt
    theta1[i] = theta1[i-1] + theta2[i-1] * dt

# Analytical solution for small-angle approximation
theta_sm = theta0 * np.cos(np.sqrt(g / L) * t)

# Plot results
plt.figure(figsize=(4.5, 3))

plt.plot(t, theta1, label='full')
```

```
plt.plot(t, theta_sm, label='s.m.', linestyle='dashed')
plt.xlabel('Time (s)')
plt.ylabel('Angle (rad)')
plt.title('Pendulum Motion')
plt.legend(loc="upper left")
plt.grid(True)
plt.tight_layout()
plt.show()
```




```
<Figure size 450x300 with 1 Axes>
```

Chapter 9: Data analysis

9.1 Pandas Dataframes

```
import pandas as pd
```

9.1.1 Creating dataframes

```
# Creating a DataFrame from a dictionary of lists
data = {
    'Metal': ['Aluminum', 'Copper', 'Iron', 'Titanium', 'Nickel'],
    'Density': [2.70, 8.96, 7.87, 4.54, 8.90],
    'Young Modulus': [69, 110, 210, 116, 200]
}
df = pd.DataFrame(data)
print("------")
print(df)
print("------")
# Checking the size of the DataFrame
size = df.shape
print(f"The DataFrame has {size[0]} rows and {size[1]} columns.")
```

Output:

```
------
      Metal  Density  Young Modulus
0  Aluminum     2.70             69
1    Copper     8.96            110
2      Iron     7.87            210
3  Titanium     4.54            116
4    Nickel     8.90            200
------
The DataFrame has 5 rows and 3 columns.
```

```
print("------")
print(df.head(2))  # First 2 rows
print("------")
print(df.tail(3))  # Last 3 rows
```

Output:

```
------
      Metal  Density  Young Modulus
0  Aluminum     2.70             69
1    Copper     8.96            110
------
      Metal  Density  Young Modulus
2      Iron     7.87            210
3  Titanium     4.54            116
4    Nickel     8.90            200
```

Setting the index to a specific column

```
# Setting the 'Metal' column as the index
df.set_index('Metal', inplace=True)
print("New dataframe:")
print(df)
```

Output:

```
New dataframe:
          Density  Young Modulus
Metal
Aluminum     2.70             69
Copper       8.96            110
Iron         7.87            210
Titanium     4.54            116
Nickel       8.90            200
```

9.1.2 Accessing and editing content

Accessing columns

```
print("------")
print(df.head(2))  # First 2 rows
print("...")
print("------")

# Accessing a column by index
name_column = df['Density']
print(name_column)
print(type(name_column))
print("------")

# Accessing a column by index
young_column = df.iloc[:, 1]  # Access the second column
print(young_column)
print("-----")
```

Output:

```
------
          Density  Young Modulus
Metal
Aluminum     2.70             69
Copper       8.96            110
...
------
Metal
Aluminum    2.70
Copper      8.96
Iron        7.87
Titanium    4.54
Nickel      8.90
Name: Density, dtype: float64
<class 'pandas.core.series.Series'>
------
Metal
Aluminum     69
Copper      110
Iron        210
Titanium    116
Nickel      200
Name: Young Modulus, dtype: int64
-----
```

Accessing rows and extracting Series

```
print("------")
print(df.head(3))  # First 3 rows for reference
```

```
print("...")
print("------")

iron_row = df.loc["Iron"] # Extract iron row by index label
print(iron_row)
print(type(iron_row))
print("------")

fourth_row = df.iloc[3] # Fourth row by index
print(fourth_row)
print(type(fourth_row))
print("------")

# Extract series keys and values as lists
categories = list(fourth_row.keys())
values = list(fourth_row.values)
print(categories, values)
```

Output:

```
------
          Density  Young Modulus
Metal
Aluminum     2.70             69
Copper       8.96            110
Iron         7.87            210
...
------
Density            7.87
Young Modulus    210.00
Name: Iron, dtype: float64
<class 'pandas.core.series.Series'>
------
Density            4.54
Young Modulus    116.00
Name: Titanium, dtype: float64
<class 'pandas.core.series.Series'>
------
['Density', 'Young Modulus'] [4.54, 116.0]
```

Adding and removing columns

```
# Make a copy of original df
df2 = df.copy()

# Adding a new column at index=1
df2.insert(1, 'Melting Point', [660, 1085, 1538, 1668, 1455])
print('------')
print(df2)

# Removing a column by name
df3 = df2.copy()
df3 = df3.drop(columns=['Melting Point'])
print("------")
print(df3)

# Removing a column by index
df3 = df2.copy()
# Remove the
```

```
df3 = df3.drop(df3.columns[0:2], axis=1)  # Remove 1st and 2nd cols
print("------")
print(df3)
```

Output:

```
------
          Density  Melting Point  Young Modulus
Metal
Aluminum     2.70            660             69
Copper       8.96           1085            110
Iron         7.87           1538            210
Titanium     4.54           1668            116
Nickel       8.90           1455            200
------
          Density  Young Modulus
Metal
Aluminum     2.70             69
Copper       8.96            110
Iron         7.87            210
Titanium     4.54            116
Nickel       8.90            200
------
          Young Modulus
Metal
Aluminum             69
Copper              110
Iron                210
Titanium            116
Nickel              200
```

Removing rows

```
# Make a copy of original df
df2 = df.copy()

# Remove rows by their index (for example, rows with index 1 and 3)
df2 = df2.drop(index=["Iron", "Aluminum"])
print("------")
print(df2)

# Remove rows by their integer location (for example, rows at index 1 and 3)
df2 = df.copy()
indices_to_keep = [i for i in range(len(df)) if i not in [1, 3]]
df2 = df.iloc[indices_to_keep]
print("------")
print(df2)
```

Output:

```
------
          Density  Young Modulus
Metal
Copper       8.96            110
Titanium     4.54            116
Nickel       8.90            200
------
          Density  Young Modulus
Metal
Aluminum     2.70             69
```

```
Iron         7.87            210
Nickel       8.90            200
```

Accessing cells and labels

```
print("------")
print(df.head(3))  # First 3 rows for reference
print("...")

# Accessing a specific cell using loc (row label and column label)
copper_density = df.loc['Copper', 'Density']
print(f"Density of copper: {copper_density}")

# Accessing a specific cell using iloc (row index and column index)
value = df.iloc[0, 1]      # access cell in first row, second column
row_label = df.index[0]    # label of first row
col_label = df.columns[1]  # label of second column
print(f"{col_label} of {row_label}: {value}")
```

Output:

```
------
          Density  Young Modulus
Metal
Aluminum     2.70             69
Copper       8.96            110
Iron         7.87            210
...
Density of copper: 8.96
Young Modulus of Aluminum: 69
```

```
print("------")
print(df.head(3))  # Print first 3 rows for reference
print("...")

# Make a copy of original df
df2 = df.copy()

# Change cells
df2.loc['Copper', 'Density'] = 8.94
df2.iloc[2, 1] = 209

print("------")
print(df2.head(3))  # Print new dataframe
print("...")
```

Output:

```
------
          Density  Young Modulus
Metal
Aluminum     2.70             69
Copper       8.96            110
Iron         7.87            210
...
------
          Density  Young Modulus
Metal
Aluminum     2.70             69
Copper       8.94            110
Iron         7.87            209
```

...

Merging two dataframes (or adding rows)

```
# Create a second DataFrame
new_data = {
    'Metal': ['Lead', 'Platinum'],
    'Density': [11.34, 21.45],
    'Young Modulus': [16, 168]
}
second_df = pd.DataFrame(new_data)
print("------")
print(second_df)

second_df.set_index('Metal', inplace=True)
print("------")
print(second_df)

# Merging the two DataFrames
merged_df = pd.concat([df, second_df], ignore_index=False)
print("------")
print(merged_df)

# Merging the two DataFrames overriding the index label
merged_df2 = pd.concat([df, second_df], ignore_index=True)
print("------")
print(merged_df2)
```

Output:

```
------
      Metal  Density  Young Modulus
0      Lead    11.34             16
1  Platinum    21.45            168
------
          Density  Young Modulus
Metal
Lead        11.34             16
Platinum    21.45            168
------
          Density  Young Modulus
Metal
Aluminum     2.70             69
Copper       8.96            110
Iron         7.87            210
Titanium     4.54            116
Nickel       8.90            200
Lead        11.34             16
Platinum    21.45            168
------
   Density  Young Modulus
0     2.70             69
1     8.96            110
2     7.87            210
3     4.54            116
4     8.90            200
5    11.34             16
6    21.45            168
```

9.2 Dataframe Methods

```
import pandas as pd

# Create DataFrame
data = {
    'Metal': [ 'Copper', 'Iron', 'Titanium', 'Nickel', 'Aluminum'],
    'Density': [8.96, 7.87, 4.54, 8.90, 2.70],
    'Young Modulus': [110, 210, 116, 200, 69]
}
df = pd.DataFrame(data)
df.set_index('Metal', inplace=True)
print("------")
print(df)
```

Output:

```
------
          Density  Young Modulus
Metal
Copper       8.96            110
Iron         7.87            210
Titanium     4.54            116
Nickel       8.90            200
Aluminum     2.70             69
```

Filtering

```
# Filtering metals with density greater than 5 g/cm³
filtered_df = df[df['Density'] > 5]
print("------")
print(filtered_df)
```

Output:

```
------
        Density  Young Modulus
Metal
Copper     8.96            110
Iron       7.87            210
Nickel     8.90            200
```

Sorting

```
# Sorting table by ascending (alphabetical) order of metals
sorted_df = df.sort_values(by='Metal')
print("------")
print(sorted_df)

# Sorting by Young's Modulus in descending order
sorted_df_desc = df.sort_values(by='Young Modulus', ascending=False)
print("------")
print(sorted_df_desc)
```

Output:

```
------
          Density  Young Modulus
Metal
Aluminum     2.70             69
Copper       8.96            110
Iron         7.87            210
```

```
Nickel       8.90            200
Titanium     4.54            116
------
          Density  Young Modulus
Metal
Iron         7.87            210
Nickel       8.90            200
Titanium     4.54            116
Copper       8.96            110
Aluminum     2.70             69
```

Basic statistics

```
# Calculating summary statistics
summary_statistics = df.describe()
print("------")
print(summary_statistics)
```

Output:

```
------
        Density  Young Modulus
count  5.000000       5.000000
mean   6.594000     141.000000
std    2.825151      61.261734
min    2.700000      69.000000
25%    4.540000     110.000000
50%    7.870000     116.000000
75%    8.900000     200.000000
max    8.960000     210.000000
```

Max, min, mean, std

```
# Finding the maximum Young's Modulus
max_modulus = df['Young Modulus'].max()
print(f"Maximum Young's Modulus: {max_modulus}")

# Finding the minimum Young's Modulus
min_modulus = df['Young Modulus'].min()
print(f"Minimum Young's Modulus: {min_modulus}")

# Calculating the mean density
mean_density = df['Density'].mean()
print(f"Mean Density: {mean_density}")
# Calculating the mean density
std_density = df['Density'].std()
print(f"STD of Density: {std_density}")
```

Output:

```
Maximum Young's Modulus: 210
Minimum Young's Modulus: 69
Mean Density: 6.594000000000001
STD of Density: 2.8251513233807497
```

Groupby (advanced) The groupby() method in pandas is a powerful tool for grouping data and performing aggregate operations on those groups.

```
df2 = df.copy()
# Assuming we have another column 'Category'
df2['Category'] = ['Light', 'Heavy', 'Heavy', 'Light', 'Heavy']
```

```
print("------")
print(df2)

# Grouping by 'Category' and calculating the mean of each numeric column
grouped_df = df2.groupby('Category')[['Density', 'Young Modulus']].mean()
print('------')
print(grouped_df)
```

Output:

```
------
          Density  Young Modulus Category
Metal
Copper       8.96            110    Light
Iron         7.87            210    Heavy
Titanium     4.54            116    Heavy
Nickel       8.90            200    Light
Aluminum     2.70             69    Heavy
------
           Density  Young Modulus
Category
Heavy     5.036667     131.666667
Light     8.930000     155.000000
```

9.3 Dataframes and NaN values

```
import pandas as pd
import numpy as np

# Creating a DataFrame with NaN values
data = {
    'Student': ['Alice', 'Bob', 'Charlie', 'David', 'Eve', 'Frank'],
    'Maths': [85, 78, np.nan, 92, 88, np.nan],
    'Physics': [np.nan, 80, 75, np.nan, 90, 85]
}
df = pd.DataFrame(data)
print("------")
print(df)
```

Output:

```
------
   Student  Maths  Physics
0    Alice   85.0      NaN
1      Bob   78.0     80.0
2  Charlie    NaN     75.0
3    David   92.0      NaN
4      Eve   88.0     90.0
5    Frank    NaN     85.0
```

9.3.1 Removing NaN values

```
# Removing rows with any NaN values
df_clean = df.dropna()
print('------')
print(df_clean)

# Reset index
df_clean = df_clean.reset_index(drop=True)
```

```
print('------')
print(df_clean)
```

Output:

```
------
  Student  Maths  Physics
1     Bob   78.0     80.0
4     Eve   88.0     90.0
------
  Student  Maths  Physics
0     Bob   78.0     80.0
1     Eve   88.0     90.0
```

```
# Filling NaN Maths values with 0 and Physics values with 50.0
df_filled = df.copy()
df_filled['Maths'] = df['Maths'].fillna(0)
df_filled['Physics'] = df['Physics'].fillna(50.0)
print("------")
print(df_filled)
```

Output:

```
------
   Student  Maths  Physics
0    Alice   85.0     50.0
1      Bob   78.0     80.0
2  Charlie    0.0     75.0
3    David   92.0     50.0
4      Eve   88.0     90.0
5    Frank    0.0     85.0
```

9.4 Data analysis

```
# Metals dataframe with Melting Point added
data = {
    'Metal': ['Copper', 'Iron', 'Titanium', 'Nickel', 'Aluminum', 'Gold', 'Silver',
              'Platinum', 'Lead', 'Zinc', 'Tin'],
    'Density': [8.96, 7.87, 4.54, 8.90, 2.70, 19.32, 10.49, 21.45, 11.34, 7.14, 7.31],
    'Young Modulus': [110, 210, 116, 200, 69, 79, 83, 168, 16, 108, 50],
    'Melting Point': [1085, 1538, 1668, 1455, 660, 1064, 961, 1768, 327, 419, 232]
}

df = pd.DataFrame(data)
df.set_index('Metal', inplace=True)
print('------')
print(df)
```

Output:

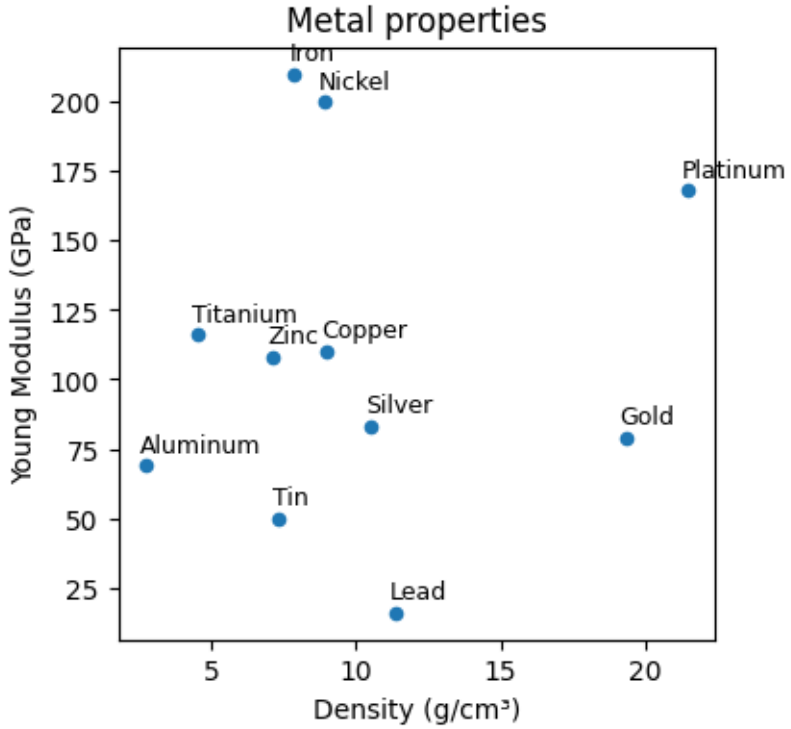
```
------
          Density  Young Modulus  Melting Point
Metal
Copper       8.96            110           1085
Iron         7.87            210           1538
Titanium     4.54            116           1668
Nickel       8.90            200           1455
Aluminum     2.70             69            660
Gold        19.32             79           1064
Silver      10.49             83            961
Platinum    21.45            168           1768
Lead        11.34             16            327
```

```
Zinc          7.14            108            419
Tin           7.31             50            232
```

9.4.1 Scatter plots

Scatter plot of Melting Point vs Density

```
# Plotting Melting Point vs Density
ax = df.plot(kind='scatter', x='Density', y='Young Modulus',
             title='Metal properties', figsize=(4, 4))
ax.set_xlabel('Density (g/cm³)')
ax.set_ylabel('Young Modulus (GPa)')

# Adding labels to the points with an offset
for row_index, row in df.iterrows():
    ax.text(row['Density'] - 0.2, row['Young Modulus'] + 5,
            row_index, fontsize=9)
```




```
<Figure size 400x400 with 1 Axes>
```

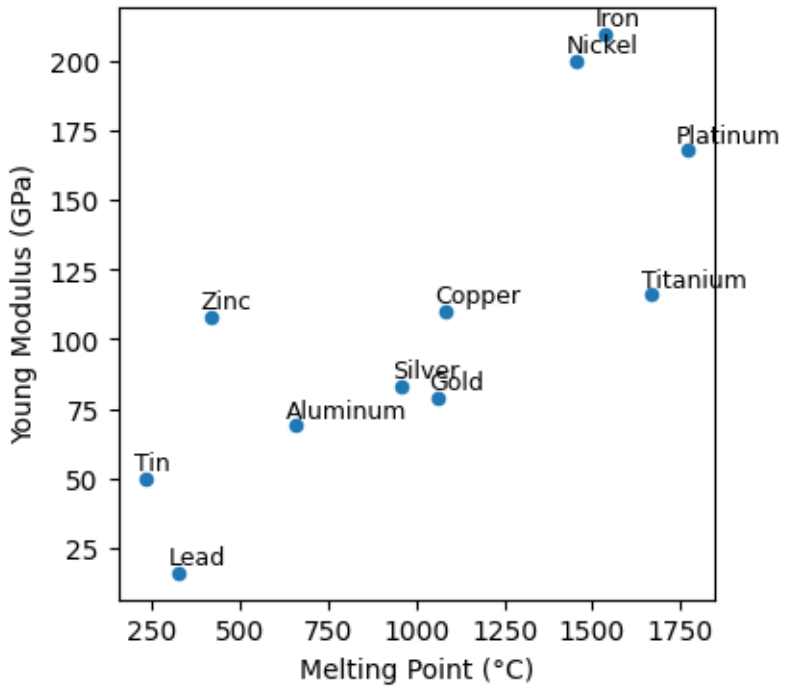
Task 1: Scatter plot of Young Modulus vs Melting point Produce a scatter plot of Young Modulus vs Melting Point for the metals in the dataframe. Label each point with the corresponding metal.

Answer

Solution

```
# Plotting Young Modulus vs Melting Point
ax = df.plot(kind='scatter', x='Melting Point', y='Young Modulus',
             figsize=(4, 4))
ax.set_ylabel('Young Modulus (GPa)')
ax.set_xlabel('Melting Point (°C)')

# Adding labels to the points with an offset
for row_index, row in df.iterrows():
```

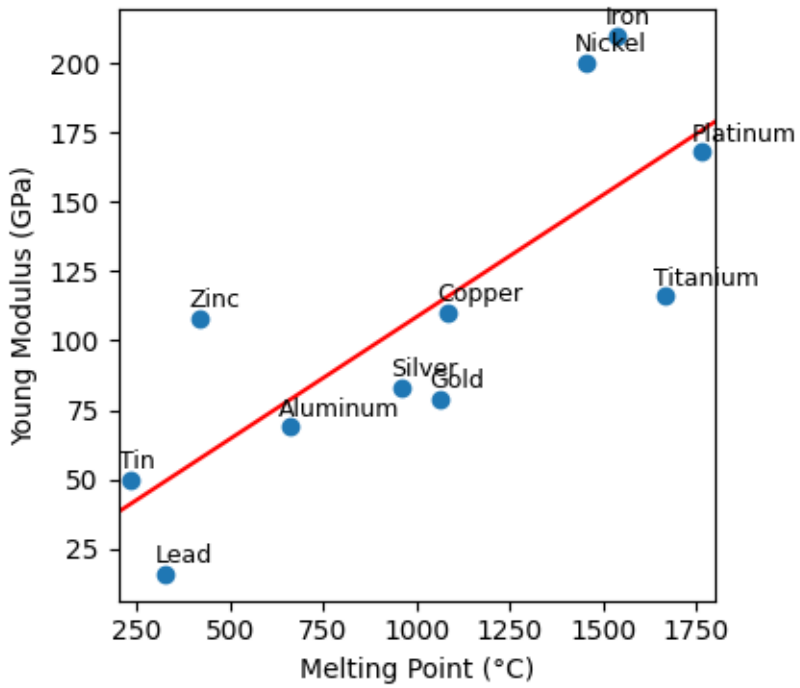
```
    ax.text(row['Melting Point'] - 30, row['Young Modulus'] + 3,
            row_index, fontsize=9)
```




```
<Figure size 400x400 with 1 Axes>
```

9.4.2 Linear regression

```
import numpy as np
import matplotlib.pyplot as plt

# Get x and y data
x = df['Melting Point']
y = df['Young Modulus']

# Perform linear regression (polynomial fit of order 1)
m, b = np.polyfit(x, y, 1)  # m is slope, b is intercept
x_fit = np.linspace(100, 2000, 10)
y_fit = m*x_fit + b

# Plotting Melting Point vs Young Modulus using plt.scatter
plt.figure(figsize=(4,4))
plt.scatter(x, y)
plt.xlabel('Melting Point (°C)')
plt.ylabel('Young Modulus (GPa)')
# Add line of best fit
plt.plot(x_fit, y_fit, color='red')

# Adjust limits for clarity
plt.xlim([200, 1800])

# Adding labels to the points with an offset
for row_index, row in df.iterrows():
    plt.text(row['Melting Point'] - 30, row['Young Modulus'] + 4, row_index, fontsize=9)

plt.show()
```



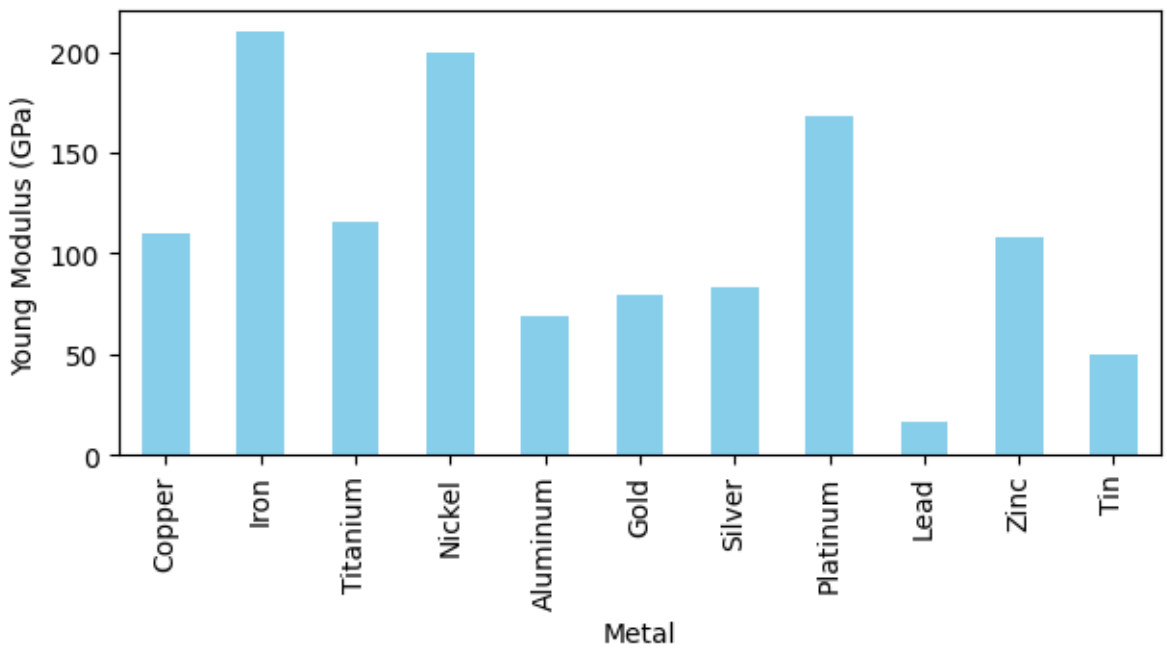

```
<Figure size 400x400 with 1 Axes>
```

9.4.3 Bar charts

The code below produces a bar chart for Young's Modulus of various metals using the `df.plot` method

```
# Plotting the bar chart for Young's Modulus
ax1 = df['Young Modulus'].plot(kind='bar',
        figsize=(7, 3), color='skyblue')

# Setting labels and title
ax1.set_xlabel('Metal')
ax1.set_ylabel('Young Modulus (GPa)')
```

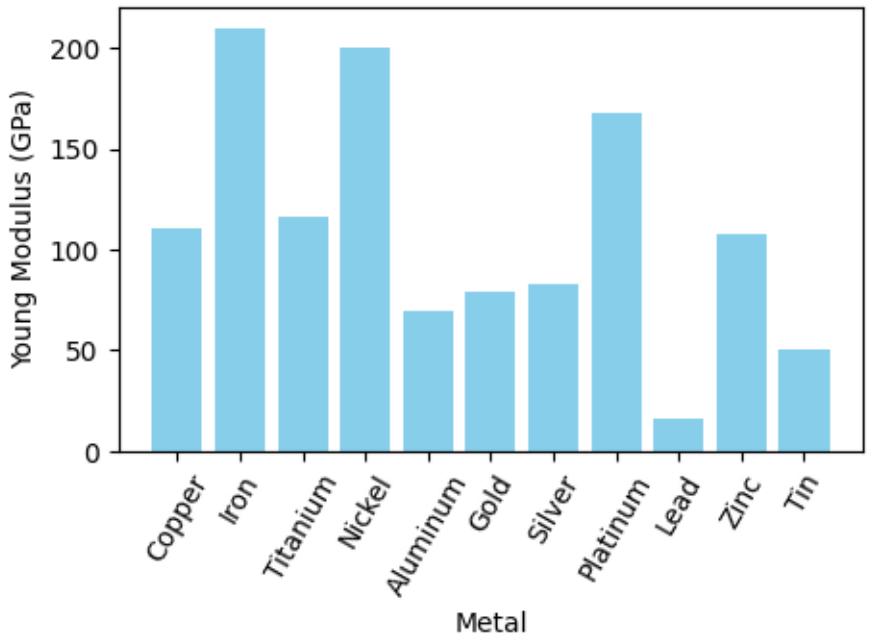
```
Text(0, 0.5, 'Young Modulus (GPa)')
```




```
<Figure size 700x300 with 1 Axes>
```

Now, using `matplotlib` and `plt.bar`:

```
# Plotting the bar chart for Young's Modulus using plt.bar
plt.figure(figsize=(5,3))
plt.bar(df.index, df['Young Modulus'], color='skyblue')

# Setting labels and title
plt.xlabel('Metal')
plt.ylabel('Young Modulus (GPa)')

# Rotating x-axis labels for better readability
plt.xticks(rotation=60)

# Display the plot
plt.show()
```




```
<Figure size 500x300 with 1 Axes>
```

9.5 Exercise 9.1 Cumulative grade distributions.

Upload the file 'IAL_June_2022.csv' into the working directory. The CSV file can be easily imported as a dataframe as follows:

```
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np

# Reading a CSV file into a DataFrame
df = pd.read_csv('IAL_June_2022.csv')
print("IAL_June_2022.csv:")
print(df)
```

Output:

```
IAL_June_2022.csv:
                   Subject    Sat    A*     A     B      C      D      E  \
0                   ARABIC   1597   657  1237  1491   1557   1576   1586
1               ACCOUNTING   1489   244   678   969   1181   1316   1406
2                  BIOLOGY   3758  1013  1960  2607   3091   3432   3624
3                 BUSINESS   2238   209   761  1327   1761   2039   2164
4                CHEMISTRY   4905   982  2422  3322   3965   4438   4733
5                ECONOMICS   2674   424  1396  2042   2354   2537   2615
6         ENGLISH LANGUAGE    269    22    71   133    193    233    253
7       ENGLISH LITERATURE    417    59   153   283    356    391    414
8      FURTHER MATHEMATICS   2705  1274  1971  2321   2506   2604   2664
9                   FRENCH    423    36   209   299    374    404    418
10                GEOGRAPHY    325    29   139   217    278    301    316
11                    GREEK    487   106   331   436    470    474    484
12                   GERMAN    245    43   171   208    223    231    243
13                  HISTORY    487    89   213   311    393    443    467
14  INFORMATION TECHNOLOGY    453    52   134   234    316    378    411
15                      LAW    336    58   118   146    178    217    254
16              MATHEMATICS  13402  3702  7544  9954  11513  12456  13006
17                  PHYSICS   4732   838  2307  3069   3604   4066   4421
```

```
18        PURE MATHEMATICS    235    60   140   177    198    218    226
19              PSYCHOLOGY    540   105   257   373    457    501    527
20                 SPANISH    932   121   645   843    905    921    928

        U   A*%    A%    B%    C%    D%    E%     U%
0    1597  41.1  77.5  93.4  97.5  98.7  99.3  100.0
1    1489  16.4  45.5  65.1  79.3  88.4  94.4  100.0
2    3758  27.0  52.2  69.4  82.3  91.3  96.4  100.0
3    2238   9.3  34.0  59.3  78.7  91.1  96.7  100.0
4    4905  20.0  49.4  67.7  80.8  90.5  96.5  100.0
5    2674  15.9  52.2  76.4  88.0  94.9  97.8  100.0
6     269   8.2  26.4  49.4  71.7  86.6  94.1  100.0
7     417  14.1  36.7  67.9  85.4  93.8  99.3  100.0
8    2705  47.1  72.9  85.8  92.6  96.3  98.5  100.0
9     423   8.5  49.4  70.7  88.4  95.5  98.8  100.0
10    325   8.9  42.8  66.8  85.5  92.6  97.2  100.0
11    487  21.8  68.0  89.5  96.5  97.3  99.4  100.0
12    245  17.6  69.8  84.9  91.0  94.3  99.2  100.0
13    487  18.3  43.7  63.9  80.7  91.0  95.9  100.0
14    453  11.5  29.6  51.7  69.8  83.4  90.7  100.0
15    336  17.3  35.1  43.5  53.0  64.6  75.6  100.0
16  13402  27.6  56.3  74.3  85.9  92.9  97.0  100.0
17   4732  17.7  48.8  64.9  76.2  85.9  93.4  100.0
18    235  25.5  59.6  75.3  84.3  92.8  96.2  100.0
19    540  19.4  47.6  69.1  84.6  92.8  97.6  100.0
20    932  13.0  69.2  90.5  97.1  98.8  99.6  100.0
```

9.5.1 Task 1

Create a new dataframe, `df22`, that only has columns:

A*% A% B% C% D% E% U%

and where the column "Subject" is set as the (row) index. Lastly, rename the columns of `df22` (remove the %) as

A* A B C D E U

The latter can be done through the command:

```
df22.columns = [col.replace("%", "") for col in df22.columns]
```

Solution

```
df22 = df.copy()
df22.set_index("Subject", inplace=True)
df22 = df22.drop(df22.columns[0:8], axis=1)
df22.columns = [col.replace("%", "") for col in df22.columns]
print("df22:")
print(df22)
```

Output:

```
df22:
                          A*     A     B     C     D     E      U
Subject
ARABIC                  41.1  77.5  93.4  97.5  98.7  99.3  100.0
ACCOUNTING              16.4  45.5  65.1  79.3  88.4  94.4  100.0
BIOLOGY                 27.0  52.2  69.4  82.3  91.3  96.4  100.0
BUSINESS                 9.3  34.0  59.3  78.7  91.1  96.7  100.0
CHEMISTRY               20.0  49.4  67.7  80.8  90.5  96.5  100.0
ECONOMICS               15.9  52.2  76.4  88.0  94.9  97.8  100.0
ENGLISH LANGUAGE         8.2  26.4  49.4  71.7  86.6  94.1  100.0
```

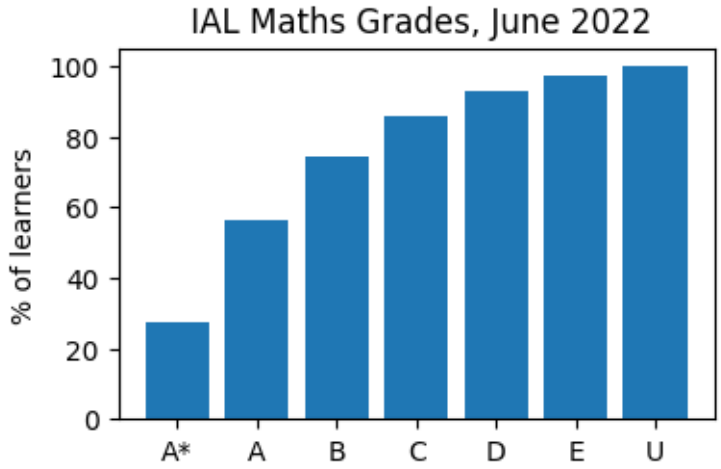
```
ENGLISH LITERATURE      14.1  36.7  67.9  85.4  93.8  99.3  100.0
FURTHER MATHEMATICS     47.1  72.9  85.8  92.6  96.3  98.5  100.0
FRENCH                   8.5  49.4  70.7  88.4  95.5  98.8  100.0
GEOGRAPHY                8.9  42.8  66.8  85.5  92.6  97.2  100.0
GREEK                   21.8  68.0  89.5  96.5  97.3  99.4  100.0
GERMAN                  17.6  69.8  84.9  91.0  94.3  99.2  100.0
HISTORY                 18.3  43.7  63.9  80.7  91.0  95.9  100.0
INFORMATION TECHNOLOGY  11.5  29.6  51.7  69.8  83.4  90.7  100.0
LAW                     17.3  35.1  43.5  53.0  64.6  75.6  100.0
MATHEMATICS             27.6  56.3  74.3  85.9  92.9  97.0  100.0
PHYSICS                 17.7  48.8  64.9  76.2  85.9  93.4  100.0
PURE MATHEMATICS        25.5  59.6  75.3  84.3  92.8  96.2  100.0
PSYCHOLOGY              19.4  47.6  69.1  84.6  92.8  97.6  100.0
SPANISH                 13.0  69.2  90.5  97.1  98.8  99.6  100.0
```

9.5.2 Task 2

Using the dataframe `df22` from Exercise 1, plot a bar chart showing showing the cumulative % of learners (bar height) achieving the A* A, B, C, D, E and U for the subject of MATHEMATICS. You may want to extract the relevant row as a Series.

Solution

```
df22 = df.copy()
df22.set_index("Subject", inplace=True)
df22 = df22.drop(df22.columns[0:8], axis=1)
df22.columns = [col.replace("%", "") for col in df22.columns]

maths = df22.loc["MATHEMATICS"]
grades = df22.columns

# Plot bar chart
plt.figure(figsize=(4,2.5))
plt.bar(grades, maths.values, align='center')
plt.ylabel('% of learners')
plt.title(' IAL Maths Grades, June 2022')
plt.show()
```




```
<Figure size 400x250 with 1 Axes>
```

9.5.3 Task 3

Using the dataframe `df22` from Exercise 1, generate a stacked barchart that plots the cumulative frequency distribution of grades (A* to U) for four subjects of your choosing. Note that each bar should represent a specific subject. The frequencies should be read from the dataframe; do not type them in manually.

Challenge: include all subjects.

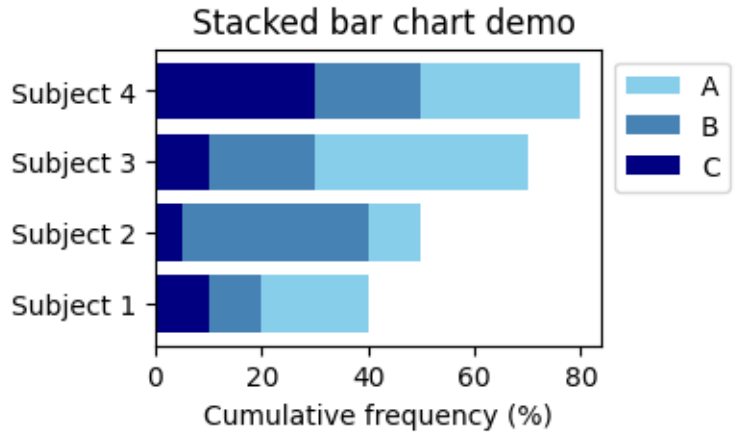
It may be usfeul to percentage of learners that got surpassed given grade as a numpy array. Eg,

```
percentages = np.array(df22["B%"])
```

Each element in the array`percentages` corresponds to a particular subject.

The code below shows how to produce a stacked bar chart. Use it as reference.

```
import matplotlib.pyplot as plt

# Dummy data
subjects = ['Subject 1', 'Subject 2', 'Subject 3', 'Subject 4']
grades = ["A", "B", "C"]
a_grade = np.array([10, 5, 10, 30])
b_grade = np.array([20, 40, 30, 50])
c_grade = np.array([40, 50, 70, 80])

# plot bars in stack manner
plt.figure(figsize=(3,2))
plt.barh(subjects, c_grade, color="skyblue")
plt.barh(subjects, b_grade, color = "steelblue")
plt.barh(subjects, a_grade, color="navy")

plt.xlabel('Cumulative frequency (%)')
plt.title(" Stacked bar chart demo")
plt.legend(grades, bbox_to_anchor=(1, 1))
plt.show()
```

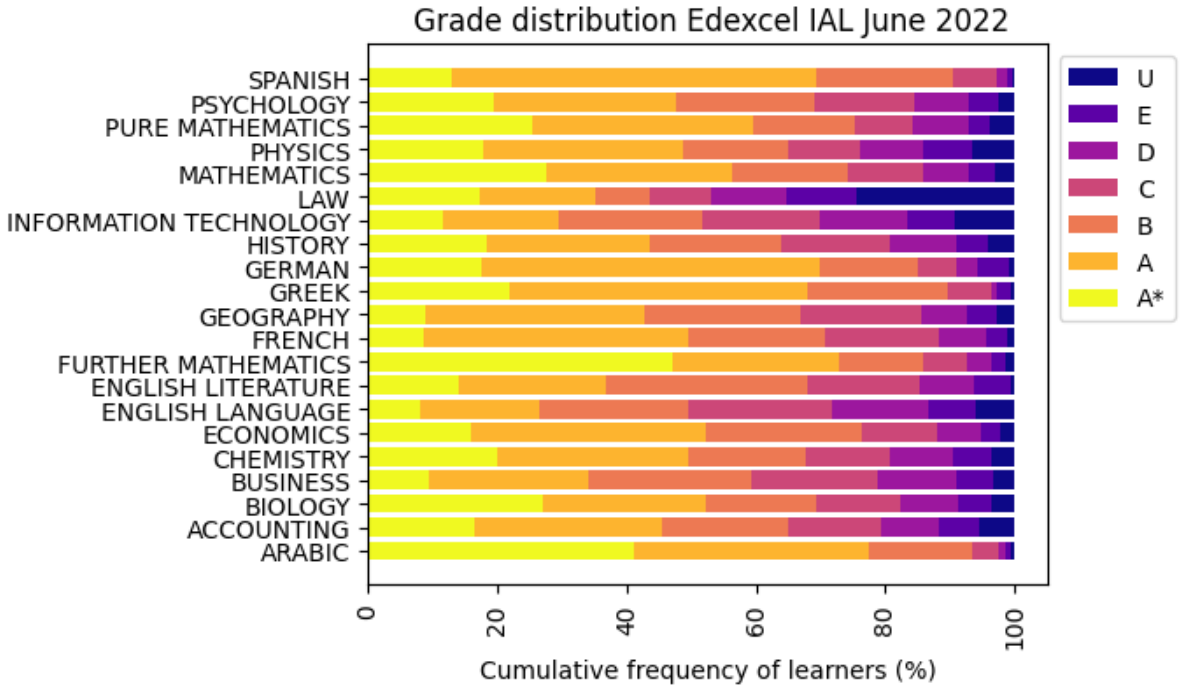



```
<Figure size 300x200 with 1 Axes>
```

Solution

```
import matplotlib.pyplot as plt
import numpy as np

df22 = df.copy()
df22.set_index("Subject", inplace=True)
df22 = df22.drop(df22.columns[0:8], axis=1)
df22.columns = [col.replace("%", "") for col in df22.columns]
```

```
# List of numpy arrays
percentages_by_grade = []
for column in df22.columns:
  percentages = np.array(df22[column])
  percentages_by_grade.append(percentages)
percentages_by_grade.reverse()

# List of grade labels
grades = list(df22.columns)
grades.reverse()

# List of subjects
subjects = list(df22.index)

# Using 'plasma' colormap to generate a list of colors
cmap = plt.get_cmap('plasma')
colors = cmap(np.linspace(0, 1, 7))
# Plotting bars in stack manner
plt.figure(figsize=(5,4))
for i, percentages in enumerate(percentages_by_grade):
  if i==0:
    bottom = 0
  else:
    bottom = sum(percentages_by_grade[0:i])
  plt.barh(subjects, percentages, color=colors[i])
plt.xlabel('Cumulative frequency of learners (%)')
plt.title(" Grade distribution Edexcel IAL June 2022")
plt.legend(grades, bbox_to_anchor=(1, 1))
plt.xticks(rotation=90)
plt.show()
```




```
<Figure size 500x400 with 1 Axes>
```

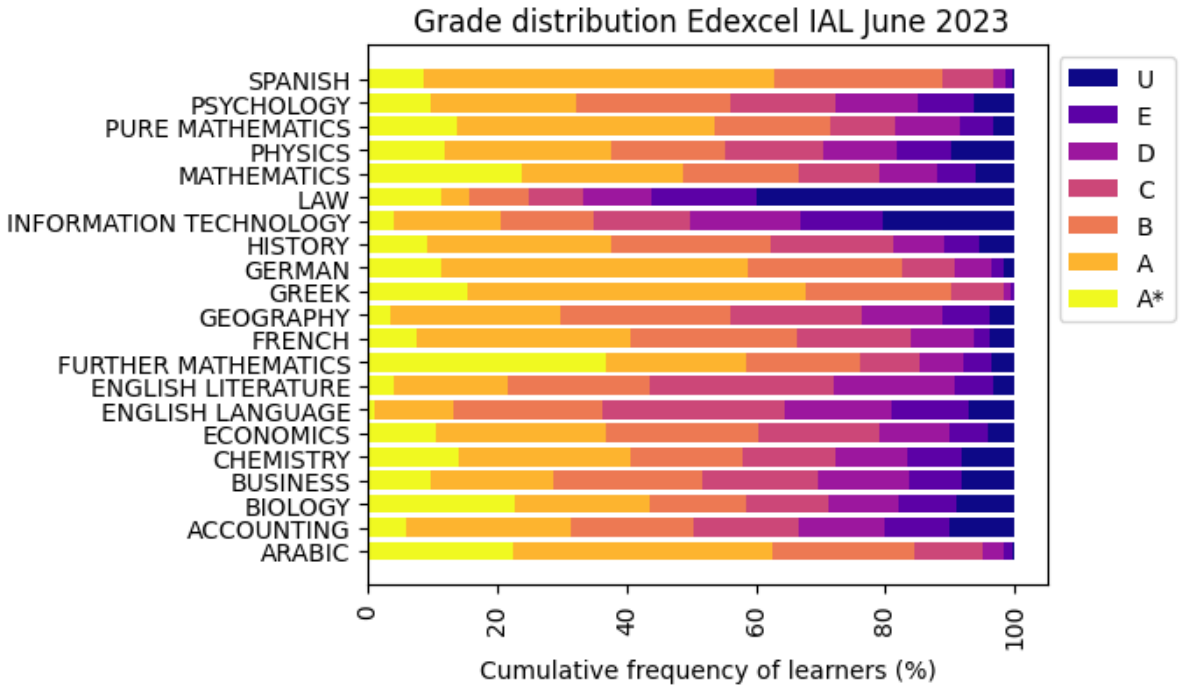
Below is the full analysis for the 2023 statistics in 'IAL_June_2023.csv'

```
import matplotlib.pyplot as plt
```

```
import numpy as np
import pandas as pd

# Reading the 2023 CSV file into a DataFrame
df23 = pd.read_csv('IAL_June_2023.csv')
df23.set_index("Subject", inplace=True)
df23 = df23.drop(df23.columns[0:8], axis=1)
df23.columns = [col.replace("%", "") for col in df23.columns]
print('------')
print(df23.head())

# List of numpy arrays
percentages_by_grade = []
for column in df23.columns:
  percentages = np.array(df23[column])
  percentages_by_grade.append(percentages)
percentages_by_grade.reverse()

# List of grade labels
grades = list(df23.columns)
grades.reverse()

# List of subjects
subjects = list(df23.index)

# List of colors (colormap)
cmap = plt.get_cmap('plasma')
colors = cmap(np.linspace(0, 1, 7))

# Plotting bars in stack manner
plt.figure(figsize=(5,4))
for i, percentages in enumerate(percentages_by_grade):
  if i==0:
    bottom = 0
  else:
    bottom = sum(percentages_by_grade[0:i])
  plt.barh(subjects, percentages, color=colors[i])
plt.xlabel('Cumulative frequency of learners (%)')
plt.title(" Grade distribution Edexcel IAL June 2023")
plt.legend(grades, bbox_to_anchor=(1, 1))
plt.xticks(rotation=90)
plt.show()
```

Output:

```
------
              A*     A     B     C     D     E      U
Subject
ARABIC      22.4  62.5  84.6  95.0  98.2  99.7  100.0
ACCOUNTING   6.0  31.5  50.4  66.6  79.9  89.8  100.0
BIOLOGY     22.6  43.6  58.5  71.2  82.0  90.9  100.0
BUSINESS     9.6  28.6  51.6  69.7  83.8  91.9  100.0
CHEMISTRY   14.1  40.6  57.9  72.2  83.4  91.7  100.0
```




```
<Figure size 500x400 with 1 Axes>
```

9.6 Exercise 9.2 Evolution of the grade distributions

You have two datasets, 'IAL_June_2022.csv' and 'IAL_June_2023.csv', containing the grade distribution of learners in years 2022 and 2023, respectively, for the same set of subjects. Your task is to create a scatter plot to visualize the relationship between the percentages of learners achieving grade 'A' or higher in 2022 versus 2023.

Scatter Plot: Plot the percentage of learners achieving at least a grade 'A' in 2022 (x-axis) against the percentage of learners achieving at least a grade 'A' in 2023 (y-axis). Each data point in the scatter plot represents a specific subject. Make sure to include:

1. **Line of Best Fit**: Add a line of best fit to the scatter plot to show the trend.
2. **Center of Mass**: Highlight the center of mass (mean x, mean y) with a prominent marker.
3. **No-change Reference Line**: Include a dashed diagonal line y = x as reference.
4. **Labels**: Add labels for each point, displaying the first two characters of the subject name.
5. **Plot Adjustments**: Ensure that equal plot limits are set for both axes (e.g. 10 to 80) and enable grid lines for better readability.

Extra: Reuse your code to produce a second scatter plot that shows the relationship between the percentages of learners achieving grade 'B' or higher in 2022 versus 2023.

Solution

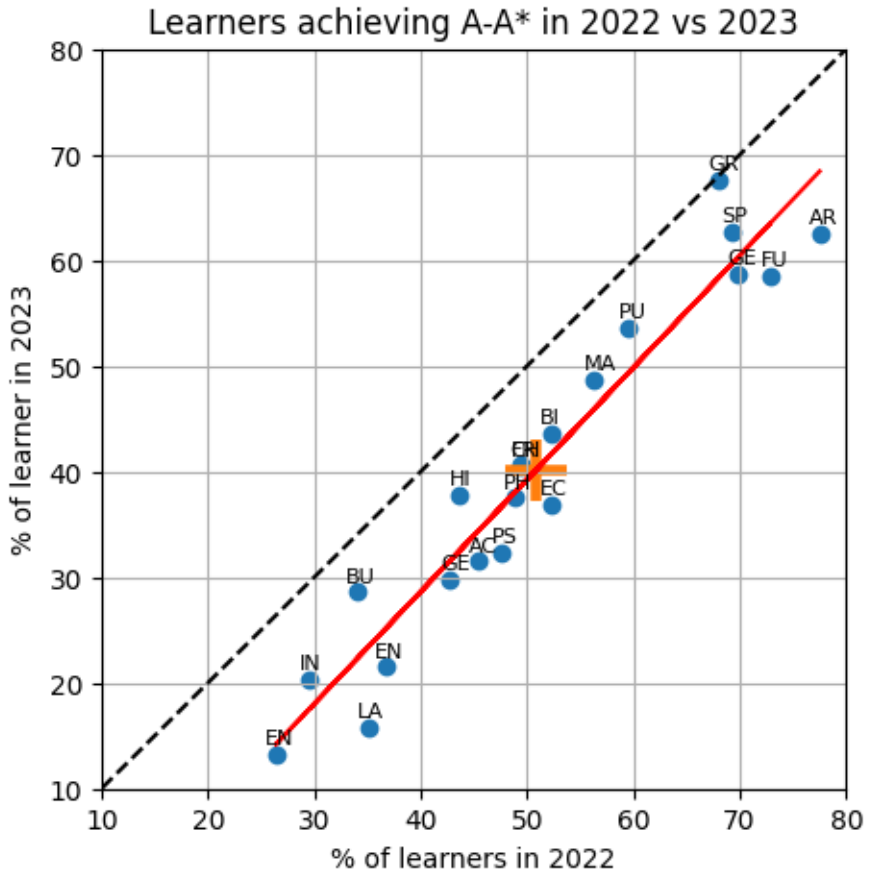
```
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np

# Reading the 2022 CSV file into a DataFrame
df22 = pd.read_csv('IAL_June_2022.csv')
df22.set_index("Subject", inplace=True)
df22 = df22.drop(df22.columns[0:8], axis=1)
df22.columns = [col.replace("%", "") for col in df22.columns]
print('------')
print(df22.head())

# Reading the 2023 CSV file into a DataFrame
```

```
df23 = pd.read_csv('IAL_June_2023.csv')
df23.set_index("Subject", inplace=True)
df23 = df23.drop(df23.columns[0:8], axis=1)
df23.columns = [col.replace("%", "") for col in df23.columns]
print('------')
print(df23.head())
```

Output:

```
------
              A*     A     B     C     D     E      U
Subject
ARABIC      41.1  77.5  93.4  97.5  98.7  99.3  100.0
ACCOUNTING  16.4  45.5  65.1  79.3  88.4  94.4  100.0
BIOLOGY     27.0  52.2  69.4  82.3  91.3  96.4  100.0
BUSINESS     9.3  34.0  59.3  78.7  91.1  96.7  100.0
CHEMISTRY   20.0  49.4  67.7  80.8  90.5  96.5  100.0
------
              A*     A     B     C     D     E      U
Subject
ARABIC      22.4  62.5  84.6  95.0  98.2  99.7  100.0
ACCOUNTING   6.0  31.5  50.4  66.6  79.9  89.8  100.0
BIOLOGY     22.6  43.6  58.5  71.2  82.0  90.9  100.0
BUSINESS     9.6  28.6  51.6  69.7  83.8  91.9  100.0
CHEMISTRY   14.1  40.6  57.9  72.2  83.4  91.7  100.0
```

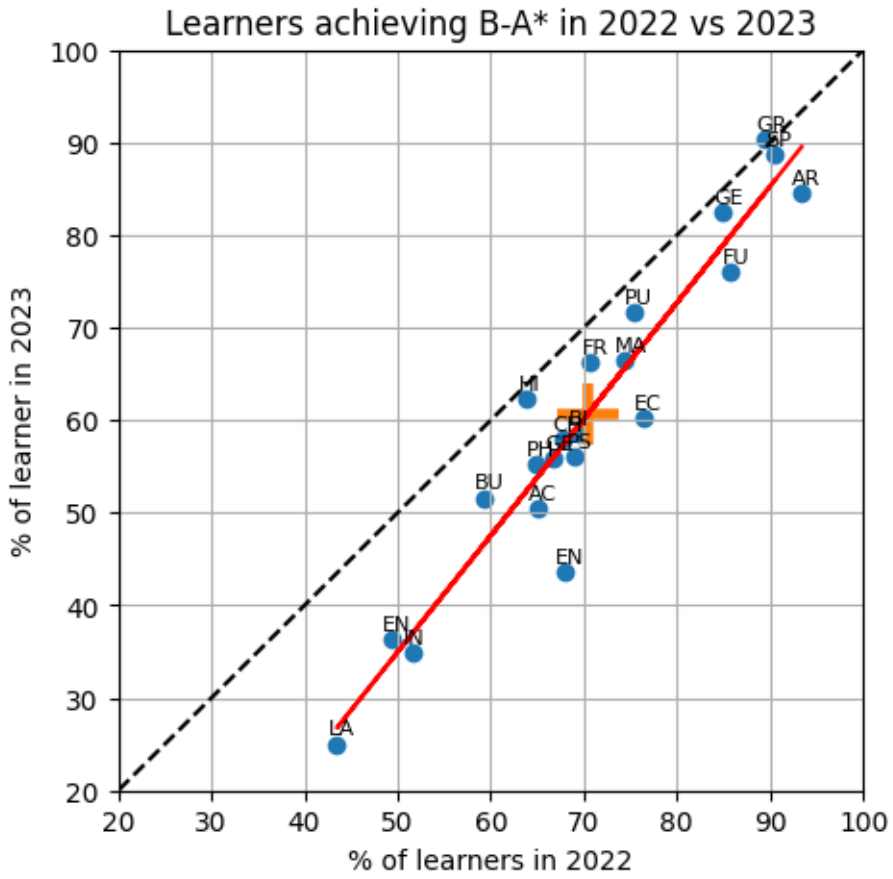
```
import numpy as np
# Define x and y data
grade = "A"
x = df22[grade]
y = df23[grade]

# Plotting using plt.scatter
plt.figure(figsize=(5, 5))
plt.scatter(x, y)
plt.xlabel('% of learners in 2022')
plt.ylabel('% of learner in 2023')
plt.title('Learners achieving A-A* in 2022 vs 2023')

# Add line of best fit
m, c = np.polyfit(x, y, 1)
y_fit = m*x+c
plt.plot(x, y_fit, color='red')

# Add center of mass (mean x, mean y)
plt.scatter(np.mean(x), np.mean(y), marker='+', s=500,linewidths=4)

# Adjust limits for clarity
plt.xlim([10, 80])
plt.ylim([10, 80])
plt.grid("on")
plt.plot([0, 100], [0, 100], '--', color="k")
# Adding labels to the points with an offset
for subject in df22.index:
  plt.text(df22.loc[subject,grade]-1, df23.loc[subject,grade]+1,
           subject[:2], fontsize=8)

plt.show()
```




```
<Figure size 500x500 with 1 Axes>
```

```
import numpy as np
# Define x and y data
grade = "B"
x = df22[grade]
y = df23[grade]

# Plotting using plt.scatter
plt.figure(figsize=(5, 5))
plt.scatter(x, y)
plt.xlabel('% of learners in 2022')
plt.ylabel('% of learner in 2023')
plt.title('Learners achieving B-A* in 2022 vs 2023')

# Add line of best fit
m, c = np.polyfit(x, y, 1)
y_fit = m*x+c
plt.plot(x, y_fit, color='red')

# Add center of mass (mean x, mean y)
plt.scatter(np.mean(x), np.mean(y), marker='+', s=500,linewidths=4)

# Adjust limits for clarity
plt.xlim([20, 100])
plt.ylim([20, 100])
plt.grid("on")

plt.plot([0, 100], [0, 100], '--', color="k")
# Adding labels to the points with an offset
```

```
for subject in df22.index:
  plt.text(df22.loc[subject,grade]-1, df23.loc[subject,grade]+1,
           subject[:2], fontsize=8)

plt.show()
```




```
<Figure size 500x500 with 1 Axes>
```