I. SETTING UP YOUR COMPUTER

A. Python and Anaconda

To use the materials in this course, you will need to have Python and Anaconda installed on your system. Anaconda is a distribution of Python that comes with many useful packages and tools pre-installed, making it easier to manage dependencies and environments.

B. Installing Anaconda

Follow these steps to install Anaconda on Windows:

1. Download the Anaconda installer from the official Anaconda website.
2. Run the installer and follow the on-screen instructions.
3. After installation, open the Anaconda Prompt from the Start Menu.

C. Using Conda Environments

A conda environment is an isolated environment that allows you to install specific versions of Python and packages. This ensures that your projects are not affected by changes or updates in other environments.

To create and activate a conda environment, use the following commands in the Anaconda Prompt:

```
conda create --name myenv
conda activate myenv
```

Here, `myenv` is the name of your environment. Once activated, your prompt will typically change to indicate that you are working within that environment.

D. Using Jupyter Notebooks with Conda Environments

When using Jupyter Notebooks, it is important to ensure that the notebook uses the correct versions of Python and installed packages. Follow these steps to set up Jupyter Notebooks with your conda environments:

1. Install the `nb_conda_kernels` package in your base environment:

   ```
   conda install nb_conda_kernels
   ```

2. For each conda environment you want to use with Jupyter Notebook, install the `ipykernel` package:

   ```
   conda activate myenv
   conda install ipykernel
   python -m ipykernel install --user --name myenv --display-name "Python (myenv)"
   ```

   Here, `myenv` is the name of your environment, and `"Python (myenv)"` is the name that will appear in the Jupyter Notebook interface.

3. Launch Jupyter Notebook:

   ```
   jupyter notebook
   ```

   In the Jupyter Notebook interface, you can select the kernel corresponding to the environment you want to use by navigating to `Kernel > Change kernel` and selecting the appropriate one.

To summarize, `conda activate` is essential when working from the terminal to ensure you are using the correct environment. For Jupyter Notebooks, setting up `nb_conda_kernels` and `ipykernel` allows you to select and use kernels from different conda environments, ensuring that your notebooks run with the correct dependencies.

II. ELEMENTS OF PYTHON PROGRAMMING

Let's go through some general details of Python programming.

A. Statements

1. Simple Python Statements

Every line of a Python code is a statement. Sometimes, a statement can be long spanning multiple lines, which we call as multi-line statements.

```python
# Simple print statement
print("Hello, World!")

# Assigning values to variables
x = 10
y = 20
z = x + y

# Printing the result
print(f"The sum of {x} and {y} is {z}")
```

```
OUTPUT:
Hello, World!
The sum of 10 and 20 is 30
```

This code contains three types of Python statements, namely,

- **Print:** Output text to the console using `print()`.
- **Assignment:** Assign values to variables (e.g., `x = 10`).
- **Expression:** Evaluate expressions and store the result in a variable (e.g., `z = x + y`).

2. Comments in Python

Comments in Python are used to explain code and make it more readable. They are not executed as part of the program.

```python
# This is a single-line comment
print("Hello, World!")  # This prints a greeting

'''
This is a multi-line comment
or docstring (can be used to describe a function)
'''

"""
This is another form of multi-line comment
or docstring
"""

# Example of a function with a docstring
def add(a, b):
```

```
    """
    This function takes two numbers and returns their sum.
    """
    return a + b
```

```
OUTPUT:
Hello, World!
```

This code contains different types of Python comments, namely,

- **Single-line comment:** Prefixed with a `#` symbol, used for short explanations or notes.
- **Multi-line comment:** Enclosed in triple quotes (`'''...'''`) or (`"""..."""`), used for longer descriptions or docstrings.

3. Multi-line Statements

Sometimes, a single statement in Python is too long to fit on one line. To break it up, you can use a backslash (\) or parentheses.

```
# Using backslash
total = 1 + 2 + 3 + \
        4 + 5 + 6 + \
        7 + 8 + 9
print(total)

# Using parentheses
total = (1 + 2 + 3 +
         4 + 5 + 6 +
         7 + 8 + 9)
print(total)
```

```
OUTPUT:
45
45
```

This code shows two methods to write multi-line statements, namely,

- **Backslash:** Use a backslash (\) to indicate that the statement continues on the next line.
- **Parentheses:** Use parentheses to enclose the expression, allowing it to span multiple lines.

4. Indentation

Indentation in Python is used to define the scope of loops, functions, and conditional statements.

```
# Correct indentation
if True:
    print("This is true")
    if 5 > 2:
        print("5 is greater than 2")

# Incorrect indentation
if True:
print("This is true")
if 5 > 2:
print("5 is greater than 2")
```

```
OUTPUT (correct indentation):
This is true
5 is greater than 2

OUTPUT (incorrect indentation):
IndentationError: expected an indented block
```

This code demonstrates the importance of correct indentation. Incorrect indentation results in an `IndentationError`.

B. Datatypes and Variables

1. *Elementary datatypes*

Python supports various data types that are used to define the type of a variable. Some of the common data types are integers, floating-point numbers, strings, lists, dictionaries, and complex numbers.

```python
# Integer
a = 10
print(a, type(a))

# Floating-point number
b = 10.5
print(b, type(b))
```

```
OUTPUT:
10 <class 'int'>
10.5 <class 'float'>
```

In the above code, we define an integer variable `a` and a floating-point variable `b`. We print their values and types to see the output.

```python
# String
c = "Hello, World!"
print(c, type(c))

# List
d = [1, 2, 3, 4, 5]
print(d, type(d))
```

```
OUTPUT:
Hello, World! <class 'str'>
[1, 2, 3, 4, 5] <class 'list'>
```

In this code, we define a string variable `c` and a list variable `d`. We print their values and types to see the output.

```python
# Dictionary
e = {'name': 'Alice', 'age': 25, 'city': 'New York'}
print(e, type(e))
```

```
OUTPUT:
{'name': 'Alice', 'age': 25, 'city': 'New York'} <class 'dict'>
```

In this code, we define a dictionary variable `e`. We print its value and type to see the output.

```
# Complex Number
f = 3 + 4j
print(f, type(f))
```

```
OUTPUT:
(3+4j) <class 'complex'>
```

In this code, we define a complex number variable `f`. We print its value and type to see the output.
The above codes demonstrate the use of different Python data types, namely,

- **Integer:** Represents whole numbers (e.g., `10`).
- **Float:** Represents real numbers with a decimal point (e.g., `10.5`).
- **String:** Represents text enclosed in quotes (e.g., `"Hello, World!"`).
- **List:** Represents a collection of items enclosed in square brackets (e.g., `[1, 2, 3, 4, 5]`).
- **Dictionary:** Represents a collection of key-value pairs (e.g., `'name': 'Alice', 'age': 25, 'city': 'New York'`).
- **Complex Number:** Represents a complex number with a real and imaginary part (e.g., `3 + 4j`).

2. Variable Naming Conventions

```
# Camel Case
myVariableName = "Camel Case"

# Pascal Case
MyVariableName = "Pascal Case"

# Snake Case
my_variable_name = "Snake Case"
```

- Camel Case: Starts with a lowercase letter, each subsequent word starts with an uppercase letter (e.g., `myVariableName`).
- Pascal Case: Similar to Camel Case but starts with an uppercase letter (e.g., `MyVariableName`).
- Snake Case: Uses underscores to separate words, all in lowercase (e.g., `my_variable_name`).

3. List

List is a very useful object for collecting data. One of the most common scenarios is that splitting a string returns a list.

```python
# Define a DNA sequence string
seq = 'AATCCGCTACGCTGAATCCGACTACA'

# Split the sequence by 'A', resulting in a list of substrings
sub_seqs = seq.split('A')

# Check the type of sub_seqs (should be list)
print(type(sub_seqs))  # Output: <class 'list'>

# Create a list of integers
list_A = [1, 2, 3]

# Check the type of list_A (should be list)
print(type(list_A))  # Output: <class 'list'>

# Check the type of a slice of list_A (should be list)
print(type(list_A[0:1]))  # Output: <class 'list'>

# Check the type of the first element of list_A (should be int)
print(type(list_A[0]))  # Output: <class 'int'>

# Create a list with mixed types
list_B = [1, 'one']  # Contains an integer and a string

# Check the types of the elements in list_B
print(type(list_B[0]), type(list_B[1]))  # Output: <class 'int'> <class 'str'>

# Check the length of list_B
print(len(list_B))  # Output: 2
```

```
OUTPUT:
<class 'list'>
<class 'list'>
<class 'list'>
<class 'int'>
<class 'int'> <class 'str'>
2
```

---

4. Tuples

Tuples are similar to lists but are immutable, meaning they cannot be changed after their creation.

```python
# Creating a tuple
tuple_A = (1, 2, 3)
print(tuple_A, type(tuple_A))

# Accessing elements in a tuple
print(tuple_A[0])

# Tuples can contain mixed types
tuple_B = (1, "one", [1, 2, 3])
print(tuple_B)
```

```
# Trying to modify a tuple (will result in an error)
# tuple_A[0] = 10  # Uncommenting this line will raise a TypeError
```

```
OUTPUT:
(1, 2, 3) <class 'tuple'>
1
(1, 'one', [1, 2, 3])
```

This code demonstrates the creation and use of tuples in Python. Attempting to modify a tuple will result in a `TypeError`.

5. Sets

Sets are unordered collections of unique elements.

```
# Creating a set
set_A = {1, 2, 3, 4, 5}
print(set_A, type(set_A))

# Adding elements to a set
set_A.add(6)
print(set_A)

# Removing elements from a set
set_A.remove(3)
print(set_A)

# Sets automatically remove duplicates
set_B = {1, 2, 2, 3, 4, 4, 5}
print(set_B)

# Union of sets
set_C = set_A.union({7, 8})
print(set_C)

# Intersection of sets
set_D = set_A.intersection({2, 4, 6, 8})
print(set_D)

# Difference of sets
set_E = set_A.difference({1, 2})
print(set_E)

# Symmetric difference of sets
set_F = set_A.symmetric_difference({3, 5, 7})
print(set_F)
```

```
OUTPUT:
{1, 2, 3, 4, 5} <class 'set'>
{1, 2, 3, 4, 5, 6}
{1, 2, 4, 5, 6}
{1, 2, 3, 4, 5}
{1, 2, 3, 4, 5, 6, 7, 8}
{2, 4, 6}
{3, 4, 5, 6}
{1, 2, 4, 6, 7}
```

This code demonstrates the creation and manipulation of sets in Python, including union, intersection, difference, and symmetric difference operations.

6. Dictionaries

Dictionaries are collections of key-value pairs, where each key is unique.

```python
# Creating a dictionary
dict_A = {'name': 'Bruce', 'age': 25, 'city': 'Gotham city'}
print(dict_A, type(dict_A))

# Accessing values by key
print(dict_A['name'])
print(dict_A['age'])

# Adding a new key-value pair
dict_A['profession'] = 'None'
print(dict_A)

# Updating an existing key-value pair
dict_A['age'] = 26
print(dict_A)

# Removing a key-value pair
del dict_A['city']
print(dict_A)

# Checking if a key exists
print('name' in dict_A)
print('city' in dict_A)

# Iterating over keys and values
for key, value in dict_A.items():
    print(f"{key}: {value}")
```

```
OUTPUT:
{'name': 'Bruce', 'age': 25, 'city': 'Gotham city'} <class 'dict'>
Alice
25
{'name': 'Bruce', 'age': 25, 'city': 'Gotham city', 'profession': 'None'}
{'name': 'Bruce', 'age': 25, 'city': 'Gotham city', 'profession': 'None'}
{'name': 'Bruce', 'age': 26, 'profession': 'None'}
True
False
name: Bruce
age: 25
profession: None
```

This code demonstrates the creation and manipulation of dictionaries in Python, including accessing, adding, updating, removing, checking for keys, and iterating over key-value pairs.

C. Operators

1. *Arithmetic Operators*

Arithmetic operators in Python are used to perform mathematical operations such as addition, subtraction, multiplication, division, and more.

```python
# Addition
a = 10
b = 5
result = a + b
print("Addition:", result)

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

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

```
OUTPUT:
Addition: 15
Subtraction: 5
Multiplication: 50
```

In the above code, we perform addition, subtraction, and multiplication using the variables `a` and `b`. We print the results to see the output.

```python
# Division
#ToDo Floor division
result = a / b
print("Division:", result)

# Modulus
result = a % b
print("Modulus:", result)

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

```
OUTPUT:
Division: 2.0
Modulus: 0
Exponentiation: 100000
```

In this code, we perform division, modulus, and exponentiation using the variables `a` and `b`. We print the results to see the output.

The above codes demonstrate the use of different arithmetic operators in Python, namely,

- **Addition:** Adds two operands (e.g., `a + b`).
- **Subtraction:** Subtracts the second operand from the first (e.g., `a - b`).
- **Multiplication:** Multiplies two operands (e.g., `a * b`).

- **Division:** Divides the first operand by the second (e.g., `a / b`).
- **Modulus:** Returns the remainder when the first operand is divided by the second (e.g., `a % b`).
- **Exponentiation:** Raises the first operand to the power of the second (e.g., `a ** b`).

2. Comparison Operators

Comparison operators in Python are used to compare values. They return either `True` or `False` based on the comparison.

```
# Equal to
a = 10
b = 5
print("Equal to:", a == b)

# Not equal to
print("Not equal to:", a != b)

# Greater than
print("Greater than:", a > b)
```

```
OUTPUT:
Equal to: False
Not equal to: True
Greater than: True
```

In the above code, we use the equal to, not equal to, and greater than operators to compare the variables `a` and `b`. We print the results to see the output.

```
# Less than
print("Less than:", a < b)

# Greater than or equal to
print("Greater than or equal to:", a >= b)

# Less than or equal to
print("Less than or equal to:", a <= b)
```

```
OUTPUT:
Less than: False
Greater than or equal to: True
Less than or equal to: False
```

In this code, we use the less than, greater than or equal to, and less than or equal to operators to compare the variables `a` and `b`. We print the results to see the output.

The above codes demonstrate the use of different comparison operators in Python, namely,

- **Equal to:** Checks if two operands are equal (e.g., `a == b`).
- **Not equal to:** Checks if two operands are not equal (e.g., `a != b`).
- **Greater than:** Checks if the first operand is greater than the second (e.g., `a > b`).
- **Less than:** Checks if the first operand is less than the second (e.g., `a < b`).
- **Greater than or equal to:** Checks if the first operand is greater than or equal to the second (e.g., `a >= b`).
- **Less than or equal to:** Checks if the first operand is less than or equal to the second (e.g., `a <= b`).

3. *Logical Operators*

Logical operators in Python are used to combine conditional statements. They include `and`, `or`, and `not`.

```
# Logical AND
a = True
b = False
print("Logical AND:", a and b)

# Logical OR
print("Logical OR:", a or b)

# Logical NOT
print("Logical NOT a:", not a)
print("Logical NOT b:", not b)
```

```
OUTPUT:
Logical AND: False
Logical OR: True
Logical NOT a: False
Logical NOT b: True
```

In this code, we use logical operators to combine the boolean values `a` and `b`. We print the results to see the output. The above codes demonstrate the use of different logical operators in Python, namely,

- **Logical AND:** Returns `True` if both operands are true (e.g., `a and b`).
- **Logical OR:** Returns `True` if at least one operand is true (e.g., `a or b`).
- **Logical NOT:** Returns `True` if the operand is false (e.g., `not a`).

4. *Bitwise Operators*

Bitwise operators in Python are used to perform bit-level operations on integers. They include `&` (AND), `|` (OR), `^` (XOR), `~` (NOT), `<<` (left shift), and `>>` (right shift).

```
# Bitwise AND
a = 10  # 1010 in binary
b = 4   # 0100 in binary
result = a & b
print("Bitwise AND:", result)

# Bitwise OR
result = a | b
print("Bitwise OR:", result)

# Bitwise XOR
result = a ^ b
print("Bitwise XOR:", result)

#ToDo
# Bitwise NOT
result = ~a
print("Bitwise NOT:", result)
```

```
OUTPUT:
Bitwise AND: 0
Bitwise OR: 14
Bitwise XOR: 14
Bitwise NOT: -11
```

In this code, we use bitwise operators to perform operations on the binary representations of `a` and `b`. We print the results to see the output.

```
# Bitwise left shift
result = a << 1
print("Bitwise left shift:", result)

# Bitwise right shift
result = a >> 1
print("Bitwise right shift:", result)
```

```
OUTPUT:
Bitwise left shift: 20
Bitwise right shift: 5
```

In this code, we use bitwise shift operators to shift the binary representation of `a` to the left and right. We print the results to see the output.

The above codes demonstrate the use of different bitwise operators in Python, namely,

- **Bitwise AND:** Performs a logical AND operation on each bit of the binary representations of the operands (e.g., `a & b`).
- **Bitwise OR:** Performs a logical OR operation on each bit of the binary representations of the operands (e.g., `a | b`).
- **Bitwise XOR:** Performs a logical XOR operation on each bit of the binary representations of the operands (e.g., `a ^ b`).
- **Bitwise NOT:** Inverts each bit of the binary representation of the operand (e.g., `~a`).
- **Left Shift:** Shifts the bits of the binary representation of the operand to the left by a specified number of positions (e.g., `a << 1`).
- **Right Shift:** Shifts the bits of the binary representation of the operand to the right by a specified number of positions (e.g., `a >> 1`).

5. Assignment Operators

Assignment operators in Python are used to assign values to variables. They include `=`, `+=`, `-=`, `*=`, `/=`, `%=`, `**=`, `&=`, `|=`, `^=`, `>>=`, and `<<=`.

```
# Simple assignment
a = 10
print("Assignment:", a)

# Add and assign
a += 5
print("Add and assign:", a)

# Subtract and assign
```

```
a -= 3
print("Subtract and assign:", a)

# Multiply and assign
a *= 2
print("Multiply and assign:", a)

# Divide and assign
a /= 4
print("Divide and assign:", a)
```

```
OUTPUT:
Assignment: 10
Add and assign: 15
Subtract and assign: 12
Multiply and assign: 24
Divide and assign: 6.0
```

In this code, we use different assignment operators to perform arithmetic operations and assign the result to the variable a. We print the results to see the output.

```
# Modulus and assign
a %= 4
print("Modulus and assign:", a)

#ToDo
# Exponentiation and assign
a **= 3
print("Exponentiation and assign:", a)

# Bitwise AND and assign
a &= 2
print("Bitwise AND and assign:", a)

# Bitwise OR and assign
a |= 1
print("Bitwise OR and assign:", a)

# Bitwise XOR and assign
a ^= 3
print("Bitwise XOR and assign:", a)

# Bitwise left shift and assign
a <<= 2
print("Bitwise left shift and assign:", a)

# Bitwise right shift and assign
a >>= 1
print("Bitwise right shift and assign:", a)
```

```
OUTPUT:
Modulus and assign: 2.0
Exponentiation and assign: 8.0
Bitwise AND and assign: 0
Bitwise OR and assign: 1
Bitwise XOR and assign: 2
Bitwise left shift and assign: 8
Bitwise right shift and assign: 4
```

In this code, we use different assignment operators to perform bitwise operations and assign the result to the variable `a`. We print the results to see the output.

The above codes demonstrate the use of different assignment operators in Python, namely,

- **Assignment:** Assigns a value to a variable (e.g., `a = 10`).
- **Add and assign:** Adds a value to the variable and assigns the result (e.g., `a += 5`).
- **Subtract and assign:** Subtracts a value from the variable and assigns the result (e.g., `a -= 3`).
- **Multiply and assign:** Multiplies the variable by a value and assigns the result (e.g., `a *= 2`).
- **Divide and assign:** Divides the variable by a value and assigns the result (e.g., `a /= 4`).
- **Modulus and assign:** Computes the modulus of the variable and a value, then assigns the result (e.g., `a %= 4`).
- **Exponentiation and assign:** Raises the variable to the power of a value and assigns the result (e.g., `a **= 3`).
- **Bitwise AND and assign:** Performs a bitwise AND on the variable and a value, then assigns the result (e.g., `a &= 2`).
- **Bitwise OR and assign:** Performs a bitwise OR on the variable and a value, then assigns the result (e.g., `a |= 1`).
- **Bitwise XOR and assign:** Performs a bitwise XOR on the variable and a value, then assigns the result (e.g., `a ^= 3`).
- **Bitwise left shift and assign:** Shifts the bits of the variable to the left by a specified number of positions and assigns the result (e.g., `a <<= 2`).
- **Bitwise right shift and assign:** Shifts the bits of the variable to the right by a specified number of positions and assigns the result (e.g., `a >>= 1`).

D. Control Flow

Control flow in Python allows us to dictate the order in which the statements in a program are executed. Key control flow tools include conditional statements and loops.

1. If, Else, and Elif Statements

If statements are used to execute a block of code only if a specified condition is true. The `else` and `elif` (else if) statements provide additional conditional checks and alternative blocks of code.

```
# If statement
a = 10
if a > 5:
    print("a is greater than 5")

# If-Else statement
b = 3
if b > 5:
    print("b is greater than 5")
else:
    print("b is not greater than 5")

# If-Elif-Else statement
```

```
c = 7
if c > 10:
    print("c is greater than 10")
elif c == 7:
    print("c is equal to 7")
else:
    print("c is less than 10 and not equal to 7")
```

```
OUTPUT:
a is greater than 5
b is not greater than 5
c is equal to 7
```

The above code demonstrates the use of `if`, `else`, and `elif` statements to control the flow of the program based on the conditions specified.

2. For Loops

For loops in Python are used to iterate over a sequence (e.g., a list, tuple, dictionary, set, or string).

```
# For loop over a list
numbers = [1, 2, 3, 4, 5]
for num in numbers:
    print(num)

# For loop with range
for i in range(5):
    print(i)
```

```
OUTPUT:
1
2
3
4
5
0
1
2
3
4
```

In the above code, the first for loop iterates over a list of numbers and prints each number. The second for loop iterates over a range of numbers from 0 to 4.

3. While Loops

While loops in Python are used to execute a block of code as long as a specified condition is true.

```
# While loop
count = 0
while count < 5:
    print(count)
    count += 1
```

```
OUTPUT:
0
1
2
3
4
```

In the above code, the while loop continues to execute as long as the variable `count` is less than 5. The variable `count` is incremented by 1 in each iteration.

4. Break and Continue Statements

The `break` statement is used to exit a loop prematurely, while the `continue` statement skips the rest of the code inside the loop for the current iteration and moves to the next iteration.

```
# Break statement
for i in range(10):
    if i == 5:
        break
    print(i)

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

```
OUTPUT:
0
1
2
3
4
1
3
5
7
9
```

In the above code, the first for loop exits when `i` equals 5 due to the `break` statement. The second for loop skips printing the even numbers due to the `continue` statement.

5. Chained Comparison

Python allows chained comparisons, which can be a more concise way of writing multiple comparisons.

```
# Chained comparison
x = 5
print(1 < x < 10)
print(10 < x < 20)
print(x < 10 < x*10 < 100)
print(10 > x <= 9)
print(5 == x > 4)
```

```
OUTPUT:
True
False
True
True
True
```

In the above code, chained comparisons are used to perform multiple comparisons in a single statement.

6. *Boolean Values*

Boolean values `True` and `False` are used to control the flow in conditional statements.

```python
# Boolean values
is_active = True
if is_active:
    print("The system is active")

is_logged_in = False
if not is_logged_in:
    print("User is not logged in")
```

```
OUTPUT:
The system is active
User is not logged in
```

In the above code, boolean values are used to control the execution of the conditional statements.

7. *Integer Comparison*

Integer comparison is straightforward in Python using comparison operators.

```python
# Integer comparison
a = 10
b = 20

if a < b:
    print("a is less than b")
else:
    print("a is not less than b")
```

```
OUTPUT:
a is less than b
```

In the above code, integers `a` and `b` are compared using the less than operator.

8. String Comparison

String comparison in Python can be performed using comparison operators, which compare the strings lexicographically based on their Unicode values.

```
# String comparison
string1 = "Bruce"
string2 = "Clark"

if string1 < string2:
    print("Bruce comes before Clark in alphabetical order")
else:
    print("Bruce does not come before Clark in alphabetical order")
```

```
OUTPUT:
Bruce comes before Gotham in alphabetical order
```

In the above code, strings `string1` and `string2` are compared lexicographically using the less than operator.

9. Case-Insensitive Comparison

To perform case-insensitive comparisons, convert the strings to the same case (upper or lower) before comparing.

```
# Case-insensitive comparison
name1 = "Bruce"
name2 = "bruce"

if name1.lower() == name2.lower():
    print("The names are equal (case-insensitive)")
else:
    print("The names are not equal (case-insensitive)")
```

```
OUTPUT:
The names are equal (case-insensitive)
```

In the above code, the strings `name1` and `name2` are converted to lowercase before comparing to ensure case-insensitive comparison.

E. Reading and Writing in Python

1. Reading and Writing Text Files

Reading from and writing to text files is a common operation in Python. Here is an example of how to perform these operations.

```
# Writing to a text file
with open('example.txt', 'w') as file:
    file.write("Hello, World!\n")
    file.write("This is a sample text file.\n")

# Reading from a text file
```

```
with open('example.txt', 'r') as file:
    content = file.read()
    print(content)
```

```
OUTPUT:
Hello, World!
This is a sample text file.
```

In this code:

- **Writing:** We use `open()` with the mode `'w'` to write to a file.
- **Reading:** We use `open()` with the mode `'r'` to read from a file.

2. Reading CSV Files with Plain Python

CSV (Comma Separated Values) files are commonly used for storing tabular data. Here is an example of how to read a CSV file using plain Python.

```
import csv

# Create a csv file
with open('example.csv', 'w') as file:
    file.write("name, age, city\n")
    file.write("Bruce, 25, Gotham city\n")
    file.write("Clark, 29, Metropolis\n")
    file.write("Louis, 25, Metropolis\n")

# Reading a CSV file
with open('example.csv', newline='') as csvfile:
    reader = csv.reader(csvfile)
    for row in reader:
        print(', '.join(row))
```

```
OUTPUT:
name,  age,  city
Bruce,  25,  Gotham city
Clark,  29,  Metropolis
Louis,  25,  Metropolis
```

In this code:

- We use the `csv.reader()` function to read the CSV file.
- The `for` loop iterates through each row of the CSV file.
- We print each row by joining the elements with a comma.

3. Reading CSV Files with Pandas

Pandas is a powerful library for data manipulation and analysis. It provides a more convenient way to read and write CSV files.

```
import pandas as pd

# Reading a CSV file using Pandas
df = pd.read_csv('example.csv')
print(df)
```

```
OUTPUT:
    name   age          city
0  Bruce    25   Gotham city
1  Clark    29    Metropolis
2  Louis    25    Metropolis
```

In this code:

- We use the `pd.read_csv()` function from the Pandas library to read the CSV file.
- The data is loaded into a DataFrame, which provides powerful data manipulation capabilities.
- We print the DataFrame to display the contents of the CSV file.

Suppose the title line (or the header) is not available in the csv file, you can read it by setting `header=None`

```
import csv
import pandas as pd

# Create a csv file
with open('example.csv', 'w') as file:
    file.write("Bruce, 25, Gotham city\n")
    file.write("Clark, 29, Metropolis\n")
    file.write("Louis, 25, Metropolis\n")

df = pd.read_csv('example.csv',header=None)
print(df)
```

```
OUTPUT:
       0   1             2
0  Bruce  25   Gotham city
1  Clark  29    Metropolis
2  Louis  25    Metropolis
```

F. Miscellaneous Topics

1. *String manipulations: Slicing vs. Splitting*

Learning how to manipulate strings is very useful. Think of a situation when your Ph.D. advisor asks you to download a file with a lot of texts, count how many times a string like 'compound' occurs in it, and find out how many unique compounds are mentioned in that file.

```
# Assign a string to the variable 'mystr'
mystr = 'May the force be with you!'

# Access the first character (index 0) of the string
print(mystr[0])   # Output: M
```

```python
# Access the substring from index 0 to 1 (not inclusive)
print(mystr[0:1])  # Output: M

# Access the substring from index 0 to 5 (not inclusive), Slicing
print(mystr[0:5])  # Output: May t

# Assign a comma-separated string to the variable 'mystr'
mystr = '1,2,3,4'

# Split the string by commas, resulting in a list
print(mystr.split(','))  # Output: ['1', '2', '3', '4']

# Access the first two entries of the list
print(mystr.split(',')[0:2]) # Output: ['1', '2']

# Assign a DNA sequence string to the variable 'seq'
seq = 'AATCCGCTACGCTGAATCCGACTACA'

# Split the sequence by 'A', resulting in a list of substrings
sub_seqs = seq.split('A')  # Here 'A' is the marker or delimiter

# Print the list of substrings
print(sub_seqs)  # Output: ['', '', 'TCCGCT', 'CGCTG', '', 'TCCG', 'CT', 'C']

# Print the type of the resulting list
print(type(sub_seqs))  # Output: <class 'list'>

# Calculate the number of substrings in the list
N = len(sub_seqs)
print(N)  # Output: 9

# Iterate through the list and print the index and substring
for i in range(0, N):
    print(i, sub_seqs[i])  # NOTE: 4 empty spaces for indentation

# Combine the first three substrings
print(sub_seqs[0] + sub_seqs[1] + sub_seqs[2])  # Output: TCCGCT
```

```
<OUTPUT>
M
M
May t
['1', '2', '3', '4']
['1', '2']
['', '', 'TCCGCT', 'CGCTG', '', 'TCCG', 'CT', 'C', '']
<class 'list'>
9
0
1
2 TCCGCT
3 CGCTG
4
5 TCCG
6 CT
7 C
8
TCCGCT
```

2. *Shallow Copy vs. Deep Copy*

Understanding the difference between shallow copy and deep copy is crucial when working with mutable objects like lists in Python. A shallow copy creates a new reference to the same object, meaning changes to one will affect the other. A deep copy creates a new object with the same contents, but changes to one will not affect the other.

```
# Create a list
original_list = [1, 2, 3, 4]

# Perform a shallow copy (alias)
shallow_copied_list = original_list

# Modify the shallow copied list
shallow_copied_list[0] = 'A'

# Print the lists to observe the effects
print("Original List after shallow copy modification:", original_list)
# Output: ['A', 2, 3, 4]
print("Shallow Copied List:", shallow_copied_list)
# Output: ['A', 2, 3, 4]

# Create a list again to reset original_list
original_list = [1, 2, 3, 4]

# Perform a deep copy
deep_copied_list = original_list.copy()

# Modify the deep copied list
deep_copied_list[1] = 'B'

# Print the lists to observe the effects
print("Original List after deep copy modification:", original_list)
# Output: [1, 2, 3, 4]
print("Deep Copied List:", deep_copied_list)
# Output: [1, 'B', 3, 4]
```

```
<OUTPUT>
Original List after shallow copy modification: ['A', 2, 3, 4]
Shallow Copied List: ['A', 2, 3, 4]
Original List after deep copy modification: [1, 2, 3, 4]
Deep Copied List: [1, 'B', 3, 4]
```

---

3. *String Concatenation vs. List Appending and Concatenation*

Understanding how to concatenate strings and append elements to lists, as well as concatenate two lists, is fundamental in Python. String concatenation combines two strings into one, while list appending adds an element to the end of a list, and list concatenation combines two lists into one.

```
# String Concatenation

# Create two strings
```

```
string1 = "Hello"
string2 = "World"

# Concatenate strings
concatenated_string = string1 + " " + string2

# Print the concatenated string
print("Concatenated String:", concatenated_string)  # Output: Hello World

# List Appending

# Create a list
my_list = [1, 2, 3, 4]

# Append an element to the list
my_list.append(5)

# Append another element to the list
my_list.append("six")

# Print the list after appending elements
print("List after appending elements:", my_list)
# Output: [1, 2, 3, 4, 5, 'six']

# List Concatenation

# Create two lists
list1 = [1, 2, 3]
list2 = [4, 5, 6]

# Concatenate lists
concatenated_list = list1 + list2

# Print the concatenated list
print("Concatenated List:", concatenated_list)
# Output: [1, 2, 3, 4, 5, 6]
```

```
<OUTPUT>
Concatenated String: Hello World
List after appending elements: [1, 2, 3, 4, 5, 'six']
Concatenated List: [1, 2, 3, 4, 5, 6]
```

---

4. Multiple Assignment in One Line

Python allows the assignment of multiple variables in a single line, which can make the code more concise and readable.

```
# Multiple assignment
a, b, c = 5, 10, 15

print(f"a: {a}, b: {b}, c: {c}")
```

```
OUTPUT:
a: 5, b: 10, c: 15
```

In the above code, the variables `a`, `b`, and `c` are assigned the values 5, 10, and 15 respectively in a single line.
Multiple Assignment:

- Assign multiple variables in one line (e.g., `a, b, c = 5, 10, 15`).

5. Array Assignment

Python's numpy library provides powerful tools for array manipulation, including the ability to assign new values to an array.

```python
# Array assignment using numpy
import numpy as np

# Create an array
array = np.array([1, 2, 3, 4])

# Assign new values
array[:] = [5, 6, 7, 8]

print(f"Updated Array: {array}")
```

```
OUTPUT:
Updated Array: [5 6 7 8]
```

In the above code, a numpy array is created and then all its values are updated using slicing.
Array Assignment:

- Use numpy for array manipulation.
- Assign new values to an array using slicing (e.g., `array[:] = [5, 6, 7, 8]`).

6. Print Repeated Characters

Python allows for easy repetition of characters using string multiplication, which can be useful for formatting output.

```python
# Print repeated characters using format method
a = 'Repeated characters'
print('{0}\nAnswer:{1}\n{0}'.format('-'*11, a))
```

```
OUTPUT:
-----------
Answer:Repeated characters
-----------
```

In the above code, the hyphen character is repeated 11 times and used to format the output.
Print Repeated Characters:

- Use string formatting to repeat characters and include them in output (e.g., `'-'*11`).

7. *Formatted Strings*

Formatted strings, also known as f-strings, provide a way to embed expressions inside string literals, using curly braces .

```
# Formatted strings
string_var = 'Hello'
integer_var = 42
float_var = 3.14159

# Print string, integer, and float
print(f"String: {string_var}")
print(f"Integer: {integer_var}")
print(f"Float: {float_var:.2f}")

# Right aligned text
print(f"Right Aligned: {string_var:>15}")

# Trailing zeros in float
print(f"Float with Trailing Zeros: {float_var:.5f}")
```

```
OUTPUT:
String: Hello
Integer: 42
Float: 3.14
Right Aligned:           Hello
Float with Trailing Zeros: 3.14159
```

In the above code, various uses of formatted strings are demonstrated, including printing variables, formatting floats, and aligning text.

Formatted Strings:

- **String:** Directly print string variables.
- **Integer:** Print integers (e.g., `integer_var`).
- **Float:** Format floats to a specified number of decimal places (e.g., `float_var:.2f`).
- **Right Aligned:** Align text to the right (e.g., `string_var:>15`).
- **Trailing Zeros:** Control the number of decimal places in float (e.g., `float_var:.5f`).

G. Functions and Modules

1. *Python Functions*

Functions in Python are blocks of reusable code that perform a specific task. Functions help to organize code into logical units and make it more readable and maintainable. They can also accept parameters and return values.

```
# Defining a Basic Function

def greet(name):
    """
    Print a greeting message.

    Parameters:
```

```
    name (str): The name of the person to greet.
    """
    print(f"Hello, {name}!")
```

In this code block, a basic function named `greet` is defined. The function takes one parameter, `name`, and prints a greeting message. The docstring within triple quotes describes the purpose of the function and its parameter. Docstrings provide documentation for the function and can be accessed using `help(greet)` or `greet.__doc__`.

```
# Calling the Function

greet("Alice")
```

```
Hello, Alice!
```

This code block demonstrates how to call the `greet` function defined earlier. By passing the argument `"Alice"` to the function, it prints the greeting message `"Hello, Alice!"` to the console.

```
# Function with Return Value

def add(a, b):
    """
    Add two numbers and return the result.

    Parameters:
    a (int or float): The first number.
    b (int or float): The second number.

    Returns:
    int or float: The sum of the two numbers.
    """
    return a + b
```

In this code block, a function named `add` is defined, which takes two parameters `a` and `b`, and returns their sum. The docstring explains the parameters and the return value of the function. This function can be used to perform addition operations with integers or floating-point numbers.

```
# Using the Return Value

result = add(10, 5)
print(result)
```

```
15
```

This code block demonstrates how to use the `add` function. By calling `add(10, 5)`, the function returns the result `15`, which is then printed to the console.

```python
# Function with Default Parameters

def multiply(a, b=1):
    """
    Multiply two numbers and return the result. The second number has a default value of 1.

    Parameters:
    a (int or float): The first number.
    b (int or float, optional): The second number. Defaults to 1.

    Returns:
    int or float: The product of the two numbers.
    """
    return a * b
```

In this code block, the `multiply` function is defined with a default value for the second parameter. The docstring describes this optional parameter and its default value. This allows the function to be called with either one or two arguments.

```python
# Using the Default Parameter

print(multiply(5))    # Using default value for b
print(multiply(5, 3)) # Providing both parameters
```

```
5
15
```

This code block shows how to use the `multiply` function with and without the second parameter. When only one argument is provided, the default value of `b` is used, resulting in a product of `5`. When both arguments are provided, the product of `5` and `3` is computed, resulting in `15`.

```python
# Function with Keyword Arguments

def greet(name, greeting="Hello"):
    """
    Print a custom greeting message.

    Parameters:
    name (str): The name of the person to greet.
    greeting (str, optional): The greeting message. Defaults to "Hello".
    """
    print(f"{greeting}, {name}!")

# Calling function with keyword arguments
greet(name="Bob", greeting="Hi")
greet(name="Alice")  # Uses default greeting
```

```
Hi, Bob!
Hello, Alice!
```

In this code block, the `greet` function is defined with a default value for the `greeting` parameter. This allows the function to be called with or without specifying the `greeting` argument. The function demonstrates keyword arguments where the greeting can be customized or defaults to "Hello".

2. Python Modules

Modules in Python are files containing Python code. They can define functions, classes, and variables that can be imported and used in other Python scripts. Modules help to organize and reuse code efficiently.

```
# Importing the os module
import os

# Get the current working directory
current_directory = os.getcwd()
print(f"Current Directory: {current_directory}")

# List files and directories in the current directory
files_and_dirs = os.listdir('.')
print(f"Files and Directories: {files_and_dirs}")
```

```
Current Directory: /Users/rr/repos/NumericalMethods/notebooks
Files and Directories: ['NM01_ComputerScienceConcepts.ipynb', '.ipynb_checkpoints', 'NM02.ipynb']
```

In this code block, the `os` module is used to interact with the operating system. The `getcwd()` function retrieves the current working directory, and `listdir()` lists all files and directories in it.

```
# Importing the sys module
import sys

# Get the Python version
python_version = sys.version
print(f"Python Version: {python_version}")
```

```
Python Version: 3.11.6 (main, Oct  2 2023, 20:46:14) [Clang 14.0.3 (clang-1403.0.22.14.1)]
```

This code block shows how to use the `sys` module to access system-specific parameters. The `version` attribute provides the Python version.

```
# Importing the math module
import math

# Calculate the square root
sqrt_value = math.sqrt(25)
print(f"Square Root of 25: {sqrt_value}")

# Calculate the value of pi
pi_value = math.pi
print(f"Value of Pi: {pi_value}")
```

```
Square Root of 25: 5.0
Value of Pi: 3.141592653589793
```

In this code block, the `math` module provides mathematical functions. `sqrt()` calculates the square root of a number, and `pi` gives the value of π.

```
# Importing the cmath module
import cmath

# Calculate the square root of a negative number
sqrt_neg = cmath.sqrt(-16)
print(f"Square Root of -16: {sqrt_neg}")

# Calculate the value of e^(i*pi)
exp_value = cmath.exp(cmath.pi * 1j)
print(f"Value of e^(i*pi): {exp_value}")
```

```
Square Root of -16: 4j
Value of e^(i*pi): (-1+1.2246467991473532e-16j)
```

The `cmath` module handles complex numbers. `sqrt()` can compute the square root of negative numbers, and `exp()` calculates the exponential function for complex arguments.

```
# Importing the numpy module
import numpy as np

# Create an array
array = np.array([1, 2, 3, 4])

# Perform element-wise operations
squared_array = np.square(array)
print(f"Array Squared: {squared_array}")

# Calculate the mean of the array
mean_value = np.mean(array)
print(f"Mean of Array: {mean_value}")
```

```
Array Squared: [ 1  4  9 16]
Mean of Array: 2.5
```

The `numpy` module is used for numerical computations with arrays. `array()` creates an array, `square()` performs element-wise squaring, and `mean()` calculates the average value of the array elements.

```
# content of mymodule.py
def say_hello(name):
    """
    Print a hello message to the given name.

    Parameters:
    name (str): The name of the person to greet.
    """
```

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

def add_numbers(a, b):
    """
    Return the sum of two numbers.

    Parameters:
    a (int or float): The first number.
    b (int or float): The second number.

    Returns:
    int or float: The sum of the two numbers.
    """
    return a + b
```

```
# Using the user-defined module

# Import the user-defined module
import mymodule

# Call functions from the module
mymodule.say_hello("Charlie")
result = mymodule.add_numbers(10, 20)
print(f"Sum: {result}")
```

```
Hello, Charlie!
Sum: 30
```

In this code block, a user-defined module named `mymodule` is created with two functions: `say_hello` and `add_numbers`. The module is then imported and its functions are called to demonstrate how to use custom Python modules.

```
# Import sys to modify the path for module searching
import sys

# Add the directory containing the user-defined module to the system path
sys.path.append('/Users/rr/repos/NumericalMethods/notebooks/mymodules')
#ToDo, define algebra.py

# Import functions from the user-defined module 'algebra'
from algebra import square as my_sqr
from algebra import add as my_add

# Use the imported functions
print(f"Square of 5: {my_sqr(5)}")
print(f"Sum of 5 and 7: {my_add(5, 7)}")
```

```
Square of 5: 25
Sum of 5 and 7: 12
```

In this code block, the `sys` module is used to modify the system path to include a specific directory containing user-defined modules. Functions from the `algebra` module in this directory are imported and used.

H. Standard Python Functions and Modules

Python provides a rich standard library and many third-party libraries that extend its functionality. This section covers some essential Python functions and commonly used modules like numpy, scipy, pandas, and matplotlib.

1. *Standard Python Functions*

Python comes with many built-in functions that you can use to perform common tasks.

```
# Standard functions
# Absolute value
print("Absolute value of -5:", abs(-5))

# Maximum and minimum
print("Maximum of 1, 2, 3:", max(1, 2, 3))
print("Minimum of 1, 2, 3:", min(1, 2, 3))

# Sum
numbers = [1, 2, 3, 4, 5]
print("Sum of numbers:", sum(numbers))

# Length of a list
print("Length of numbers list:", len(numbers))

# Round a number
print("Round 3.14159 to 2 decimal places:", round(3.14159, 2))
```

```
OUTPUT:
Absolute value of -5: 5
Maximum of 1, 2, 3: 3
Minimum of 1, 2, 3: 1
Sum of numbers: 15
Length of numbers list: 5
Round 3.14159 to 2 decimal places: 3.14
```

The above code demonstrates the use of some standard Python functions:
Standard Python Functions:

- **abs():** Returns the absolute value of a number (e.g., `abs(-5)`).
- **max():** Returns the largest of the input values (e.g., `max(1, 2, 3)`).
- **min():** Returns the smallest of the input values (e.g., `min(1, 2, 3)`).
- **sum():** Returns the sum of the items in an iterable (e.g., `sum(numbers)`).
- **len():** Returns the length of an object (e.g., `len(numbers)`).
- **round():** Rounds a number to a specified number of decimal places (e.g., `round(3.14159, 2)`).

2. *Numpy*

Numpy is a fundamental package for scientific computing in Python. It provides support for arrays, matrices, and many mathematical functions.

```
import numpy as np

# Create an array
array = np.array([1, 2, 3, 4])

# Array operations
print("Original array:", array)
print("Array + 2:", array + 2)
print("Array * 2:", array * 2)
print("Array squared:", array ** 2)

# Mean and standard deviation
print("Mean of array:", np.mean(array))
print("Standard deviation of array:", np.std(array))
```

```
OUTPUT:
Original array: [1 2 3 4]
Array + 2: [3 4 5 6]
Array * 2: [2 4 6 8]
Array squared: [ 1  4  9 16]
Mean of array: 2.5
Standard deviation of array: 1.118033988749895
```

The above code demonstrates the use of numpy for array creation and manipulation.
Numpy:

- Create arrays (e.g., `np.array([1, 2, 3, 4])`).
- Perform element-wise operations on arrays (e.g., `array + 2`).
- Calculate statistical measures (e.g., `np.mean(array)`, `np.std(array)`).

3. Scipy

Scipy builds on numpy and provides additional functionality for scientific and technical computing, such as optimization, integration, interpolation, eigenvalue problems, and more.

```
from scipy import integrate

# Define a function to integrate
def f(x):
    return x**2

# Integrate function from 0 to 1
result, error = integrate.quad(f, 0, 1)
print("Integral of x^2 from 0 to 1:", result)
```

```
OUTPUT:
Integral of x^2 from 0 to 1: 0.33333333333333337
```

The above code demonstrates the use of scipy to perform integration.
Scipy:

- Perform integration (e.g., `integrate.quad(f, 0, 1)`).
- Provides advanced mathematical functions beyond numpy.

4. *Pandas*

Pandas is a powerful library for data manipulation and analysis. It provides data structures like DataFrame and Series for handling structured data.

```
import pandas as pd

# Create a DataFrame
data = {'Name': ['Bruce', 'Clark', 'Diana'],
        'City': ['Gotham', 'Metropolis', 'Themyscira'],
        'Profession': ['None', 'Reporter', 'Curator']}
df = pd.DataFrame(data)

# Display the DataFrame
print(df)

# Access a column
print("Names:", df['Name'])

# Calculate summary statistics
print("Summary statistics for DataFrame:\n", df.describe(include='all'))
```

```
OUTPUT:
    Name        City Profession
0  Bruce      Gotham       None
1  Clark  Metropolis   Reporter
2  Diana  Themyscira    Curator

Names: 0    Bruce
1    Clark
2    Diana
Name: Name, dtype: object

Summary statistics for DataFrame:
           Name      City Profession
count         3         3          3
unique        3         3          3
top       Bruce     Gotham     Batman
freq          1         1          1
```

The above code demonstrates the use of pandas to create a DataFrame and perform basic operations.
Pandas:

- Create DataFrames (e.g., `pd.DataFrame(data)`).
- Access and manipulate data in DataFrames (e.g., `df['Name']`).
- Calculate summary statistics (e.g., `df.describe(include='all')`).

5. *Matplotlib*

Matplotlib is a comprehensive library for creating static, animated, and interactive visualizations in Python.

```
import matplotlib.pyplot as plt

# Data to plot
x = [1, 2, 3, 4]
y = [1, 4, 9, 16]

# Create a plot
plt.plot(x, y)
plt.xlabel('x-axis')
plt.ylabel('y-axis')
plt.title('Simple Plot')
plt.show()
```

```
OUTPUT:
(Displays a plot with x vs. y)
```

The above code demonstrates the use of matplotlib to create a simple plot.
Matplotlib:

- Create plots (e.g., `plt.plot(x, y)`).
- Label axes and add titles to plots (e.g., `plt.xlabel()`, `plt.ylabel()`, `plt.title()`).
- Display plots (e.g., `plt.show()`).