**INTRODUCTION TO PYTHON**

**WEEK 4 OBJECT AND CLASSES**

**DAY 1 INHERITANCE AND PRIVATE VARIABLES**

**INHERITENCE**

Inheritance is a fundamental concept in object-oriented programming (OOP) that allows a class (subclass or derived class) to inherit the properties and methods of another class (superclass or base class). This promotes code reuse and allows for the creation of a hierarchy of classes.

In Python, you can achieve inheritance by creating a new class that is a modified version of an existing class. The new class is called the subclass or derived class, and the existing class is the superclass or base class. Here's a basic example:

```python

# Superclass/Base class

class Animal:

def \_\_init\_\_(self, name):

self.name = name

def speak(self):

pass # Abstract method, to be overridden by subclasses

# Subclass/Derived class

class Dog(Animal):

def speak(self):

return f"{self.name} says Woof!"

# Subclass/Derived class

class Cat(Animal):

def speak(self):

return f"{self.name} says Meow!"

# Example usage

dog = Dog("Buddy")

cat = Cat("Whiskers")

print(dog.speak()) # Output: Buddy says Woof!

print(cat.speak()) # Output: Whiskers says Meow!

```

In this example, the `Animal` class is the superclass with a basic `speak` method. The `Dog` and `Cat` classes are subclasses that inherit from the `Animal` class. They override the `speak` method to provide their specific implementation.

Key points:

1. \*\*Initialization (`\_\_init\_\_`) Method\*\*: The `\_\_init\_\_` method of the superclass is not automatically called in the subclass. You need to explicitly call it using `super().\_\_init\_\_(...)` in the subclass's `\_\_init\_\_` method if you want to initialize the attributes of the superclass.

```python

class Dog(Animal):

def \_\_init\_\_(self, name, breed):

super().\_\_init\_\_(name)

self.breed = breed

```

2. \*\*Method Overriding\*\*: Subclasses can override methods of the superclass to provide their own implementation.

3. \*\*Access to Superclass Methods\*\*: Subclasses can also access methods of the superclass using `super()`.

4. \*\*Inheriting Attributes and Methods\*\*: Subclasses inherit attributes and methods from the superclass.

5. \*\*Multiple Inheritance\*\*: Python supports multiple inheritance, where a class can inherit from more than one superclass.

```python

class Bird:

def fly(self):

return "I can fly!"

class Sparrow(Animal, Bird):

pass

sparrow = Sparrow("Tweetie")

print(sparrow.speak()) # Output: Tweetie says Meow!

print(sparrow.fly()) # Output: I can fly!

```

In this example, `Sparrow` is a subclass of both `Animal` and `Bird`, inheriting from both classes.

**PRIVATE VARIABLES**

In Python, you can create private variables in a class by prefixing the variable name with double underscores (`\_\_`). This is known as name mangling, and it makes it harder to access the variable from outside the class.

Here's an example:

```python

class MyClass:

def \_\_init\_\_(self):

self.\_\_private\_variable = 42

def get\_private\_variable(self):

return self.\_\_private\_variable

def set\_private\_variable(self, value):

self.\_\_private\_variable = value

# Create an instance of the class

obj = MyClass()

# Access the private variable using a getter method

print(obj.get\_private\_variable()) # Output: 42

# Modify the private variable using a setter method

obj.set\_private\_variable(100)

# Access the modified private variable

print(obj.get\_private\_variable()) # Output: 100

# Attempting to access the private variable directly from outside the class will result in an AttributeError

# Uncommenting the line below will raise an AttributeError

# print(obj.\_\_private\_variable)

```

Even though the double underscore prefix makes it harder to access the variable directly, it doesn't make it truly private or secure. It's more of a convention to indicate that the variable is intended for internal use within the class. If someone really wants to access it, they can still do so, but it's a signal that they should not.

It's important to note that while using a single underscore before a variable (e.g., `\_variable`) is a convention to indicate that it's intended to be private, it doesn't provide any actual protection. It's more of a signal to other developers that the variable is intended for internal use. The double underscore is more about name mangling and making it less accessible from outside the class.

**ITERATORS, GENERATORS, AND MORE**

**ITERATORS AND GENERATORS**

Iterators

In any programming environment, it will always be useful to iterate. By now you have probably noticed that most container objects can be looped over using a for statement:

for element in [1, 2, 3]:

print(element)

for element in (1, 2, 3):

print(element)

for key in {'one':1, 'two':2}:

print(key)

for char in "123":

print(char)

for line in open("myfile.txt"):

print(line, end='')

This style of access is clear, concise, and convenient. The use of iterators pervades and unifies Python. Behind the scenes, the for statement calls iter() on the container object. The function returns an iterator object that defines the method \_\_next\_\_() which accesses elements in the container one at a time. When there are no more elements, \_\_next\_\_() raises a StopIteration exception which tells the for loop to terminate. You can call the \_\_next\_\_() method using the next() built-in function; this example shows how it all works:

>>>

>>> s = 'abc'

>>> it = iter(s)

>>> it

<iterator object at 0x00A1DB50>

>>> next(it)

'a'

>>> next(it)

'b'

>>> next(it)

'c'

>>> next(it)

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

next(it)

StopIteration

Having seen the mechanics behind the iterator protocol, it is easy to add iterator behavior to your classes. Define an \_\_iter\_\_() method which returns an object with a \_\_next\_\_() method. If the class defines \_\_next\_\_(), then \_\_iter\_\_() can just return self:

class Reverse:

"""Iterator for looping over a sequence backwards."""

def \_\_init\_\_(self, data):

self.data = data

self.index = len(data)

def \_\_iter\_\_(self):

return self

def \_\_next\_\_(self):

if self.index == 0:

raise StopIteration

self.index = self.index - 1

return self.data[self.index]

>>>

>>> rev = Reverse('spam')

>>> iter(rev)

<\_\_main\_\_.Reverse object at 0x00A1DB50>

>>> for char in rev:

... print(char)

...

m

a

p

s

**Generators**

Generators are a simple and powerful tool for creating iterators. They are written like regular functions but use the yield statement whenever they want to return data. Each time next() is called on it, the generator resumes where it left off (it remembers all the data values and which statement was last executed). An example shows that generators can be trivially easy to create:

def reverse(data):

for index in range(len(data)-1, -1, -1):

yield data[index]

>>>

>>> for char in reverse('golf'):

... print(char)

...

f

l

o

g

Anything that can be done with generators can also be done with class-based iterators as described in the previous section. What makes generators so compact is that the \_\_iter\_\_() and \_\_next\_\_() methods are created automatically.

Another key feature is that the local variables and execution state are automatically saved between calls. This made the function easier to write and much more clear than an approach using instance variables like self.index and self.data.

In addition to automatic method creation and saving program state, when generators terminate, they automatically raise StopIteration. In combination, these features make it easy to create iterators with no more effort than writing a regular function.

**GENERATORS EXPRESSIONS**

Some simple generators can be coded succinctly as expressions using a syntax similar to list comprehensions but with parentheses instead of square brackets. These expressions are designed for situations where the generator is used right away by an enclosing function. Generator expressions are more compact but less versatile than full generator definitions and tend to be more memory friendly than equivalent list comprehensions.

Examples:

>>>

>>> sum(i\*i for i in range(10)) # sum of squares

285

>>> xvec = [10, 20, 30]

>>> yvec = [7, 5, 3]

>>> sum(x\*y for x,y in zip(xvec, yvec)) # dot product

260

>>> from math import pi, sin

>>> sine\_table = {x: sin(x\*pi/180) for x in range(0, 91)}

>>> unique\_words = set(word for line in page for word in line.split())

>>> valedictorian = max((student.gpa, student.name) for student in graduates)

>>> data = 'golf'

>>> list(data[i] for i in range(len(data)-1, -1, -1))

['f', 'l', 'o', 'g']

Operating System Interface

The os module provides dozens of functions for interacting with the operating system:

>>>

>>> import os

>>> os.getcwd() # Return the current working directory

'C:\\Python37'

>>> os.chdir('/server/accesslogs') # Change current working directory

>>> os.system('mkdir today') # Run the command mkdir in the system shell

0

Be sure to use the import os style instead of from os import \*. This will keep os.open() from shadowing the built-in open() function which operates much differently.

The built-in dir() and help() functions are useful as interactive aids for working with large modules like os:

>>>

>>> import os

>>> dir(os)

<returns a list of all module functions>

>>> help(os)

<returns an extensive manual page created from the module's docstrings>

For daily file and directory management tasks, the shutil module provides a higher level interface that is easier to use:

>>>

>>> import shutil

>>> shutil.copyfile('data.db', 'archive.db')

'archive.db'

>>> shutil.move('/build/executables', 'installdir')

'installdir'

**Command Line Arguments**

Common utility scripts often need to process command line arguments. These arguments are stored in the sys module’s argv attribute as a list. For instance the following output results from running python demo.py one two three at the command line:

>>>

>>> import sys

>>> print(sys.argv)

['demo.py', 'one', 'two', 'three']

The argparse module provides a mechanism to process command line arguments. It should always be preferred over directly processing sys.argv manually.

Take, for example, the below snippet of code:

>>>

>>> import argparse

>>> from getpass import getuser

>>> parser = argparse.ArgumentParser(description='An argparse example.')

>>> parser.add\_argument('name', nargs='?', default=getuser(), help='The name of someone to greet.')

>>> parser.add\_argument('--verbose', '-v', action='count')

>>> args = parser.parse\_args()

>>> greeting = ["Hi", "Hello", "Greetings! its very nice to meet you"][args.verbose % 3]

>>> print(f'{greeting}, {args.name}')

>>> if not args.verbose:

>>> print('Try running this again with multiple "-v" flags!')

**GENERATORS**

Generators in Python are a way to create iterators using a function rather than a class. They allow you to iterate over a potentially large set of data without loading the entire set into memory at once. Generators are defined using a special kind of function and the `yield` keyword.

Here's a simple example of a generator function:

```python

def simple\_generator():

yield 1

yield 2

yield 3

# Using the generator

gen = simple\_generator()

print(next(gen)) # Output: 1

print(next(gen)) # Output: 2

print(next(gen)) # Output: 3

# If you try to call next() again, it will raise a StopIteration exception because the generator is exhausted

```

Generators maintain their state between successive calls, so you can use them to generate a sequence of values on the fly.

Here's an example of a generator that generates an infinite sequence of Fibonacci numbers:

```python

def fibonacci\_generator():

a, b = 0, 1

while True:

yield a

a, b = b, a + b

# Using the generator to generate the first 5 Fibonacci numbers

fib\_gen = fibonacci\_generator()

for \_ in range(5):

print(next(fib\_gen))

# Output: 0, 1, 1, 2, 3

```

Generators are memory-efficient because they generate values on the fly and only store the current state of the generator. They are particularly useful when dealing with large datasets or when you want to create an infinite sequence.

You can also use the `for` loop to iterate over the values generated by a generator without explicitly calling `next()`:

```python

for value in simple\_generator():

print(value)

# Output: 1, 2, 3

```

In summary, generators are a powerful tool in Python for creating iterators in a concise and memory-efficient manner. They are created using functions with the `yield` keyword, and they allow you to iterate over a potentially large set of data without loading the entire set into memory.

**ERROR OUTPUT REDIRECTION AND PROGRAM TERMINATION**

The sys module also has attributes for stdin, stdout, and stderr. The latter is useful for emitting warnings and error messages to make them visible even when stdout has been redirected:

>>>

>>> sys.stderr.write('Warning, log file not found starting a new one\n')

Warning, log file not found starting a new one

The most direct way to terminate a script is to use sys.exit().

String Pattern Matching

The re module provides regular expression tools for advanced string processing. For complex matching and manipulation, regular expressions offer succinct, optimized solutions:

>>>

>>> import re

>>> re.findall(r'\bf[a-z]\*', 'which foot or hand fell fastest')

['foot', 'fell', 'fastest']

>>> re.sub(r'(\b[a-z]+) \1', r'\1', 'cat in the the hat')

'cat in the hat'

When only simple capabilities are needed, string methods are preferred because they are easier to read and debug:

>>>

>>> 'tea for too'.replace('too', 'two')

'tea for two'

**Internet Access**

There are a number of modules for accessing the internet and processing internet protocols. Two of the simplest are urllib.request for retrieving data from URLs and smtplib for sending mail:

>>>

>>> from urllib.request import urlopen

>>> with urlopen('http://tycho.usno.navy.mil/cgi-bin/timer.pl') as response:

... for line in response:

... line = line.decode('utf-8') # Decoding the binary data to text.

... if 'EST' in line or 'EDT' in line: # look for Eastern Time

... print(line)

<BR>Nov. 25, 09:43:32 PM EST

**HOW TO MATCH ANY PATTERN OF TEXT**

To match any pattern of text in Python, you can use regular expressions, which are supported by the `re` module. Regular expressions (regex) allow you to define a pattern and then search or match text based on that pattern. Here's a simple example:

```python

import re

# Define a simple pattern

pattern = re.compile(r'apple')

# Test if the pattern matches a string

text = "I like apples and oranges."

match = pattern.search(text)

if match:

print("Pattern found:", match.group())

else:

print("Pattern not found.")

```

In this example, the `re.compile` function is used to create a regular expression pattern. The `r` before the string denotes a raw string, which is often used with regular expressions to avoid issues with backslashes. The `search` method is then used to check if the pattern is present in the given text. If a match is found, the `group()` method is used to retrieve the matched text.

You can use various metacharacters and special sequences in your regular expression pattern to match different types of characters and sequences. Here are some common examples:

- `.` (dot): Matches any character except a newline.

- `\*`: Matches 0 or more occurrences of the preceding character or group.

- `+`: Matches 1 or more occurrences of the preceding character or group.

- `?`: Matches 0 or 1 occurrence of the preceding character or group.

- `^`: Matches the start of a string.

- `$`: Matches the end of a string.

- `\d`: Matches any digit (equivalent to [0-9]).

- `\w`: Matches any word character (alphanumeric + underscore).

For example, the pattern `r'\d+'` would match one or more consecutive digits in a string.

Here's an example using a more complex pattern:

```python

import re

# Match a date in the format YYYY-MM-DD

pattern = re.compile(r'\d{4}-\d{2}-\d{2}')

text = "The meeting is scheduled for 2023-11-15. Don't forget!"

match = pattern.search(text)

if match:

print("Date found:", match.group())

else:

print("Date not found.")

```

This example uses the `\d{4}-\d{2}-\d{2}` pattern to match a date in the format YYYY-MM-DD. Adjust your pattern according to the specific text patterns you want to match. Regular expressions can be powerful, but they can also be complex, so be sure to refer to the Python `re` module documentation for more details: <https://docs.python.org/3/library/re.html>

**REGULAR EXPRESSIONS**

Regular expressions, often referred to as regex or regexp, provide a concise and flexible means for matching strings of text. They are a powerful tool for text processing, allowing you to define patterns that can be used for searching, matching, and manipulating text. In Python, the `re` module provides support for regular expressions.

Here are some key concepts and examples related to regular expressions in Python:

1. \*\*Creating a Pattern:\*\*

To use regular expressions, you need to create a pattern using the `re.compile()` function or directly use regex functions with patterns.

```python

import re

pattern = re.compile(r'ab+c')

```

In this example, `r'ab+c'` is a raw string representing a regex pattern that matches 'a', followed by one or more 'b's, and ending with 'c'.

2. \*\*Basic Matching:\*\*

- `search()`: Searches the string for a match and returns a match object if there is a match.

```python

text = "abc, abbc, abbbc"

match = pattern.search(text)

if match:

print("Match found:", match.group())

```

3. \*\*Anchors and Boundaries:\*\*

- `^`: Matches the start of a string.

- `$`: Matches the end of a string.

- `\b`: Matches a word boundary.

```python

pattern = re.compile(r'\bword\b')

```

4. \*\*Character Classes:\*\*

- `[...]`: Matches any single character within the brackets.

- `[^...]`: Matches any single character not within the brackets.

- `\d`: Matches any digit (equivalent to `[0-9]`).

- `\w`: Matches any word character (alphanumeric + underscore).

```python

pattern = re.compile(r'[aeiou]')

```

5. \*\*Quantifiers:\*\*

- `\*`: Matches 0 or more occurrences of the preceding character or group.

- `+`: Matches 1 or more occurrences of the preceding character or group.

- `?`: Matches 0 or 1 occurrence of the preceding character or group.

- `{n}`: Matches exactly n occurrences of the preceding character or group.

- `{n,}`: Matches n or more occurrences of the preceding character or group.

- `{n,m}`: Matches between n and m occurrences of the preceding character or group.

```python

pattern = re.compile(r'\d{2,4}')

```

6. \*\*Special Sequences:\*\*

- `\s`: Matches any whitespace character.

- `\S`: Matches any non-whitespace character.

- `\b`: Matches a word boundary.

- `\B`: Matches a non-word boundary.

```python

pattern = re.compile(r'\bword\b')

```

7. \*\*Grouping and Capturing:\*\*

You can use parentheses `()` to create groups, and `group()` method of the match object to retrieve the matched text within a group.

```python

pattern = re.compile(r'(\d+)-(\d+)-(\d+)')

```

This pattern matches a date in the format "YYYY-MM-DD" and captures each part separately.

```python

match = pattern.search("2023-11-15")

if match:

year, month, day = match.groups()

print(f"Year: {year}, Month: {month}, Day: {day}")

```

8. \*\*Flags:\*\*

The `re` module supports flags that modify the behavior of patterns. For example, `re.IGNORECASE` can be used to perform a case-insensitive match.

```python

pattern = re.compile(r'apple', re.IGNORECASE)

```

These are some fundamental concepts and examples related to regular expressions in Python. Regular expressions can become quite complex, and it's often helpful to experiment with them using online tools or regex testers. The Python `re` module documentation (https://docs.python.org/3/library/re.html) is a valuable resource for more in-depth information and examples

**DAY 3 DATES AND TIME, DATA COMPRESSION, OUTPUT FORMATTING, AND MORE!**

Dates and Times, Data Compression, Performance Measurement, and Quality Control

The datetime module supplies classes for manipulating dates and times in both simple and complex ways. While date and time arithmetic is supported, the focus of the implementation is on efficient member extraction for output formatting and manipulation. The module also supports objects that are timezone aware.

>>>

>>> # dates are easily constructed and formatted

>>> from datetime import date

>>> now = date.today()

>>> now

datetime.date(2003, 12, 2)

>>> now.strftime("%m-%d-%y. %d %b %Y is a %A on the %d day of %B.")

'12-02-03. 02 Dec 2003 is a Tuesday on the 02 day of December.'

>>> # dates support calendar arithmetic

>>> birthday = date(1964, 7, 31)

>>> age = now - birthday

>>> age.days

14368

**Data Compression**

Common data archiving and compression formats are directly supported by modules including: zlib, gzip, bz2, lzma, zipfile and tarfile.

>>>

>>> import zlib

>>> s = b'witch which has which witches wrist watch'

>>> len(s)

41

>>> t = zlib.compress(s)

>>> len(t)

37

>>> zlib.decompress(t)

b'witch which has which witches wrist watch'

>>> zlib.crc32(s)

226805979

**Performance Measurement**

Some Python users develop a deep interest in knowing the relative performance of different approaches to the same problem. Python provides a measurement tool that answers those questions immediately.

For example, it may be tempting to use the tuple packing and unpacking feature instead of the traditional approach to swapping arguments. The timeit module quickly demonstrates a modest performance advantage:

>>>

>>> from timeit import Timer

>>> Timer('t=a; a=b; b=t', 'a=1; b=2').timeit()

0.57535828626024577

>>> Timer('a,b = b,a', 'a=1; b=2').timeit()

0.54962537085770791

In contrast to timeit’s fine level of granularity, the profile and pstats modules provide tools for identifying time critical sections in larger blocks of code.

Quality Control

One approach for developing high quality software is to write tests for each function as it is developed and to run those tests frequently during the development process.

The doctest module provides a tool for scanning a module and validating tests embedded in a program’s docstrings. Test construction is as simple as cutting-and-pasting a typical call along with its results into the docstring. This improves the documentation by providing the user with an example and it allows the doctest module to make sure the code remains true to the documentation:

def average(values):

"""Computes the arithmetic mean of a list of numbers.

>>> print(average([20, 30, 70]))

40.0

"""

return sum(values) / len(values)

import doctest

doctest.testmod() # automatically validate the embedded tests

The unittest module is not as effortless as the doctest module, but it allows a more comprehensive set of tests to be maintained in a separate file:

import unittest

class TestStatisticalFunctions(unittest.TestCase):

def test\_average(self):

self.assertEqual(average([20, 30, 70]), 40.0)

self.assertEqual(round(average([1, 5, 7]), 1), 4.3)

with self.assertRaises(ZeroDivisionError):

average([])

with self.assertRaises(TypeError):

average(20, 30, 70)

unittest.main() # Calling from the command line invokes all tests.

**DATETIME MODULE**

The `datetime` module in Python provides classes for working with dates and times. It includes several classes, such as `datetime`, `date`, `time`, `timedelta`, `timezone`, and others, to facilitate manipulation and formatting of date and time information.

Here is a brief overview of some of the key classes in the `datetime` module:

1. \*\*`datetime` Class:\*\*

The `datetime` class is used to represent a point in time, including both date and time components.

```python

from datetime import datetime

current\_datetime = datetime.now()

print(current\_datetime)

```

You can also create a `datetime` object with specific date and time values:

```python

specific\_datetime = datetime(2023, 11, 15, 12, 30, 0)

print(specific\_datetime)

```

2. \*\*`date` Class:\*\*

The `date` class represents a date (year, month, day) without the time component.

```python

from datetime import date

current\_date = date.today()

print(current\_date)

```

3. \*\*`time` Class:\*\*

The `time` class represents time without the date component.

```python

from datetime import time

specific\_time = time(12, 30, 0)

print(specific\_time)

```

4. \*\*`timedelta` Class:\*\*

The `timedelta` class represents the difference between two dates or times.

```python

from datetime import timedelta

duration = timedelta(days=5, hours=3)

new\_datetime = current\_datetime + duration

print(new\_datetime)

```

5. \*\*Formatting and Parsing:\*\*

The `strftime` method is used to format a `datetime` object as a string, and `strptime` is used to parse a string into a `datetime` object.

```python

formatted\_date = current\_datetime.strftime("%Y-%m-%d %H:%M:%S")

print(formatted\_date)

parsed\_datetime = datetime.strptime("2023-11-15 12:30:00", "%Y-%m-%d %H:%M:%S")

print(parsed\_datetime)

```

6. \*\*Time Zones:\*\*

The `timezone` class allows you to work with time zones. You can use `datetime` objects with time zone information.

```python

from datetime import datetime, timezone, timedelta

utc\_now = datetime.now(timezone.utc)

eastern\_time = utc\_now.astimezone(timezone(timedelta(hours=-5)))

print(eastern\_time)

```

These are just some basic examples of using the `datetime` module in Python. The module provides a rich set of functionalities for working with dates and times, including arithmetic operations, comparisons, and more. For more details, you can refer to the official documentation: [datetime — Basic date and time types](<https://docs.python.org/3/library/datetime.html>).

**PERFOMANCE MEASUREMENT WITH THE TIME IT MODULE**

The `datetime` module in Python provides classes for working with dates and times. It includes several classes, such as `datetime`, `date`, `time`, `timedelta`, `timezone`, and others, to facilitate manipulation and formatting of date and time information.

Here is a brief overview of some of the key classes in the `datetime` module:

1. \*\*`datetime` Class:\*\*

The `datetime` class is used to represent a point in time, including both date and time components.

```python

from datetime import datetime

current\_datetime = datetime.now()

print(current\_datetime)

```

You can also create a `datetime` object with specific date and time values:

```python

specific\_datetime = datetime(2023, 11, 15, 12, 30, 0)

print(specific\_datetime)

```

2. \*\*`date` Class:\*\*

The `date` class represents a date (year, month, day) without the time component.

```python

from datetime import date

current\_date = date.today()

print(current\_date)

```

3. \*\*`time` Class:\*\*

The `time` class represents time without the date component.

```python

from datetime import time

specific\_time = time(12, 30, 0)

print(specific\_time)

```

4. \*\*`timedelta` Class:\*\*

The `timedelta` class represents the difference between two dates or times.

```python

from datetime import timedelta

duration = timedelta(days=5, hours=3)

new\_datetime = current\_datetime + duration

print(new\_datetime)

```

5. \*\*Formatting and Parsing:\*\*

The `strftime` method is used to format a `datetime` object as a string, and `strptime` is used to parse a string into a `datetime` object.

```python

formatted\_date = current\_datetime.strftime("%Y-%m-%d %H:%M:%S")

print(formatted\_date)

parsed\_datetime = datetime.strptime("2023-11-15 12:30:00", "%Y-%m-%d %H:%M:%S")

print(parsed\_datetime)

```

6. \*\*Time Zones:\*\*

The `timezone` class allows you to work with time zones. You can use `datetime` objects with time zone information.

```python

from datetime import datetime, timezone, timedelta

utc\_now = datetime.now(timezone.utc)

eastern\_time = utc\_now.astimezone(timezone(timedelta(hours=-5)))

print(eastern\_time)

```

These are just some basic examples of using the `datetime` module in Python. The module provides a rich set of functionalities for working with dates and times, including arithmetic operations, comparisons, and more. For more details, you can refer to the official documentation: [datetime — Basic date and time types](<https://docs.python.org/3/library/datetime.html>).

**OUTPUT FORMATTING**

Output formatting in Python depends on the context and the type of data you want to present. Here are some common methods for formatting output in Python:

1. \*\*String Formatting:\*\*

- The `%` operator: You can use the `%` operator for string formatting. For example:

```python

name = "Alice"

age = 25

print("Name: %s, Age: %d" % (name, age))

```

- The `format` method: This method is more flexible and is recommended for modern Python code.

```python

name = "Alice"

age = 25

print("Name: {}, Age: {}".format(name, age))

```

- f-strings (Python 3.6 and newer):

```python

name = "Alice"

age = 25

print(f"Name: {name}, Age: {age}")

```

2. \*\*String Concatenation:\*\*

You can concatenate strings using the `+` operator:

```python

name = "Alice"

age = 25

print("Name: " + name + ", Age: " + str(age))

```

3. \*\*Formatted String Literals (f-strings):\*\*

As mentioned earlier, f-strings provide a concise and readable way to format strings. They are available in Python 3.6 and later.

```python

name = "Alice"

age = 25

print(f"Name: {name}, Age: {age}")

```

4. \*\*Using `print` with Multiple Arguments:\*\*

The `print` function in Python can take multiple arguments, and it automatically separates them with a space:

```python

name = "Alice"

age = 25

print("Name:", name, "Age:", age)

```

5. \*\*String Methods:\*\*

Various string methods can be used for formatting, such as `str.format()`:

```python

name = "Alice"

age = 25

print("Name: {}, Age: {}".format(name, age))

```

6. \*\*Precision in Floating-Point Numbers:\*\*

If you are dealing with floating-point numbers and need to control precision, you can use formatting options:

```python

pi = 3.141592653589793

print("Value of pi: {:.2f}".format(pi))

```

These are just a few examples, and the choice of formatting method depends on your specific needs and the version of Python you are using. f-strings are generally preferred for their simplicity and readability, but other methods are still widely used, especially in older codebases or when compatibility with older Python versions is required.

**DAY 4 LOGGING, MANAGING PACKAGES WITH PIP, AND FLOATING POINT ARITHMETIC EDIT**