Python Course Notes

International Programming School Algorithmics

Course Instructor:

Associate Engineer, Andrii Voznesenskyi

Mathematics and Information Systems Faculty
Warsaw University of Technologies

May 20, 2023

Restrictions:

This document is intended for use only by individuals associated with Algorithmics and the author. Unauthorized distribution or sharing of this document, in whole or in part, is strictly prohibited.

Copyright Information:

Purpose:

This document is intended for the exclusive use of colleagues at Algorithmics International School or students seeking to enhance their individual understanding of the course and engage in self-education. It is of utmost importance to strictly adhere to the restrictions and copyright information provided within this document.

"Education is the kindling of a flame, not the filling of a vessel."
Socrates, Ancient Greek philosopher

Contents

1 Introduction to Python

Python is a high-level, interpreted dynamically-typed programming language. It was created by Guido van Rossum and first released in 1991. Python is designed to be highly readable, with a simple and consistent syntax that promotes readability and therefore reduces the cost of program maintenance.

This particular section will be modified a bit late with accordance to author's direct intention...

2 Installation

To install Python, you can download the latest version from the official Python website. Python can run on a variety of platforms including Windows, Mac, and various distributions of Linux. There are also several distributions of Python for specific purposes like Anaconda, which is focused on data analysis and scientific computing.

This particular section will be modified a bit late with accordance to author's direct intention...

3 Your First Python Program1111

4 Your First Python Program

Here's how you can write your first Python program. This simple program outputs the string "Hello, World!" to the console:

Example 4.1.

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

4.1 The print() function

The print() function is a built-in function in Python that is used to output or display information to the console or terminal. It takes one or more arguments (values or expressions) and displays them as text.

In the example above, the print() function is used to display the string "Hello, World!" to the console. The string is enclosed in double quotation marks, indicating that it is a string literal.

You can also use single quotation marks to enclose strings. Both double and single quotation marks are valid ways to define strings in Python.

4.2 Using f-strings with the print() function

In addition to printing simple strings, you can use f-strings (formatted string literals) with the print() function to dynamically format and display values within a string.

To use an f-string, you prefix the string with the letter f or F and enclose the expression you want to evaluate within curly braces .

Here's an example that demonstrates the usage of an f-string with the print() function:

Example 4.2.

```
1 name = "Alice"
2 age = 25
3 print(f"My name is {name} and I am {age} years old.")
```

When this program is executed, it will output the following text to the console:

```
My name is Alice and I am 25 years old.
```

In the f-string, the expressions name and age are evaluated and their values are inserted into the string at the respective positions.

F-strings are a powerful feature in Python that allow you to easily format strings with variables or expressions. They provide a concise and readable way to combine text and values in output statements.

4.3 Using f-strings with the `print()` function

In addition to printing simple strings, you can use f-strings (formatted string literals) with the `print()` function to dynamically format and display values within a string.

To use an f-string, you prefix the string with the letter `f` or `F` and enclose the expression you want to evaluate within curly braces .

Here's an example that demonstrates the usage of an f-string with the `print()` function:

Example 4.3.

```
1 name = "Alice"
2 age = 25
3 print(f"My name is {name} and I am {age} years old.")
```

When this program is executed, it will output the following text to the console:

```
My name is Alice and I am 25 years old.
```

In the f-string, the expressions `name` and `age` are evaluated and their values are inserted into the string at the respective positions.

F-strings are a powerful feature in Python that allow you to easily format strings with variables or expressions. They provide a concise and readable way to combine text and values in output statements.

4.4 Using the print() function's properties

The print() function in Python has some useful properties that can be used to modify its behavior. Here are a few commonly used properties:

- **sep**: This property allows you to specify the separator between multiple arguments passed to the print() function. By default, the separator is a space character. You can change it by assigning a different value to print()'s sep property.

- **end**: This property allows you to specify the string that should be printed at the end of the print() function's output. By default, the end property is set to a newline character (), which causes the next output to appear on a new line. You can change it by assigning a different value to print()'s end property.
- **file**: This property allows you to redirect the output of the print() function to a file or a different output stream instead of the default standard output (console). By default, the file property is set to sys.stdout, which represents the standard output. You can change it by assigning a different file object or output stream to print()'s file property.

Here's an example that demonstrates the usage of these properties:

Example 4.4.

```
name = "Alice"
age = 25
output_file = open("output.txt", "w")

print(f"My name is {name}", end=", ")
print(f"and I am {age} years old.", file=output_file, sep="|")

output_file.close()
```

In this example, we have set the end property of the first print() statement to ", " to print a comma followed by a space instead of the default newline character. We have also set the file property of the second print() statement to output_file, which is a file object representing a file named "output.txt". Additionally, we have set the sep property of the second print() statement to "|" to separate the two arguments with a pipe character.

When this program is executed, it will print the following text to the console:

```
My name is Alice, and I am 25 years old.
```

It will also create a file named "output.txt" with the following content:

```
My name is Alice|and I am 25 years old.
```

These properties provide flexibility in controlling the output format and destination when using the print() function in Python.

4.5 Using the print().format() function in Python

The print().format() function in Python is used to format strings. Here are some examples:

1. Basic usage:

```
    print("Hello, {}. You are {} years old.".format("Alice", 25))
```

This will output: "Hello, Alice. You are 25 years old."

2. Positional arguments:

```
print("{1}, {0}".format("world", "Hello"))
```

This will output: "Hello, world". The numbers inside the curly braces represent the index of the arguments in the format method.

3. Named arguments:

```
print("{greeting}, {name}".format(greeting="Hello",
    name="world"))
```

This will output: "Hello, world". The strings inside the curly braces represent the names of the arguments in the format method.

5 Variables and Types

In Python, variables are created when you assign a value to them. Python is dynamically typed, which means that you don't need to specify the type of a variable when you declare it. Here's an example of variable assignment in Python:

Example 5.1.

```
x = 5
y = "Hello, World!"
```

In this example, x is an integer and y is a string.

Variables in Python can store values of different types, such as integers, floats, strings, booleans, lists, tuples, and dictionaries. The type of a variable can change dynamically based on the value assigned to it. Python provides built-in functions to determine the type of a variable, such as the `type()` function.

For example, consider the following code snippet:

Example 5.2.

```
x = 5
print(type(x)) # Output: <class 'int'>

x = "Hello, World!"
print(type(x)) # Output: <class 'str'>
```

In this code, the `type()` function is used to determine the type of the variable `x` before and after reassignment. The output shows that `x` changes from an integer (`int`) to a string (`str`).

Python also supports type hints, which allow you to specify the expected type of a variable. Although these hints are not enforced by the Python interpreter, they can be useful for documentation and code readability. Type hints can be added using the colon (`:`) followed by the type after the variable name.

Example 5.3. *Here's an example:*

```
x: int = 5
y: str = "Hello, World!"
```

In this code, the type hints indicate that `x` should be an integer and `y` should be a string. While these hints are optional, they can help make your code more understandable and catch potential type-related errors early on.

Python's flexibility in handling different variable types and its support for dynamic typing make it a versatile language for various programming tasks.

Python supports several data types, including:

5.1 Numeric Types

Numeric Types:

- **int** - integers, such as 1, 2, -5, etc.

  Example 5.4.

```
x = 5
print(x) # Output: 5
```

- **float** - floating-point numbers, such as 3.14, -2.5, etc.

  Example 5.5.

```
y = 3.14
print(y) # Output: 3.14
```

- **complex** - complex numbers, represented as *real + imaginary*, e.g., 2 + 3j.

  Example 5.6.

```
z = 2 + 3j
print(z) # Output: (2+3j)
```

5.2 Sequence Types

Sequence Types:

- **str** - strings, a sequence of characters enclosed in single or double quotes, e.g., "Hello", 'World'.

  Example 5.7.

```
s = "Hello, World!"
print(s) # Output: Hello, World!
```

- **list** - ordered, mutable sequences of objects, enclosed in square brackets, e.g., [1, 2, 3].

Example 5.8.

```
lst = [1, 2, 3]
print(lst) # Output: [1, 2, 3]
```

- **tuple** - ordered, immutable sequences of objects, enclosed in parentheses, e.g., (4, 5, 6).

Example 5.9.

```
tup = (4, 5, 6)
print(tup) # Output: (4, 5, 6)
```

5.3 Mapping Type

Mapping Type:

- **dict** - dictionaries, unordered collections of key-value pairs, enclosed in curly braces, e.g., 'name': 'John', 'age': 25.

Example 5.10.

```
person = {'name': 'John', 'age': 25}
print(person) # Output: {'name': 'John', 'age': 25}
```

5.4 Set Types

Set Types:

- **set** - unordered collections of unique elements, enclosed in curly braces, e.g., 1, 2, 3.

Example 5.11.

```
s = {1, 2, 3}
print(s) # Output: {1, 2, 3}
```

- **frozenset** - immutable sets, enclosed in parentheses, e.g., frozenset(4, 5, 6).

Example 5.12.

```
fs = frozenset({4, 5, 6})
print(fs) # Output: frozenset({4, 5, 6})
```

5.5 Boolean Type

Boolean Type:

- **bool** - boolean values, either *True* or *False*.

Example 5.13.

```
a = True
b = False
print(a) # Output: True
print(b) # Output: False
```

5.6 Binary Types

Binary Types:

- **bytes** - sequences of integers in the range 0-255, enclosed in parentheses with a *b* prefix, e.g., b'Hello'.
  Example 5.14.

```
1 bt = b'Hello'
2 print(bt) # Output: b'Hello'
```

- **bytearray** - mutable sequences of integers in the range 0-255, enclosed in parentheses with a *bytearray* prefix, e.g., bytearray(b'World').
  Example 5.15.

```
1 ba = bytearray(b'World')
2 print(ba) # Output: bytearray(b'World')
```

Each data type in Python has its own characteristics and methods associated with it, allowing you to perform various operations and manipulations. Python also provides built-in functions to convert between different data types.

It's important to note that variables in Python are dynamically typed, meaning that you can assign values of different types to the same variable. Python automatically adjusts the type of the variable based on the assigned value. This flexibility allows for easy and convenient programming.

6 Operators

Python includes a variety of operators for performing operations on values. These include arithmetic operators, comparison operators, assignment operators, logical operators, and more.

6.1 Arithmetic Operators

Python includes arithmetic operators for performing mathematical operations on values. Here are some commonly used arithmetic operators:

- $+$ **(Addition)**: Adds two values.
- $-$ **(Subtraction)**: Subtracts one value from another.
- $*$ **(Multiplication)**: Multiplies two values.
- $/$ **(Division)**: Divides one value by another.
- $//$ **(Floor Division)**: Performs division and rounds the result down to the nearest whole number.
- $\%$ **(Modulus)**: Returns the remainder after division.
- $**$ **(Exponentiation)**: Raises a value to the power of another value.

Here are some examples of using arithmetic operators:

Example 6.1.

```
x = 5 + 3 # Addition: x = 8
y = 10 - 4 # Subtraction: y = 6
z = 3 * 2 # Multiplication: z = 6
w = 12 / 4 # Division: w = 3.0
v = 13 // 5 # Floor Division: v = 2
r = 15 % 4 # Modulus: r = 3
s = 2 \** 4 # Exponentiation: s = 16
```

In these examples, the arithmetic operators perform the specified operations on the values and assign the results to the respective variables.

6.2 Comparison Operators

Comparison operators are used to compare two values and return a boolean result (True or False) based on the comparison. Here are some commonly used comparison operators:

- $>$ **(Greater than)**: Returns True if the left value is greater than the right value.
- $<$ **(Less than)**: Returns True if the left value is less than the right value.
- $>=$ **(Greater than or equal to)**: Returns True if the left value is greater than or equal to the right value.

- <= **(Less than or equal to)**: Returns True if the left value is less than or equal to the right value.
- == **(Equal to)**: Returns True if the left value is equal to the right value.
- ! = **(Not equal to)**: Returns True if the left value is not equal to the right value.

Here are some examples of using comparison operators:

Example 6.2.

```
x = 5 > 3 # Greater than: x = True
y = 10 < 4 # Less than: y = False
z = 3 >= 2 # Greater than or equal to: z = True
w = 12 <= 4 # Less than or equal to: w = False
v = 5 == 5 # Equal to: v = True
r = 15 != 4 # Not equal to: r = True
```

In these examples, the comparison operators compare the values and return the boolean results based on the comparison.

6.3 Assignment Operators

Assignment operators are used to assign values to variables. The basic assignment operator in Python is =. It assigns the value on the right side to the variable on the left side.

Here are some examples of using assignment operators:

Example 6.3.

```
x = 5 # Assigns the value 5 to the variable x
y = 2 # Assigns the value 2 to the variable y
```

Python also provides compound assignment operators that combine an arithmetic operation with assignment. These include + =, − =, ∗ =, / =, and

6.4 Logical Operators

Logical operators are used to combine and manipulate boolean values (`True` and `False`). Python includes three logical operators: `and`, `or`, and `not`.

Here's an example that demonstrates the usage of logical operators:

Example 6.4.

```
x = 5
y = 10

print(x > 0 and y < 100) # Output: True
print(x > 0 or y < 100) # Output: True
print(not x > 0) # Output: False
```

In this example, the `and` operator returns `True` if both the expressions on its left and right side are `True`. The `or` operator returns `True` if at least one of the expressions on its left or right side is `True`. The `not` operator negates the boolean value.

7 Control Flow

Control flow refers to the order in which the program's code executes. The control flow of a Python program is regulated by conditional statements, loops, and function calls.

7.1 If Statements

The ‘if‘ statement is used for conditional execution in Python. It executes a block of code if a specified condition is true:

Example 7.1.

```
1 if 5 > 2:
2     print("Five is greater than two!")
```

In this example, the condition is ‘5 ¿ 2‘ which is true, so the print statement is executed.

7.1.1 The else keyword

The ‘else‘ keyword in Python is used to define a block of code to be executed if the condition in the ‘if‘ statement is false:

Below the function which will tell if the 5 is less than 2 or else if the 5 is greater or equal than 5

Example 7.2.

```
1 if 5 < 2:
2     print("Five is less than two!")
3 else:
4     print("Five is not less than two!")
```

In this example, because the condition ‘5 ¡ 2‘ is false, the print statement under the ‘else‘ keyword is executed.

7.1.2 The elif keyword

The ‘elif‘ keyword in Python is short for "else if". It allows you to specify a new condition to be checked if the first condition is false:

Example 7.3. *Lets make the consideration if the x is less than defined numbers:*

```
1 x = 20
2 if x < 10:
3     print("x is less than 10")
4 elif x < 30:
5     print("x is less than 30")
6 else:
7     print("x is 30 or more")
```

In this example, because ‘x‘ is ‘20‘, the condition ‘x ¡ 10‘ is false, so Python moves on to the ‘elif‘ statement. The condition ‘x ¡ 30‘ is true, so the corresponding print statement is executed.

7.2 While Loops

A ‘while‘ loop in Python executes a block of code as long as a specified condition is true:

Example 7.4.

```
i = 1
while i < 6:
    print(i)
    i += 1
```

In this example, as long as ‘i‘ is less than ‘6‘, the loop will continue to execute and print the value of ‘i‘. The ‘i += 1‘ statement increments ‘i‘ by ‘1‘ after each loop iteration.

Here are a few more complex examples and constructions using while loops:

Example 7.5.

```
% Sum of numbers from 1 to 10
total = 0
i = 1
while i <= 10:
    total += i
    i += 1
print("Sum:", total)

% Countdown from 10 to 1
num = 10
while num >= 1:
    print(num)
    num -= 1
print("Blastoff!")

% Finding the first Fibonacci number greater than 1000
a, b = 0, 1
while b <= 1000:
    a, b = b, a + b
print("First Fibonacci number greater than 1000:", b)
```

In Example 1, the loop calculates the sum of numbers from 1 to 10 using the `total` variable. Example 2 demonstrates a countdown from 10 to 1, and the loop prints the current value of `num` in each iteration. Finally, Example 3 finds the first Fibonacci number greater than 1000 using the variables `a` and `b` to keep track of the current and previous Fibonacci numbers.

Feel free to customize and experiment with these examples to further explore the capabilities of while loops in Python.

7.3 For Loops

A ‘for‘ loop in Python is used to iterate over a sequence (like a list, tuple, set, or string) or other iterable objects. Iterating over a sequence is called traversal.

7.3.1 Iterating over lists

Here's an example of a 'for' loop in Python that iterates over a list:

Example 7.6.

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

In this example, 'fruit' is a variable that takes the value of the next item in 'fruits' each time through the loop. So, this 'for' loop will print each fruit in the 'fruits' list.

7.3.2 Using the range() function

The range() function in Python is used to generate a sequence of numbers which might be used to iterating through using for loop. Here are some examples:

1. With only one argument:

   Example 7.7.

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

   This will output the numbers 0 through 4. Note that 'range(5)' generates numbers from 0 to 4, not up to 5. This is because 'range()' generates numbers up to, but not including, the end value you specify.

2. With two arguments:

   Example 7.8.

   ```
       for i in range(2, 5):
           print(i)
   ```

   This will output the numbers 2 through 4. Here, 'range(2, 5)' generates numbers from 2 up to, but not including, 5.

3. With three arguments:

   Example 7.9.

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

   This will output the numbers 0, 2, 4, 6, and 8. Here, 'range(0, 10, 2)' generates numbers from 0 up to, but not including, 10, with a step of 2.

7.3.3 Nested for loops

You can use a ‘for‘ loop inside another ‘for‘ loop to iterate over multiple dimensions of data. This is known as a nested loop. For example:

Example 7.10.

```
for i in range(3):
    for j in range(3):
        print(i, j)
```

In this example, the outer loop runs three times, and each time it runs, the inner loop also runs three times. This results in a total of nine iterations of the inner loop, once for each combination of ‘i‘ and ‘j‘ values.

7.3.4 The enumerate() function

When iterating over a sequence, you may sometimes want to know the index of the current item in the sequence. You can use the ‘enumerate()‘ function for this:

Example 7.11.

```
for i, fruit in enumerate(fruits):
    print(f"The fruit at index {i} is {fruit}.\n")
```

In this example, ‘enumerate(fruits)‘ generates a sequence of pairs, where the first element of each pair is the index and the second element is the corresponding item from ‘fruits‘. The ‘for‘ loop then unpacks each pair into ‘i‘ and ‘fruit‘. At the end of the f-stting constructed there is a new line symbol `\n`.

8 Functions

Functions in Python are defined using the ‘def‘ keyword. Functions allow for code reuse and can make programs more modular and easier to understand. Here’s how you can define a function in Python:

Example 8.1.

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

In this example, ‘my$_function$‘$isafunctionthatprints$”$Hellofromafunction$”$whencalled$.

8.1 Calling a Function

To call a function in Python, you simply need to write the function’s name followed by parentheses. Here’s how you can call the function we defined earlier:

```
my_function()
```

8.2 Parameters and Arguments

Functions can also take parameters, which are values that you can pass into the function. The parameters are defined in the function definition, and the values you pass in are called arguments. Here’s an example of a function with parameters:

Lets imagine we have to write the function to greet the user after they input their name:

Example 8.2.

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

greet("Alice")
```

In this example, ‘name‘ is a parameter. You can call this function with an argument like this: This will output: ”Hello, Alice!”.

8.3 Return Values

Functions in Python can also return values. This is done using the ‘return‘ statement. Here’s an example of a function that returns a value:

Example 8.3.

```
def square(number):
    return number ** 2
```

In this example, the ‘square‘ function takes a number as a parameter and returns the square of that number. You can call this function and store its return value in a variable like this:

```
squared = square(5)
print(squared)  # Output: 25
```

8.4 Default Parameter Value

In Python, you can also provide a default value for a function's parameters. This value will be used if the function is called without an argument for that parameter. Here's an example:

Lets create a function that takes the parameter of the name of the user but also has its default value set as the Stranger string:

Example 8.4.

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

In this example, if the 'greet' function is called without an argument, it will use "Stranger" as the default value for the 'name' parameter:

```
greet()  # Output: Hello, Stranger!
```

8.5 Multiple Parameters and Arguments

Python functions can take multiple parameters. You just need to separate them with a comma in the function definition. Here's an example:

A function to compute the sum of two numbers:

Example 8.5.

```
def add_numbers(num1, num2):
        return num1 + num2
```

You can call this function with two arguments like this:

```
sum = add_numbers(5, 7)
print(sum) # Output: 12
```

8.6 Variable-length Arguments

Sometimes you may want to define a function that can take any number of arguments. You can do this in Python using *args and **kwargs. Here's an example:

A function that accepts any number of arguments and returns their sum:

Example 8.6.

```
def add_numbers(*args):
return sum(args)
```

You can call this function with any number of arguments like this:

```
sum = add_numbers(1, 2, 3, 4, 5)
print(sum) # Output: 15
```

8.7 Recursive Functions

A recursive function is a function that calls itself during its execution. This enables the function to be repeated several times, as it can call itself during its execution. Here's an example of a recursive function:

A function to compute the factorial of a number using recursion:

Example 8.7.

```
def factorial(n):
if n == 1:
return 1
else:
return n * factorial(n-1)
```

You can call this function like this:

```
fact = factorial(5)
print(fact) # Output: 120
```

8.8 Lambda Functions

In Python, anonymous function means that a function is without a name. The lambda keyword is used to create anonymous functions. Here's an example:

A lambda function to compute the square of a number:

Example 8.8.

```
square = lambda x : x ** 2
```

You can call this function like this:

```
sq = square(5)
print(sq) # Output: 25
```

9 Object-Oriented Programming

Object-Oriented Programming (OOP) is a programming paradigm that focuses on the concept of objects, which are instances of classes. It allows for the organization of code into reusable and modular components. Python is an object-oriented programming language that fully supports OOP concepts. In this section, we will explore the key principles and features of OOP in Python.

9.1 Classes and Objects

At the core of OOP is the concept of classes and objects. A class is a blueprint for creating objects, which are instances of that class. It defines the attributes (data) and methods (functions) that the objects will have. Here's an example of a simple class in Python:

Example 9.1.

```
class Circle:
    def __init__(self, radius):
        self.radius = radius

    def calculate_area(self):
        return 3.14 * self.radius**2

# Create an instance of the Circle class
my_circle = Circle(5)

# Accessing attributes and calling methods
print(my_circle.radius)  # Output: 5
print(my_circle.calculate_area())  # Output: 78.5
```

In the above example, we define a `Circle` class with an attribute `radius` and a method `calculate_area()`. We then create an instance of the `Circle` class called `my_circle` and access its attributes and methods using the dot notation.

9.2 Inheritance

Inheritance is a powerful feature in OOP that allows a class to inherit attributes and methods from another class. The class that is being inherited from is called the base class or superclass, and the class that inherits from it is called the derived class or subclass. The derived class can add its own attributes and methods or override the ones inherited from the base class.

Example 9.2.

```
class Animal:
    def __init__(self, name):
        self.name = name

    def speak(self):
        raise NotImplementedError("Subclass must implement
            this method")
```

```
7
8
9  class Dog(Animal):
10     def speak(self):
11         return "Woof!"
12
13 class Cat(Animal):
14     def speak(self):
15         return "Meow!"
16
17 my_dog = Dog("Buddy")
18 my_cat = Cat("Whiskers")
19
20 print(my_dog.speak())  # Output: Woof!
21 print(my_cat.speak())  # Output: Meow!
```

In the above example, we define an `Animal` base class with an abstract method `speak()`. We then create two derived classes, `Dog` and `Cat`, that inherit from the `Animal` class and provide their own implementation of the `speak()` method.

9.3 Encapsulation and Data Hiding

Encapsulation is the practice of bundling data and methods together within a class, hiding the internal details of how the data is stored and processed. In Python, we can achieve encapsulation by using private and protected attributes and methods.

Private attributes and methods are indicated by prefixing them with two underscores (__). They can only be accessed within the class itself and not from outside the class or its subclasses. Protected attributes and methods are indicated by prefixing them with a single underscore (_). They can be accessed within the class, its subclasses, and even from outside the class, but it is considered best practice not to do so. Here's an example:

Example 9.3.

```
1  class Car:
2      def __init__(self, brand, model, year):
3          self.__brand = brand
4          self._model = model
5          self.year = year
6
7      def start_engine(self):
8          print("Engine started.")
9
10     def __private_method(self):
11         print("This is a private method.")
12
13 my_car = Car("Toyota", "Camry", 2022)
14 print(my_car.__brand)  # Raises an AttributeError
15 print(my_car._model)  # Accessible, but conventionally
       considered protected
16 print(my_car.year)  # Accessible
```

```
my_car.start_engine()  # Outputs: Engine started.
my_car.__private_method()  # Raises an AttributeError
```

In the above example, the `Car` class has a private attribute `__brand`, a protected attribute `_model`, and a public attribute `year`. It also has a public method `start_engine()` and a private method `__private_method()`.

When accessing the attributes and methods, we see that `__brand` raises an `AttributeError` because it is private and cannot be accessed from outside the class. `_model` is accessible, but it is conventionally considered protected and should be accessed with caution. `year` can be accessed freely as it is a public attribute. Similarly, the public method `start_engine()` can be called, but the private method `__private_method()` raises an `AttributeError`.

Encapsulation helps in creating robust and maintainable code by controlling the access to class attributes and methods, preventing unintended modifications, and encapsulating the internal implementation details.

These are just some of the key features of object-oriented programming in Python. There are many more concepts and techniques that can be explored to leverage the power of OOP in your Python programs.

9.4 Geometric Figures and Methods

In addition to the concepts discussed above, we can implement classes for geometric figures such as triangles, rectangles, and circles, with methods to calculate their areas and perimeters. Here's an example:

Example 9.4.

```
import math

class GeometricFigure:
    def area(self):
        raise NotImplementedError("Subclass must implement
            this method")

    def perimeter(self):
        raise NotImplementedError("Subclass must implement
            this method")

class Triangle(GeometricFigure):
    def __init__(self, base, height):
        self.base = base
        self.height = height

    def area(self):
        return 0.5 * self.base * self.height

    def perimeter(self):
        return 3 * self.base

class Rectangle(GeometricFigure):
```

```
22      def __init__(self, length, width):
23          self.length = length
24          self.width = width
25
26      def area(self):
27          return self.length * self.width
28
29      def perimeter(self):
30          return 2 * (self.length + self.width)
31
32  class Circle(GeometricFigure):
33      def __init__(self, radius):
34          self.radius = radius
35
36      def area(self):
37          return math.pi * self.radius**2
38
39      def perimeter(self):
40          return 2 * math.pi * self.radius
41
42  # Create instances of geometric figures
43  triangle = Triangle(5, 3)
44  rectangle = Rectangle(4, 6)
45  circle = Circle(2)
46
47  # Calculate and display areas and perimeters
48  print("Triangle: Area =", triangle.area(), "Perimeter =",
        triangle.perimeter())
49  print("Rectangle: Area =", rectangle.area(), "Perimeter =",
        rectangle.perimeter())
50  print("Circle: Area =", circle.area(), "Circumference =",
        circle.perimeter())
```

In the above example, we explored the concepts of inheritance, polymorphism, and encapsulation in object-oriented programming.

Inheritance allows a class to inherit attributes and methods from a base class. In this case, the Triangle, Rectangle, and Circle classes inherit from the GeometricFigure class. By doing so, they gain access to the area() and perimeter() methods defined in the base class. This promotes code reuse and allows for a hierarchical organization of related classes.

Polymorphism allows objects of different classes to be treated as objects of a common base class. In our example, we can treat instances of Triangle, Rectangle, and Circle as objects of the GeometricFigure class. This means we can use them interchangeably when calling the area() and perimeter() methods. The appropriate implementation for each class will be automatically invoked at runtime, based on the actual type of the object.

Encapsulation is the practice of bundling data and methods together within a class, hiding the internal details of how the data is stored and processed. In our example, the internal workings of each geometric figure class are encapsulated within their respective

classes. The data (such as base, height, length, width, and radius) is stored as instance variables and can only be accessed or modified through the defined methods (area() and perimeter()). This helps in creating modular and maintainable code, as the internal implementation can be changed without affecting the code that uses the objects.

By employing inheritance, polymorphism, and encapsulation, we can design flexible and modular object-oriented programs that are easier to understand, maintain, and extend.

10 Turtle library

10.1 General operations on the Turtle pseudo object

The Turtle library is a popular tool in Python used for creating simple graphics. It is especially suitable for beginner programmers because of its easy-to-understand syntax and interactive nature. The library creates a panel and pen that you can move around to create intricate designs.

The following is an example of how to use the Turtle library to draw a square:

Example 10.1.

```
from turtle import *

# Move turtle
for _ in range(4):
    forward(100)
    right(90)

# Keep the window open
exitonclick()
```

In the above code, we first import the turtle module. Then, we create a turtle screen with a white background. Next, we create a turtle object 'my_turtle'. In the for loop, the turtle moves forward by 100 units and then turns right by 90 degrees. This is repeated four times to complete a square. The 'turtle.done()' line is used to keep the turtle graphics window open.

The Turtle library provides numerous other functionalities, such as changing the pen color, filling shapes with color, and much more, allowing users to create a wide range of graphics.

10.2 Changing the pen color and filling shapes

You can change the color of the pen in the Turtle library using the 'color()' function. You can also fill shapes using the 'begin$_fill()$'and'$end_fill()$'$functions.Hereisanexample$:

Example 10.2.

```
from turtle import *

color("red")

begin_fill()
for _ in range(4):
    forward(100)
    right(90)
end_fill()

exitonclick()
```

In the above code, we set the pen color to red using the 'color()' function. Then, we use the 'begin$_fill()$'$functionbeforedrawingthesquareandthe$'$end_fill()$'$functionafterdrawingthesquare.T$

10.3 Changing the pen size

You can change the pen size in the Turtle library using the ‘pensize()‘ function. Here is an example:

Example 10.3.

```
from turtle import *

pensize(10)

for _ in range(4):
    forward(100)
    right(90)

exitonclick()
```

In the above code, we set the pen size to 10 using the ‘pensize()‘ function. This makes the lines drawn by the turtle thicker.

11 Complex Python Programs

In this section, we will present some more complex Python programs to demonstrate the power and flexibility of the language.

11.1 Factorial Function

Factorial of a non-negative integer n is the product of all positive integers less than or equal to n. It is denoted by n!.

Here's a Python function that calculates the factorial of a number using recursion:

Example 11.1.

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n-1)
```

11.2 Fibonacci Sequence

The Fibonacci sequence is a sequence of numbers where the next number in the sequence is found by adding up the two numbers before it.

Here's a Python function that returns the n-th number in the Fibonacci sequence using recursion:

Example 11.2.

```
def fibonacci(n):
    if n <= 1:
        return n
    else:
        return fibonacci(n-1) + fibonacci(n-2)
```

11.3 Factorial Function with Dynamic Programming

By using dynamic programming techniques, we can optimize the calculation of the factorial by storing previously calculated values in a lookup table:

```
def factorial(n, lookup={}):
    if n == 0 or n == 1:
        lookup[n] = 1

    # check if value already exists in lookup table
    if n not in lookup:
        lookup[n] = n * factorial(n-1)

    return lookup[n]
```

This approach ensures that each factorial is only calculated once and then stored in a dictionary for quick access.

11.4 Fibonacci Sequence with Dynamic Programming

Similarly, we can optimize the Fibonacci sequence calculation using dynamic programming to avoid recalculating the same values:

```
def fibonacci(n, lookup={}):
    # Base case
    if n == 0 or n == 1:
        lookup[n] = n

    # If the value is not calculated previously, calculate it
    if n not in lookup:
        lookup[n] = fibonacci(n-1, lookup) + fibonacci(n-2,
           lookup)

    return lookup[n]
```

With dynamic programming, the above Fibonacci function will run significantly faster when called repeatedly or with large inputs, because it avoids the expensive computation of recalculating the Fibonacci sequence from scratch each time.

11.5 Sorting Algorithm: Bubble Sort

Bubble Sort is a simple sorting algorithm that repeatedly steps through the list, compares adjacent elements and swaps them if they are in the wrong order.

Here's a Python implementation of Bubble Sort:

Example 11.3.

```
def bubble_sort(arr):
    n = len(arr)
    for i in range(n):
        for j in range(0, n-i-1):
            if arr[j] > arr[j+1] :
                arr[j], arr[j+1] = arr[j+1], arr[j]
    return arr
```

11.6 Prime Numbers

A prime number is a natural number greater than 1 that has no positive divisors other than 1 and itself.

Here's a Python function that checks if a number is prime:

Example 11.4.

```
def is_prime(n):
    if n <= 1:
        return False
    for i in range(2, n):
        if n % i == 0:
            return False
    return True
```

12 Data Structures in Python

Python incorporates several built-in data structures for the efficient storage and manipulation of data, which include lists, tuples, sets, and dictionaries.

12.1 Lists

In Python, a list is an ordered collection of items which are not restricted to a specific type. Lists are defined by enclosing a comma-separated sequence of items within square brackets '[]'. Here is an illustrative example of a list in Python:

Example 12.1.

```
fruits = ["apple", "banana", "cherry"]
```

You can add an item to the list using the append() method, and remove an item using the remove() method:

Example 12.2.

```
fruits.append("date")
fruits.remove("banana")
```

12.2 Tuples

A tuple is akin to a list in terms of being an ordered collection of items, however, it is defined with parentheses '()' as opposed to square brackets. The principal difference lies in the fact that tuples are immutable. Here's an example:

Example 12.3.

```
fruits = ("apple", "banana", "cherry")
```

Accessing elements in a tuple is similar to accessing elements in a list:

Example 12.4.

```
print(fruits[1])  # Output: "banana"
```

12.3 Sets

A set in Python is characterized as an unordered collection of unique items. Sets are defined by enclosing a comma-separated list of items within curly braces ''. Here's an example of a set:

Example 12.5.

```
fruits = {"apple", "banana", "cherry"}
```

You can add an item to a set using the add() method, and remove an item using the remove() method:

Example 12.6.

```
fruits.add("date")
fruits.remove("banana")
```

12.4 Dictionaries

A dictionary in Python constitutes an unordered collection of items. Each item stored in a dictionary has a key and a corresponding value. The key can be utilized to access the related value. Dictionaries are defined by enclosing a comma-separated list of key-value pairs within curly braces ‘‘. The key and value are separated by a colon ‘:‘. Here’s an example of a dictionary:

Example 12.7.

```
person = {"name": "Alice", "age": 25}
```

You can change the value of an item in a dictionary like this:

Example 12.8.

```
person["age"] = 30
```

You can add a new item to a dictionary like this:

Example 12.9.

```
person["profession"] = "Engineer"
```

This page was left empty with accordance to author's hidden intention...

13 Projects

13.1 Project 1

13.1.1 Project Description

The task of this program is to create an interactive drawing application using the turtle module in Python. The program allows the user to control a turtle object on the screen and perform various actions.

13.1.2 Specification(may be modified by user as the task is creative one)

Here is a breakdown of the tasks performed by the program:

[label=0.]Set up the Turtle:

1.
   - Create a turtle object.
   - Set the turtle's speed to 100.
   - Set the turtle's initial color to red.
   - Set the turtle's width to 1.
   - Set the turtle's shape to "turtle".
   - Put the turtle's pen down to start drawing.

2. Define Color Changing Functions:
   - Implement `turtle_color_red()` to change the turtle's color to red.
   - Implement `turtle_color_green()` to change the turtle's color to green.

3. Define Mouse Event Function:
   - Implement `fxn(x, y)` to handle mouse drag events.
   - Stop backtracking of the turtle.
   - Adjust the turtle's angle and direction towards the new coordinates (x, y).
   - Move the turtle to the new coordinates (x, y).
   - Enable the function to be called again for further dragging.

4. Define Keyboard Event Functions:
   - Implement `move_forward()` to move the turtle forward by 50 units.
   - Implement `move_backward()` to move the turtle backward by 50 units.
   - Implement `turn_left()` to rotate the turtle left by 45 degrees.
   - Implement `turn_right()` to rotate the turtle right by 45 degrees.
   - Implement `fill_screen()` to fill the entire screen with color.

5. Set up Event Listeners:
   - Get the turtle's screen object.
   - Enable listening for key and mouse events.

- Register event handlers for specific keys and mouse clicks.
- When events occur, the corresponding functions are called to perform the desired actions.

6. Enter the Main Event Loop:

- Start the turtle's event loop.
- The program continuously listens for events and responds accordingly.
- The program remains interactive until the window is closed.

The main goal of this program is to provide an interactive drawing experience where the user can control the turtle's movement, change its color, and fill the screen with color. The program utilizes various event-driven functions to respond to user inputs and update the turtle's behavior on the screen.

13.1.3 The one of possible solutions

Example 13.1.

```
from turtle import *
from random import randint
from time import sleep

t = Turtle()  # Create a turtle object
t.speed(100)  # Set the turtle's speed
t.color("red")  # Set the turtle's color
t.width(1)  # Set the turtle's width
t.shape("turtle")  # Set the turtle's shape
t.pendown()  # Put the turtle's pen down to start drawing

def turtle_color_red():
    t.color("red")  # Change the turtle's color to red

def turtle_color_green():
    t.color("green")  # Change the turtle's color to green

def fxn(x, y):
    t.ondrag(None)  # Stop backtracking
    t.setheading(t.towards(x, y))  # Move the turtle's
       angle and direction towards x and y
    t.goto(x, y)  # Go to x, y
    t.ondrag(fxn)  # Call the function again for further
       dragging

def move_forward():
    t.forward(50)  # Move the turtle forward by 50 units

def move_backward():
    t.backward(50)  # Move the turtle backward by 50 units

```

```
30 def turn_left():
31     t.left(45)  # Turn the turtle left by 45 degrees
32
33 def turn_right():
34     t.right(45)  # Turn the turtle right by 45 degrees
35
36 def fill_screen():
37     t.begin_fill()  # Start filling the shape with color
38     t.goto(-scr.window_width() / 2, -scr.window_height() /
          2)  # Go to the bottom-left corner of the screen
39     t.goto(scr.window_width() / 2, -scr.window_height() /
          2)  # Go to the bottom-right corner of the screen
40     t.goto(scr.window_width() / 2, scr.window_height() / 2)
           # Go to the top-right corner of the screen
41     t.goto(-scr.window_width() / 2, scr.window_height() /
          2)  # Go to the top-left corner of the screen
42     t.goto(-scr.window_width() / 2, -scr.window_height() /
          2)  # Go back to the bottom-left corner of the screen
43     t.end_fill()  # Stop filling the shape with color
44
45 t.speed(10)  # Set the turtle's speed
46 scr = t.getscreen()  # Get the turtle's screen
47 scr.listen()  # Enable listening for key and mouse events
48 scr.onkey(turtle_color_red, "r")  # Call turtle_color_red
      function when 'r' key is pressed
49 scr.onkey(turtle_color_green, "g")  # Call
      turtle_color_green function when 'g' key is pressed
50 scr.onkey(move_forward, "Up")  # Call move_forward function
       when 'Up' arrow key is pressed
51 scr.onkey(move_backward, "Down")  # Call move_backward
      function when 'Down' arrow key is pressed
52 scr.onkey(turn_left, "Left")  # Call turn_left function
      when 'Left' arrow key is pressed
53 scr.onkey(turn_right, "Right")  # Call turn_right function
      when 'Right' arrow key is pressed
54 scr.onclick(fxn)  # Call fxn function when the screen is
      clicked
55 scr.onkey(fill_screen, "f")  # Call fill_screen function
      when 'f' key is pressed
56 scr.mainloop()  # Start the turtle's event loop
```

13.2 Project 2 specification

13.2.1 Task 1 (10 points)

Construct a list of MIT classes in the following format (in this context, the list is not a data structure, but a simple enumeration of elements):

- Course 1 - Civil and Environmental Engineering
- Course 2 - Mechanical Engineering
- Course 3 - Materials Science and Engineering
- Course 4 - Architecture
- Course 5 - Chemistry
- Course 6 - Electrical Engineering and Computer Science
- Course 7 - Biology
- Course 8 - Physics
- Course 9 - Brain and Cognitive Sciences
- Course 10 - Chemical Engineering

Attention! Employing the `.format` function will guarantee acquiring 5 points of task performance.

13.2.2 Task 2 (10 points)

Request a user to input a number of an integer in the range $[1, \ldots, i, \ldots, 10]$, where $i \in \mathbb{N}$, and $1 \leq i \leq 10$. Once the program receives the number from the user, it returns the name of the course corresponding to the given number.
Example:

```
Program:  Enter the number...
User:  1
Program:  Course 1 - Civil and Environmental Engineering
```

13.2.3 Task 3 (10 points)

Consider the fraction $\frac{a}{b} = \frac{nominator}{denominator}$. In this task, ask the user to input the numerator and denominator values. The program should return the result and (if possible) the remainder of this division operation. For this task, create a variable used in a while loop called `i = a` and another variable called `answer`; while `i > 0`, let `i = i - b` and `answer = answer + 1`. If `i == 0`, the program should print the result as the value of the `str` function type in the following format: `"Result" + str(answer)`. If `i != 0`, the program prints `"Result"` as the `answer - 1` string value and then the program prints: `"Reminder:" + str(i+b)`. The names of any variables here in the task may be chosen by the student. Following this example for naming the variables may make the solution more readable for the grader.

13.3 Project 3 specification

13.3.1 Task 1 (10 points)

Write a function `calculate_the_sum_of_n_numbers(n)` which calculates the sum of n numbers, where n is a natural finite number.
Example:

```
calculate_the_sum_of_n_numbers(100)
```

This will be used to calculate the sum $1 + 2 + 3 + \cdots + 100 = \frac{100*101}{2} = 50 * 101 = 5050$.

13.3.2 Task 2 (10 points)

Write a function `nums_to_n(n)` to print all the odd numbers from 1 to n, where n is a natural finite number, and if the number is divisible by 5, the function should draw the user's attention to it. The function does not return anything, it prints all odd numbers from 1 to n.
Example:

```
nums_to_n(10)
```

This code should print:

```
1;
3;
5 is divisible by 5;
7;
9;
```

Remember to include a semicolon at the end of each line!

13.3.3 Task 3 (10 points)

Write a function named `analyse_the_number(x, a_less, b_greater)` which will:

- Inform the user if the number is odd or even;
- Inform the user if the number is greater than or equal to `"b_greater"`;
- Inform the user if the number is less than or equal to `"a_less"`;
- Tell the user what is the factorial of x: if $x = 6$, it will print $6 * 5 * 4 * 3 * 2 * 1$;
- Print $(x^{b\_greater})^{a\_less}$;
- If the number is from the set $[-2, 2]$ (the number x may be 2, -1, 0, 1, 2), the function will print the number as follows: if the number x is 2, the program will print: "two".

13.4 Specification of the Project 4

13.4.1 Task 1 (10 points)

Write a function `calculate_factorial(n)` that calculates and returns the factorial of a natural number n. This function should raise an exception if n is negative or if it's not an integer.
Example:

```
1 calculate_factorial(5)
```

This will return $5 * 4 * 3 * 2 * 1 = 120$.

13.4.2 Task 2 (10 points)

Write a function `print_fibonacci(n)` that prints the first n numbers in the Fibonacci sequence. The function should print each number on a new line. The function does not return anything, it only prints the Fibonacci numbers.
Example:

```
1 print_fibonacci(7)
```

This code should print:

```
0
1
1
2
3
5
8
```

13.4.3 Task 3 (10 points)

Write a function `is_prime(n)` that returns `True` if a number is prime and `False` otherwise. A prime number is a natural number greater than 1 that has no positive divisors other than 1 and itself.
Example:

```
1 is_prime(7)
```

This will return `True` because 7 is a prime number.

13.5 Specification of the Project 5

13.5.1 Task 1 (10 points)

Write a function `read_file(file_name)` that reads a text file and returns a list where each element is a line in the file. This function should handle possible exceptions due to file handling issues.
Example:

```
read_file('example.txt')
```

This function will read the text file 'example.txt' and return its contents as a list of strings.

13.5.2 Task 2 (10 points)

Write a function `find_pattern(file_name, pattern)` that uses regular expressions to find all occurrences of a specific pattern in a text file. This function should return a list of all matches.
Example:

```
find_pattern('example.txt', r'\d+')
```

This function will find all sequences of digits in 'example.txt' and return them as a list.

13.5.3 Task 3 (20 points)

Write a function `web_scrape(url, tag)` that uses BeautifulSoup to scrape a specific tag from a webpage. This function should return a list of strings, each of which corresponds to the inner content of one instance of the tag in the HTML of the webpage.
Example:

```
web_scrape('https://example.com', 'h1')
```

This function will return the inner content of all 'h1' tags in the HTML of 'https://example.com'.

13.6 Specification of the Project 6

13.6.1 Task 1 (10 points)

Write a function `is_prime(n)` that checks whether a given integer n is a prime number. The function should return `True` if the number is prime and `False` otherwise.
Example:

```
1 is_prime(7)
```

This function will return `True`, as 7 is a prime number.

13.6.2 Task 2 (15 points)

Write a function `calculate_factorial(n)` that calculates and returns the factorial of a number n. The function should raise an exception if n is negative or not an integer.
Example:

```
1 calculate_factorial(5)
```

This function will return $5 * 4 * 3 * 2 * 1 = 120$.

13.6.3 Task 3 (15 points)

Write a function `calculate_combination(n, r)` that calculates and returns the combination of n items taken r at a time, where n and r are natural numbers and $r \leq n$.
Example:

```
1 calculate_combination(5, 2)
```

This function will return 10, as there are 10 ways to choose 2 items from 5.

13.6.4 Task 4 (20 points)

Write a function `calculate_mean_median(file_name)` that reads a file containing a list of numbers, one per line, and returns a tuple containing the mean and median of these numbers. The function should handle possible exceptions due to file handling issues.
Example:

```
1 calculate_mean_median('numbers.txt')
```

This function will return a tuple (mean, median) calculated from the numbers in the file 'numbers.txt'.

13.6.5 Task 5 (20 points)

Write a function `plot_distribution(file_name)` that reads a file containing a list of numbers and uses matplotlib to plot a histogram of the data. The function should handle possible exceptions due to file handling issues.
Example:

```
1 plot_distribution('numbers.txt')
```

This function will read the numbers from the file 'numbers.txt' and display a histogram of the data.

13.7 Specification of the Project 7

13.7.1 Task 1 (20 points)

Define a Python class `Matrix` to represent a mathematical matrix. The class should include methods for matrix addition, matrix subtraction, matrix multiplication, and transpose of a matrix.
Example:

```
matrix1 = Matrix([[1, 2], [3, 4]])
matrix2 = Matrix([[5, 6], [7, 8]])
print(matrix1.add(matrix2))
print(matrix1.subtract(matrix2))
print(matrix1.multiply(matrix2))
print(matrix1.transpose())
```

This should print the results of the matrix operations.

13.7.2 Task 2 (20 points)

Write a Python function `fibonacci(n)` that calculates the nth Fibonacci number using recursion.
Example:

```
print(fibonacci(10))
```

This should print the 10th Fibonacci number.

13.7.3 Task 3 (20 points)

Write a Python function `simulate_random_walk(steps)` that simulates a one-dimensional random walk of a specified number of steps using NumPy and returns the final position.
Example:

```
print(simulate_random_walk(1000))
```

This should print the final position of the random walk after 1000 steps.

13.7.4 Task 4 (20 points)

Write a Python function `plot_random_walk(steps)` that simulates a one-dimensional random walk of a specified number of steps and creates a plot of the walk using Matplotlib.
Example:

```
plot_random_walk(1000)
```

This should display a plot of the random walk after 1000 steps.

13.7.5 Task 5 (20 points)

Write a Python function `simulate_random_walks(number_of_walks, steps)` that simulates a specified number of one-dimensional random walks of a specified number of steps each. The function should return the average final position and the standard deviation of the final positions of the walks.
Example:

```
average, std_dev = simulate_random_walks(100, 1000)
print('Average:', average)
print('Standard Deviation:', std_dev)
```

This should print the average and standard deviation of the final positions of 100 random walks of 1000 steps each.

13.8 Specification of the Project 8

13.8.1 Task 1 (25 points)

Write a Python function `solve_quadratic(a, b, c)` that solves a quadratic equation of the form $ax^2 + bx + c = 0$. The function should return the roots of the equation.
Example:

```
print(solve_quadratic(1, -3, 2))
```

This should print the roots of the equation $x^2 - 3x + 2 = 0$.

13.8.2 Task 2 (25 points)

Write a Python function `function_analysis(f, x_range)` that plots a given function $f(x)$ over a specified range of x values, calculates the maximum and minimum of the function over that range, and integrates the function over that range. The function should use the matplotlib library for plotting and the scipy library for integration.
Example:

```
import numpy as np
def f(x):
    return x**2
function_analysis(f, np.linspace(0, 1, 100))
```

This should plot the function $f(x) = x^2$ over the range from 0 to 1, print the maximum and minimum values over that range, and print the integral of $f(x)$ over that range.

13.8.3 Task 3 (25 points)

Write a Python function `solve_system(A, b)` that solves a system of linear equations given by the matrix equation $Ax = b$. The function should use the numpy library to solve the equation.
Example:

```
A = np.array([[3, 1], [1, 2]])
b = np.array([9, 8])
print(solve_system(A, b))
```

This should print the solution to the system of equations $3x_1 + x_2 = 9$ and $x_1 + 2x_2 = 8$.

13.8.4 Task 4 (25 points)

Write a Python function `calculate_derivative(f, x)` that calculates the derivative of a function at a given point. The function should use the scipy library to calculate the derivative.
Example:

```
def f(x):
    return x**3
print(calculate_derivative(f, 2))
```

This should print the derivative of the function $f(x) = x^3$ at $x = 2$.

13.9 Specification of the Project 9

13.9.1 Task 1 (25 points)

Write a Python function `polynomial_coefficients(n)` that generates the coefficients of a polynomial of degree n. The coefficients should be random integers in the range from -10 to 10. The function should return a list of the coefficients.
Example:

```
print(polynomial_coefficients(3))
```

This might print the list `[-2, 5, -10, 1]`, which corresponds to the polynomial $-2x^3 + 5x^2 - 10x + 1$.

13.9.2 Task 2 (25 points)

Write a Python function `polynomial_value(coefs, x)` that calculates the value of a polynomial at a given point x. The coefficients of the polynomial are given as a list of numbers.
Example:

```
coefs = [1, -3, 2]
x = 2
print(polynomial_value(coefs, x))
```

This should print the value of the polynomial $x^2 - 3x + 2$ at $x = 2$.

13.9.3 Task 3 (25 points)

Write a Python function `solve_quadratic(coefs)` that solves a quadratic equation given by its coefficients. The coefficients are given as a list of three numbers. The function should return the roots of the equation.
Example:

```
coefs = [1, -3, 2]
print(solve_quadratic(coefs))
```

This should print the roots of the quadratic equation $x^2 - 3x + 2 = 0$.

13.9.4 Task 4 (25 points)

Write a Python function `plot_polynomial(coefs, x_range)` that plots a polynomial given by its coefficients over a specified range of x values. The coefficients are given as a list of numbers. The function should use the matplotlib library for plotting.
Example:

```
coefs = [1, -3, 2]
x_range = np.linspace(-10, 10, 200)
plot_polynomial(coefs, x_range)
```

This should plot the polynomial $x^2 - 3x + 2$ over the range from -10 to 10.

13.10 Specification of the Project 10

13.10.1 Task 1 (20 points)

Write a Python function `scrape_webpage(url)` that takes in a URL of a webpage and scrapes the text content of that webpage using the BeautifulSoup library. The function should return a string of the webpage content. You must handle exceptions properly in your function to account for potential errors like network issues or invalid URLs.

13.10.2 Task 2 (20 points)

Extend the `scrape_webpage(url)` function to now save the scraped webpage content into a .txt file. The filename should be based on the webpage's title.

13.10.3 Task 3 (30 points)

Write a Python function `count_word_frequency(text)` that takes in a string of text and counts the frequency of each word in that text. The function should ignore common stopwords (like 'a', 'the', 'and', etc.) and return a dictionary where the keys are words and the values are their respective frequencies.

13.10.4 Task 4 (30 points)

Write a Python function `plot_word_frequency(word_frequency)` that takes in a dictionary of word frequencies (like the one produced by the `count_word_frequency(text)` function) and generates a bar chart of the top 10 most frequent words using matplotlib. The chart should have words on the x-axis and their frequencies on the y-axis.

Here some projects topics will follow...

14 Additional references

14.1 A Deep Dive into the Quadratic Equation

In the discipline of algebra, a preeminent form of a polynomial equation is the second-order polynomial equation, more commonly known as the quadratic equation. A polynomial function of a single variable x is typically expressed in the following general form:

$$f(x) = \sum_{i=0}^{n} a_i x^i = a_0 + a_1 x + a_2 x^2 + \cdots + a_{n-1} x^{n-1} + a_n x^n \tag{1}$$

where the $f : \mathbb{R} \to \mathbb{R}$ or generally $f : \mathbb{C} \to \mathbb{C}$, $a_i,\ i \in \mathbb{N}, i \in \{0, 1, ..., n\}$ are constants and they are known as the coefficients of the polynomial. The power n is a nonnegative integer and is called the degree of the polynomial. The coefficient a_n of the highest power is called the leading coefficient.

The standard form of second order equation is typically expressed as $ax^2 + bx + c = 0$, where a, b, and c are constants that are defined such that $a \neq 0$. The coefficients a, b, and c are often referred to as the quadratic, linear, and constant terms, respectively.

The problem is: find the values of the f domain, such that f(x) = 0, f defined in (1). An essential concept in the examination of these quadratic equations is the quadratic formula, which is universally used to calculate the roots of the equation. The quadratic formula is stated as:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \tag{2}$$

An interesting feature of the quadratic formula is the term under the square root, $b^2 - 4ac$. This term is known as the discriminant. It plays a pivotal role in determining the nature of the solutions to the quadratic equation.

- If the discriminant is positive, the equation possesses two distinct real roots.
- If the discriminant equals zero, the equation has one real root, often referred to as a repeated or double root.
- If the discriminant is negative, the equation gives rise to two complex roots.

Consider, for instance, the quadratic equation $x^2 - 5x + 6 = 0$. The coefficients of this equation are $a = 1$, $b = -5$, and $c = 6$. By calculating the discriminant, we find $(-5)^2 - 4 * 1 * 6 = 25 - 24 = 1$. As this is a positive value, we infer that the equation has two distinct real roots. By substituting the coefficients into the quadratic formula, we find the solutions to be:

$$x = \frac{-(-5) \pm \sqrt{(-5)^2 - 4 * 1 * 6}}{2 * 1} = \frac{5 \pm 1}{2}$$

Therefore, the roots of the equation are $x = \frac{5+1}{2} = 3$ and $x = \frac{5-1}{2} = 2$. The study of quadratic equations and their solutions is a fundamental part of algebra, and it underlies many of the more advanced topics in mathematics.

14.2 Solving Second Order Equations

A second order equation, also known as a quadratic equation, is an equation of the form $ax^2 + bx + c = 0$, where a, b, and c are constants, and $a \neq 0$.

The solutions to a quadratic equation are given by the quadratic formula (2):

The term under the square root, $b^2 - 4ac$, is known as the discriminant. It determines the nature of the roots of the quadratic equation.

- If the discriminant is positive, the equation has two distinct real roots.
- If the discriminant is zero, the equation has exactly one real root (or a repeated real root).
- If the discriminant is negative, the equation has two complex roots.

For example, let's solve the equation $x^2 - 5x + 6 = 0$. Here $a = 1$, $b = -5$, and $c = 6$. The discriminant is $(-5)^2 - 4 * 1 * 6 = 25 - 24 = 1$, which is positive. Thus the equation has two distinct real roots. Applying the quadratic formula gives:

$$x = \frac{-(-5) \pm \sqrt{(-5)^2 - 4 * 1 * 6}}{2 * 1} = \frac{5 \pm 1}{2}$$

So the solutions are $x = \frac{5+1}{2} = 3$ and $x = \frac{5-1}{2} = 2$.

Recommended Literature

References

[1] Matthes, E. (2019). *Python Crash Course: A Hands-On, Project-Based Introduction to Programming*. No Starch Press.

[2] Sweigart, A. (2015). *Automate the Boring Stuff with Python: Practical Programming for Total Beginners*. No Starch Press.

[3] Ramalho, L. (2015). *Fluent Python: Clear, Concise, and Effective Programming*. O'Reilly Media.

[4] Lutz, M. (2013). *Learning Python*. O'Reilly Media.

[5] VanderPlas, J. (2016). *Python Data Science Handbook: Essential Tools for Working with Data*. O'Reilly Media.

[6] Slatkin, B. (2015). *Effective Python: 59 Specific Ways to Write Better Python*. Addison-Wesley Professional.

[7] Downey, A. B. (2012). *Think Python: How to Think Like a Computer Scientist*. Green Tea Press.

[8] Python Software Foundation. *Python Documentation*. Retrieved from `https://docs.python.org`

[9] Python Software Foundation. *Python Tutorial*. Retrieved from `https://docs.python.org/3/tutorial/index.html`

[10] Real Python. *Python Tutorials and Articles*. Retrieved from `https://realpython.com`