Ultimate Python Cheat Sheet: Practical Python For Everyday Tasks

Working With Files

1. Reading a File

To read the entire content of a file:

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

2. Writing to a File

To write text to a file, overwriting existing content:

```
with open('example.txt', 'w') as file:
    file.write('Hello, Python!')
```

3. Appending to a File

To add text to the end of an existing file:

```
with open('example.txt', 'a') as file:
    file.write('\nAppend this line.')
```

4. Reading Lines into a List

To read a file line by line into a list:

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

5. Iterating Over Each Line in a File

To process each line in a file:

```
with open('example.txt', 'r') as file:
    for line in file:
        print(line.strip())
```

6. Checking If a File Exists

To check if a file exists before performing file operations:

```
import os
if os.path.exists('example.txt'):
    print('File exists.')
else:
    print('File does not exist.')
```

7. Writing Lists to a File

To write each element of a list to a new line in a file:

```
lines = ['First line', 'Second line', 'Third line']
with open('example.txt', 'w') as file:
```

Ultimate Python Cheat Sheet: Practical Python For Everyday Tasks

```
    for line in lines:
        file.write(f'{line}\n')
```

8. Using With Blocks for Multiple Files

To work with multiple files simultaneously using with blocks:

```
with open('source.txt', 'r') as source, open('destination.txt', 'w') as destination:
    content = source.read()
    destination.write(content)
```

9. Deleting a File

To safely delete a file if it exists:

```
import os
if os.path.exists('example.txt'):
    os.remove('example.txt')
    print('File deleted.')
else:
    print('File does not exist.')
```

10. Reading and Writing Binary Files

To read from and write to a file in binary mode (useful for images, videos, etc.):

```
# Reading a binary file
with open('image.jpg', 'rb') as file:
    content = file.read()
# Writing to a binary file
with open('copy.jpg', 'wb') as file:
    file.write(content)
```

Working With Simple HTTP APIs

1. Basic GET Request

To fetch data from an API endpoint using a GET request:

```
import requests
response = requests.get('https://api.example.com/data')
data = response.json()  # Assuming the response is JSON
print(data)
```

2. GET Request with Query Parameters

To send a GET request with query parameters:

```
import requests
params = {'key1': 'value1', 'key2': 'value2'}
response = requests.get('https://api.example.com/search', params=params)
```

```
data = response.json()
print(data)
```

3. Handling HTTP Errors

To handle possible HTTP errors gracefully:

```
import requests
response = requests.get('https://api.example.com/data')
try:
    response.raise_for_status()  # Raises an HTTPError if the status is 4xx, 5xx
    data = response.json()
    print(data)
```

except requests.exceptions.HTTPError as err:

```
    print(f'HTTP Error: {err}')
```

4. Setting Timeout for Requests

To set a timeout for API requests to avoid hanging indefinitely:

```
import requests
try:
    response = requests.get('https://api.example.com/data', timeout=5)  # Timeout in s
    data = response.json()
    print(data)
```

except requests.exceptions.Timeout:

```
    print('The request timed out')
```

5. Using Headers in Requests

To include headers in your request (e.g., for authorization):

```
import requests
```

headers = {'Authorization': 'Bearer YOUR_ACCESS_TOKEN'}

response = requests.get('https://api.example.com/protected', headers=headers)

```
data = response.json()
print(data)
```

6. POST Request with JSON Payload

To send data to an API endpoint using a POST request with a JSON payload:

```
import requests
payload = {'key1': 'value1', 'key2': 'value2'}
```

headers = {'Content-Type': 'application/json'}

response = requests.post('https://api.example.com/submit', json=payload, headers=headers)

```
print(response.json())
```

Ultimate Python Cheat Sheet: Practical Python For Everyday Tasks

7. Handling Response Encoding

To handle the response encoding properly:

```
import requests
response = requests.get('https://api.example.com/data')
response.encoding = 'utf-8'  # Set encoding to match the expected response format
data = response.text
print(data)
```

8. Using Sessions with Requests

To use a session object for making multiple requests to the same host, which can improve performance:

```
import requests
```

with requests.Session() as session:

```
    session.headers.update({'Authorization': 'Bearer YOUR_ACCESS_TOKEN'})
    response = session.get('https://api.example.com/data')
    print(response.json())
```

9. Handling Redirects

To handle or disable redirects in requests:

```
import requests
response = requests.get('https://api.example.com/data', allow_redirects=False)
print(response.status_code)
```

10. Streaming Large Responses

To stream a large response to process it in chunks, rather than loading it all into memory:

```
import requests
response = requests.get('https://api.example.com/large-data', stream=True)
for chunk in response.iter_content(chunk_size=1024):
    process(chunk)  # Replace 'process' with your actual processing function
```

Working With Lists

1. Creating a List

To conjure a list into being:

```
# A list of mystical elements
elements = ['Earth', 'Air', 'Fire', 'Water']
```

2. Appending to a List

To append a new element to the end of a list:

```
elements.append('Aether')
```

3. Inserting into a List

To insert an element at a specific position in the list:

Ultimate Python Cheat Sheet: Practical Python For Everyday Tasks

```
# Insert 'Spirit' at index 1
elements.insert(1, 'Spirit')
```

4. Removing from a List

To remove an element by value from the list:

```
elements.remove('Earth')  # Removes the first occurrence of 'Earth'
```

5. Popping an Element from a List

To remove and return an element at a given index (default is the last item):

```
last_element = elements.pop()  # Removes and returns the last element
```

6. Finding the Index of an Element

To find the index of the first occurrence of an element:

```
index_of_air = elements.index('Air')
```

7. List Slicing

To slice a list, obtaining a sub-list:

```
# Get elements from index 1 to 3
sub_elements = elements[1:4]
```

8. List Comprehension

To create a new list by applying an expression to each element of an existing one:

```
# Create a new list with lengths of each element
lengths = [len(element) for element in elements]
```

9. Sorting a List

To sort a list in ascending order (in-place):

```
elements.sort()
```

10. Reversing a List

To reverse the elements of a list in-place:

```
elements.reverse()
```

Working With Dictionaries

1. Creating a Dictionary

To forge a new dictionary:

```
# A tome of elements and their symbols
elements = {'Hydrogen': 'H', 'Helium': 'He', 'Lithium': 'Li'}
```

2. Adding or Updating Entries

To add a new entry or update an existing one:

```
elements['Carbon'] = 'C'  # Adds 'Carbon' or updates its value to 'C'
```

3. Removing an Entry

To banish an entry from the dictionary:

Ultimate Python Cheat Sheet: Practical Python For Everyday Tasks

```
del elements['Lithium']  # Removes the key 'Lithium' and its value
```

4. Checking for Key Existence

To check if a key resides within the dictionary:

if 'Helium' in elements:

```
    print('Helium is present')
```

5. Iterating Over Keys

To iterate over the keys in the dictionary:

```
for element in elements:
    print(element)  # Prints each key
```

6. Iterating Over Values

To traverse through the values in the dictionary:

```
for symbol in elements.values():
    print(symbol)  # Prints each value
```

7. Iterating Over Items

To journey through both keys and values together:

```
for element, symbol in elements.items():
    print(f'{element}: {symbol}')
```

8. Dictionary Comprehension

To conjure a new dictionary through an incantation over an iterable:

```
# Squares of numbers from 0 to 4
squares = {x: x**2 for x in range(5)}
```

9. Merging Dictionaries

To merge two or more dictionaries, forming a new alliance of their entries:

alchemists = {'Paracelsus': 'Mercury'}

philosophers = {'Plato': 'Aether'}

merged = {**alchemists, **philosophers} # Python 3.5+

10. Getting a Value with Default

To retrieve a value safely, providing a default for absent keys:

```
element = elements.get('Neon', 'Unknown')  # Returns 'Unknown' if 'Neon' is not found
```

Working With The Operating System

1. Navigating File Paths

To craft and dissect paths, ensuring compatibility across realms (operating systems):

```
import os
# Craft a path compatible with the underlying OS
path = os.path.join('mystic', 'forest', 'artifact.txt')
```

Ultimate Python Cheat Sheet: Practical Python For Everyday Tasks

```
# Retrieve the tome's directory
directory = os.path.dirname(path)
# Unveil the artifact's name
artifact_name = os.path.basename(path)
```

2. Listing Directory Contents

To reveal all entities within a mystical directory:

```
import os
contents = os.listdir('enchanted_grove')
print(contents)
```

3. Creating Directories

To conjure new directories within the fabric of the filesystem:

```
import os
# create a single directory
os.mkdir('alchemy_lab')
# create a hierarchy of directories
os.makedirs('alchemy_lab/potions/elixirs')
```

4. Removing Files and Directories

To erase files or directories, banishing their essence:

```
import os
# remove a file
os.remove('unnecessary_scroll.txt')
# remove an empty directory
os.rmdir('abandoned_hut')
# remove a directory and its contents
import shutil
shutil.rmtree('cursed_cavern')
```

5. Executing Shell Commands

To invoke the shell's ancient powers directly from Python:

```
import subprocess
# Invoke the 'echo' incantation
result = subprocess.run(['echo', 'Revealing the arcane'], capture_output=True, text=Tr
print(result.stdout)
```

6. Working with Environment Variables

To read and inscribe upon the ethereal environment variables:

```
import os
```

Ultimate Python Cheat Sheet: Practical Python For Everyday Tasks

Read the 'PATH' variable

path = os.environ.get('PATH')

Create a new environment variable

os.environ['MAGIC'] = 'Arcane'

7. Changing the Current Working Directory

To shift your presence to another directory within the filesystem:

```
import os
```

Traverse to the 'arcane_library' directory

os.chdir('arcane_library')

8. Path Existence and Type

To discern the existence of paths and their nature - be they file or directory:

```
import os
# Check if a path exists
exists = os.path.exists('mysterious_ruins')
# Ascertain if the path is a directory
is_directory = os.path.isdir('mysterious_ruins')
# Determine if the path is a file
is_file = os.path.isfile('ancient_manuscript.txt')
```

9. Working with Temporary Files

To summon temporary files and directories, fleeting and ephemeral:

```
import tempfile
```

Create a temporary file

temp_file = tempfile.NamedTemporaryFile(delete=False)

print(temp_file.name)

Erect a temporary directory

temp_dir = tempfile.TemporaryDirectory()

print(temp_dir.name)

10. Getting System Information

To unveil information about the host system, its name, and the enchantments it supports:

```
import os
import platform
# Discover the operating system
os_name = os.name  # 'posix', 'nt', 'java'
# Unearth detailed system information
system_info = platform.system()  # 'Linux', 'Windows', 'Darwin'
```

Ultimate Python Cheat Sheet: Practical Python For Everyday Tasks

Working With CLI - STDIN, STDOUT, STDERR

1. Reading User Input

Getting input from STDIN:

user_input = input("Impart your wisdom: ")

print(f"You shared: {user_input}")

2. Printing to STDOUT

To print messages to the console:

```
print("Behold, the message of the ancients!")
```

3. Formatted Printing

To weave variables into your messages with grace and precision:

```
name = "Merlin"
age = 300
print(f"{name}, of {age} years, speaks of forgotten lore.")
```

4. Reading Lines from STDIN

Trim whitespaces line by line from STDIN:

```
import sys
for line in sys.stdin:
    print(f"Echo from the void: {line.strip()}")
```

5. Writing to STDERR

To send message to STDERR:

```
import sys
sys.stderr.write("Beware! The path is fraught with peril.\n")
```

6. Redirecting STDOUT

To redirect the STDOUT:

```
import sys
original_stdout = sys.stdout  # Preserve the original STDOUT
with open('mystic_log.txt', 'w') as f:
    sys.stdout = f  # Redirect STDOUT to a file
    print("This message is inscribed within the mystic_log.txt.")
sys.stdout = original_stdout  # Restore STDOUT to its original glory
```

7. Redirecting STDERR

Redirecting STDERR:

```
import sys
with open('warnings.txt', 'w') as f:
    sys.stderr = f  # Redirect STDERR
```

Ultimate Python Cheat Sheet: Practical Python For Everyday Tasks

```
print("This warning is sealed within warnings.txt.", file=sys.stderr)
```

8. Prompting for Passwords

To prompt for passwords:

```
import getpass
secret_spell = getpass.getpass("Whisper the secret spell: ")
```

9. Command Line Arguments

Working with and parsing command line arguments:

```
import sys
```

The script's name is the first argument, followed by those passed by the invoker

script, first_arg, second_arg = sys.argv

print(f"Invoked with the sacred tokens: {first_arg} and {second_arg}")

10. Using Argparse for Complex CLI Interactions

Adding descriptions and options/arguments:

```
import argparse
parser = argparse.ArgumentParser(description="Invoke the ancient scripts.")
parser.add_argument('spell', help="The spell to cast")
parser.add_argument('--power', type=int, help="The power level of the spell")
args = parser.parse_args()
print(f"Casting {args.spell} with power {args.power}")
```

Working With Mathematical Operations and Permutations

1. Basic Arithmetic Operations

To perform basic arithmetic:

```
sum = 7 + 3  # Addition
difference = 7 - 3  # Subtraction
product = 7 * 3  # Multiplication
quotient = 7 / 3  # Division
remainder = 7 % 3  # Modulus (Remainder)
power = 7 ** 3  # Exponentiation
```

2. Working with Complex Numbers

To work with complex numbers:

z = complex(2, 3) # Create a complex number 2 + 3j

real_part = z.real # Retrieve the real part

imaginary_part = z.imag # Retrieve the imaginary part

conjugate = z.conjugate() # Get the conjugate

3. Mathematical Functions

Ultimate Python Cheat Sheet: Practical Python For Everyday Tasks

Common math functions:

```
import math
root = math.sqrt(16)  # Square root
logarithm = math.log(100, 10)  # Logarithm base 10 of 100
sine = math.sin(math.pi / 2)  # Sine of 90 degrees (in radians)
```

4. Generating Permutations

Easy way to generate permutations from a given set:

```
from itertools import permutations
paths = permutations([1, 2, 3])  # Generate all permutations of the list [1, 2, 3]
for path in paths:
    print(path)
```

5. Generating Combinations

Easy way to generate combinations:

```
from itertools import combinations
combos = combinations([1, 2, 3, 4], 2)  # Generate all 2-element combinations
for combo in combos:
    print(combo)
```

6. Random Number Generation

To get a random number:

```
import random
num = random.randint(1, 100)  # Generate a random integer between 1 and 100
```

7. Working with Fractions

When you need to work with fractions:

from fractions import Fraction

f = Fraction(3, 4) # Create a fraction 3/4

print(f + 1) # Add a fraction and an integer

8. Statistical Functions

To get Average, Median, and Standard Deviation:

```
import statistics
data = [1, 2, 3, 4, 5]
mean = statistics.mean(data)  # Average
median = statistics.median(data)  # Median
stdev = statistics.stdev(data)  # Standard Deviation
```

9. Trigonometric Functions

To work with trigonometry:

```
import math
angle_rad = math.radians(60)  # Convert 60 degrees to radians
cosine = math.cos(angle_rad)  # Cosine of the angle
```

10. Handling Infinity and NaN

To work with Infinity and NaN:

```
import math
infinity = math.inf  # Representing infinity
not_a_number = math.nan  # Representing a non-number (NaN)
```

Working With Databases

1. Establishing a Connection

To create a connection to a Postgres Database:

```
import psycopg2
connection = psycopg2.connect(
    dbname='your_database',
    user='your_username',
    password='your_password',
    host='your_host'
)
```

2. Creating a Cursor

To create a database cursor, enabling the traversal and manipulation of records:

```
cursor = connection.cursor()
```

3. Executing a Query

Selecting data from Database:

```
cursor.execute("SELECT * FROM your_table")
```

4. Fetching Query Results

Fetching data with a cursor:

```
records = cursor.fetchall()
for record in records:
    print(record)
```

5. Inserting Records

To insert data into tables in a database:

```
cursor.execute("INSERT INTO your_table (column1, column2) VALUES (%s, %s)", ('value1',
connection.commit()  # Seal the transaction
```

6. Updating Records

To alter the records:

```
cursor.execute("UPDATE your_table SET column1 = %s WHERE column2 = %s", ('new_value',
connection.commit()
```

7. Deleting Records

To delete records from the table:

```
cursor.execute("DELETE FROM your_table WHERE condition_column = %s", ('condition_valu
connection.commit()
```

8. Creating a Table

To create a new table, defining its structure:

```
cursor.execute("""
    CREATE TABLE your_new_table (
        id SERIAL PRIMARY KEY,
        column1 VARCHAR(255),
        column2 INTEGER
    )
""")
connection.commit()
```

9. Dropping a Table

To drop a table:

```
cursor.execute("DROP TABLE if exists your_table")
connection.commit()
```

10. Using Transactions

To use transactions for atomicity:

```
try:
    cursor.execute("your first transactional query")
    cursor.execute("your second transactional query")
    connection.commit()  # Commit if all is well
```

except Exception as e:

```
    connection.rollback()  # Rollback in case of any issue
    print(f"An error occurred: {e}")
```

Working With Async IO (Asyncrounous Programming)

1. Defining an Asynchronous Function

To declare an async function:

```
import asyncio
async def fetch_data():
    print("Fetching data...")
```

```
    await asyncio.sleep(2)  # Simulate an I/O operation
    print("Data retrieved.")
```

2. Running an Asynchronous Function

To invoke an asynchronous function and await them:

```
async def main():
    await fetch_data()
asyncio.run(main())
```

3. Awaiting Multiple Coroutines

To invoke multiple async functions and await all:

```
async def main():
    task1 = fetch_data()
    task2 = fetch_data()
    await asyncio.gather(task1, task2)
asyncio.run(main())
```

4. Creating Tasks

To dispatch tasks:

```
async def main():
    task1 = asyncio.create_task(fetch_data())
    task2 = asyncio.create_task(fetch_data())
    await task1
    await task2
asyncio.run(main())
```

5. Asynchronous Iteration

To traverse through asynchronously, allowing time for other functions in between:

```
async def fetch_item(item):
    await asyncio.sleep(1)  # Simulate an I/O operation
    print(f"Fetched {item}")
async def main():
    items = ['potion', 'scroll', 'wand']
    for item in items:
        await fetch_item(item)
asyncio.run(main())
```

6. Using Asynchronous Context Managers

To ensure resources are managed within the bounds of an asynchronous function:

```
async def async_context_manager():
```

```
        print("Entering context")
        await asyncio.sleep(1)
        print("Exiting context")
    async def main():
        async with async_context_manager():
            print("Within context")
    asyncio.run(main())
```

7. Handling Exceptions in Asynchronous Code

To gracefully catch and manage the errors with async functions:

```
    async def risky_spell():
        await asyncio.sleep(1)
        raise ValueError("The spell backfired!")
    async def main():
        try:
            await risky_spell()
        except ValueError as e:
            print(f"Caught an error: {e}")
    asyncio.run(main())
```

8. Asynchronous Generators

To create async generators, each arriving in its own time:

```
    async def fetch_items():
        items = ['crystal', 'amulet', 'dagger']
        for item in items:
            await asyncio.sleep(1)
            yield item
    async def main():
        async for item in fetch_items():
            print(f"Found {item}")
    asyncio.run(main())
```

9. Using Semaphores

To limit the number of concurrent tasks:

```
    async def guarded_spell(semaphore, item):
        async with semaphore:
            print(f"Processing {item}")
            await asyncio.sleep(1)
```

```
async def main():
    semaphore = asyncio.Semaphore(2)  # Allow 2 concurrent tasks
    await asyncio.gather(*(guarded_spell(semaphore, i) for i in range(5)))
asyncio.run(main())
```

10. Event Loops

To directly engage with the asynchronous loop, customizing the flow of execution:

```
async def perform_spell():
    print("Casting spell...")
    await asyncio.sleep(1)
    print("Spell cast.")
loop = asyncio.get_event_loop()
try:
    loop.run_until_complete(perform_spell())
finally:
    loop.close()
```

Working With Networks, Sockets, and Network Interfaces

1. Creating a Socket

To create a socket for network communication:

```
import socket
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

2. Connecting to a Remote Server

To establish a link with a remote server through the socket:

```
s.connect(('example.com', 80))  # Connect to example.com on port 80
```

3. Sending Data

To dispatch data through the network to a connected entity:

```
s.sendall(b'Hello, server')
```

4. Receiving Data

To receive data from the network:

```
data = s.recv(1024)  # Receive up to 1024 bytes
print('Received', repr(data))
```

5. Closing a Socket

To gracefully close the socket, severing the network link:

```
s.close()
```

6. Creating a Listening Socket

To open a socket that listens for incoming connections:

Ultimate Python Cheat Sheet: Practical Python For Everyday Tasks

```
serversocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
serversocket.bind(('localhost', 8080))  # Bind to localhost on port 8080
serversocket.listen()  # Listen for incoming connections
```

7. Accepting Connections

To accept and establish a network link:

```
clientsocket, address = serversocket.accept()
print(f"Connection from {address} has been established.")
```

8. Non-blocking Socket Operations

To set a socket's mode to non-blocking:

```
s.setblocking(False)
```

9. Working with UDP Sockets

To create a socket for UDP, a protocol for quicker, but less reliable communication:

udp_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

udp_socket.bind(('localhost', 8081)) # Bind UDP socket to localhost on port 8081

10. Enumerating Network Interfaces

To discover the names and addresses of the machine's network interfaces:

```
import socket
import netifaces
for interface in netifaces.interfaces():
    addr = netifaces.ifaddresses(interface).get(netifaces.AF_INET)
    if addr:
        print(f"Interface: {interface}, Address: {addr[0]['addr']}")
```

#

Working With Pandas Library (Dataframes)

1. Creating a DataFrame

To create a DataFrame with your own columns and data:

```
import pandas as pd
data = {
    'Element': ['Earth', 'Water', 'Fire', 'Air'],
    'Symbol': ['■', '■', '■', '■']
}
df = pd.DataFrame(data)
```

2. Reading Data from a CSV File

To read data from a CSV file, transforming it into a DataFrame:

```
df = pd.read_csv('elements.csv')
```

Ultimate Python Cheat Sheet: Practical Python For Everyday Tasks

3. Inspecting the First Few Rows

To get first rows from dataframe:

```
print(df.head())
```

4. Selecting Columns

To select specific columns from dataframe:

```
symbols = df['Symbol']
```

5. Filtering Rows

To sift through the DataFrame, selecting rows that meet your criteria:

```
fire_elements = df[df['Element'] == 'Fire']
```

6. Creating New Columns

To create new columns in DataFrame derived from the data within:

```
df['Length'] = df['Element'].apply(len)
```

7. Grouping and Aggregating Data

To gather your data into groups and extract new data through aggregation:

```
element_groups = df.groupby('Element').agg({'Length': 'mean'})
```

8. Merging DataFrames

To weave together two DataFrames, joining them by a shared key:

df2 = pd.DataFrame({'Element': ['Earth', 'Fire'], 'Quality': ['Solid', 'Plasma']})

merged_df = pd.merge(df, df2, on='Element')

9. Handling Missing Data

To clean your DataFrame, filling the voids where data is absent:

```
df.fillna(value='Unknown', inplace=True)
```

10. Pivoting and Reshaping Data

To transmute the shape of your DataFrame, revealing hidden patterns and structures with a pivot operation:

```
pivoted_df = df.pivot(index='Element', columns='Symbol', values='Length')
```

Working With Numpy Library (Arrays)

1. Creating a NumPy Array

To create an array:

```
import numpy as np
array = np.array([1, 2, 3, 4, 5])
```

2. Array of Zeros or Ones

To create an array filled with zeros:

```
zeros = np.zeros((3, 3))  # A 3x3 array of zeros
ones = np.ones((2, 4))  # A 2x4 array of ones
```

3. Creating a Range of Numbers

Ultimate Python Cheat Sheet: Practical Python For Everyday Tasks

To create a sequence of numbers:

```
range_array = np.arange(10, 50, 5)  # From 10 to 50, step by 5
```

4. Creating a Linearly Spaced Array

To create a series of values, evenly spaced between two bounds:

```
linear_spaced = np.linspace(0, 1, 5)  # 5 values from 0 to 1
```

5. Reshaping an Array

To transmute the shape of an array, altering its dimensions:

```
reshaped = np.arange(9).reshape(3, 3)  # Reshape a 1D array into a 3x3 2D array
```

6. Basic Array Operations

To perform elemental manipulations upon the arrays:

```
a = np.array([1, 2, 3])
b = np.array([4, 5, 6])
sum = a + b  # Element-wise addition
difference = b - a  # Element-wise subtraction
product = a * b  # Element-wise multiplication
```

7. Matrix Multiplication

Basic dot product Operation:

```
result = np.dot(a.reshape(1, 3), b.reshape(3, 1))  # Dot product of a and b
```

8. Accessing Array Elements

Accessing array elements with useful syntax:

```
element = a[2]  # Retrieve the third element of array 'a'
row = reshaped[1, :]  # Retrieve the second row of 'reshaped'
```

9. Boolean Indexing

To filter the elements of an array through the sieve of conditionals:

```
filtered = a[a > 2]  # Elements of 'a' greater than 2
```

10. Aggregations and Statistics

Statistical operations on np arrays:

```
mean = np.mean(a)
maximum = np.max(a)
sum = np.sum(a)
```

Working With Matplotlib Library (Data Visualization)

1. Creating a Basic Plot

To create a plot visualization:

```
import matplotlib.pyplot as plt
x = [1, 2, 3, 4, 5]
```

Ultimate Python Cheat Sheet: Practical Python For Everyday Tasks

```
y = [1, 4, 9, 16, 25]
plt.plot(x, y)
plt.show()
```

2. Adding Titles and Labels

To create names for axes and title your plot to give better context:

```
plt.plot(x, y)
plt.title('Growth Over Time')
plt.xlabel('Time')
plt.ylabel('Growth')
plt.show()
```

3. Creating a Scatter Plot

Creating a scatter plot:

```
plt.scatter(x, y)
plt.show()
```

4. Customizing Line Styles and Markers

To add symbols into your plot, enriching its usefulness:

```
plt.plot(x, y, linestyle='--', marker='o', color='b')
plt.show()
```

5. Creating Multiple Plots on the Same Axes

Creating Multiple Plots on the Same Axes:

```
z = [2, 3, 4, 5, 6]
plt.plot(x, y)
plt.plot(x, z)
plt.show()
```

6. Creating Subplots

To create subplots:

```
fig, ax = plt.subplots(2, 1)  # 2 rows, 1 column
ax[0].plot(x, y)
ax[1].plot(x, z)
plt.show()
```

7. Creating a Histogram

To create a histogram:

```
data = [1, 2, 2, 3, 3, 3, 4, 4, 4, 4]
plt.hist(data, bins=4)
plt.show()
```

Ultimate Python Cheat Sheet: Practical Python For Everyday Tasks

8. Adding a Legend

To create a legend for the plot:

```
plt.plot(x, y, label='Growth')
plt.plot(x, z, label='Decay')
plt.legend()
plt.show()
```

9. Customizing Ticks

To create your own marks upon the axes, defining the scale of your values:

```
plt.plot(x, y)
plt.xticks([1, 2, 3, 4, 5], ['One', 'Two', 'Three', 'Four', 'Five'])
plt.yticks([0, 5, 10, 15, 20, 25], ['0', '5', '10', '15', '20', '25+'])
plt.show()
```

10. Saving Figures

To save the plot as a .png:

```
plt.plot(x, y)
plt.savefig('growth_over_time.png')
```

Working With Scikit-Learn Library (Machine Learning)

1. Loading a Dataset

```
To work with datasets for your ML experiments
from sklearn import datasets
iris = datasets.load_iris()
X, y = iris.data, iris.target
```

2. Splitting Data into Training and Test Sets

To divide your data, dedicating portions to training and evaluation:

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)
```

3. Training a Model

Training a ML Model using RandomForestClassifier:

```
from sklearn.ensemble import RandomForestClassifier
model = RandomForestClassifier()
model.fit(X_train, y_train)
```

4. Making Predictions

To access the model predictions:

```
predictions = model.predict(X_test)
```

5. Evaluating Model Performance

To evaluate your model, measuring its accuracy in prediction:

```
from sklearn.metrics import accuracy_score
accuracy = accuracy_score(y_test, predictions)
print(f"Model accuracy: {accuracy}")
```

6. Using Cross-Validation

To use Cross-Validation:

```
from sklearn.model_selection import cross_val_score
scores = cross_val_score(model, X, y, cv=5)
print(f"Cross-validation scores: {scores}")
```

7. Feature Scaling

To create the appropriate scales of your features, allowing the model to learn more effectively:

```
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)
```

8. Parameter Tuning with Grid Search

To refine your model's parameters, seeking the optimal combination:

```
from sklearn.model_selection import GridSearchCV
param_grid = {'n_estimators': [10, 50, 100], 'max_depth': [None, 10, 20]}
grid_search = GridSearchCV(model, param_grid, cv=5)
grid_search.fit(X_train, y_train)
```

9. Pipeline Creation

To streamline your data processing and modeling steps, crafting a seamless flow:

```
from sklearn.pipeline import Pipeline
pipeline = Pipeline([
    ('scaler', StandardScaler()),
    ('classifier', RandomForestClassifier())
])
pipeline.fit(X_train, y_train)
```

10. Saving and Loading a Model

To preserve your model:

```
import joblib
# Saving the model
joblib.dump(model, 'model.joblib')
# Loading the model
```

```
loaded_model = joblib.load('model.joblib')
```

Working With Plotly Library (Interactive Data Visualization)

1. Creating a Basic Line Chart

To create a line chart:

```
import plotly.graph_objs as go
import plotly.io as pio
x = [1, 2, 3, 4, 5]
y = [1, 4, 9, 16, 25]
fig = go.Figure(data=go.Scatter(x=x, y=y, mode='lines'))
pio.show(fig)
```

2. Creating a Scatter Plot

To create a scatter plot:

```
fig = go.Figure(data=go.Scatter(x=x, y=y, mode='markers'))
pio.show(fig)
```

3. Creating a Bar Chart

To Create a Bar Chart:

```
categories = ['A', 'B', 'C', 'D', 'E']
values = [10, 20, 15, 30, 25]
fig = go.Figure(data=go.Bar(x=categories, y=values))
pio.show(fig)
```

4. Creating a Pie Chart

To create a Pie Chart:

```
labels = ['Earth', 'Water', 'Fire', 'Air']
sizes = [25, 35, 20, 20]
fig = go.Figure(data=go.Pie(labels=labels, values=sizes))
pio.show(fig)
```

5. Creating a Histogram

To create a Histogram:

```
data = [1, 2, 2, 3, 3, 3, 4, 4, 4, 4]
fig = go.Figure(data=go.Histogram(x=data))
pio.show(fig)
```

6. Creating Box Plots

To create a Box Plot:

```
data = [1, 2, 2, 3, 4, 4, 4, 5, 5, 6]
fig = go.Figure(data=go.Box(y=data))
```

Ultimate Python Cheat Sheet: Practical Python For Everyday Tasks

```
pio.show(fig)
```

7. Creating Heatmaps

To create a heatmap:

```
import numpy as np
z = np.random.rand(10, 10)  # Generate random data
fig = go.Figure(data=go.Heatmap(z=z))
pio.show(fig)
```

8. Creating 3D Surface Plots

To create a 3D Surface Plot:

```
z = np.random.rand(20, 20)  # Generate random data
fig = go.Figure(data=go.Surface(z=z))
pio.show(fig)
```

9. Creating Subplots

To create a subplot:

```
from plotly.subplots import make_subplots
fig = make_subplots(rows=1, cols=2)
fig.add_trace(go.Scatter(x=x, y=y, mode='lines'), row=1, col=1)
fig.add_trace(go.Bar(x=categories, y=values), row=1, col=2)
pio.show(fig)
```

10. Creating Interactive Time Series

To work with Time Series:

```
import pandas as pd
dates = pd.date_range('20230101', periods=5)
values = [10, 11, 12, 13, 14]
fig = go.Figure(data=go.Scatter(x=dates, y=values, mode='lines+markers'))
pio.show(fig)
```

Working With Dates and Times

1. Getting the Current Date and Time

To get the current data and time:

```
from datetime import datetime
now = datetime.now()
print(f"Current date and time: {now}")
```

2. Creating Specific Date and Time

To conjure a moment from the past or future, crafting it with precision:

```
specific_time = datetime(2023, 1, 1, 12, 30)
```

Ultimate Python Cheat Sheet: Practical Python For Everyday Tasks

```
print(f"Specific date and time: {specific_time}")
```

3. Formatting Dates and Times

Formatting Dates and Times:

formatted = now.strftime("%Y-%m-%d %H:%M:%S")

print(f"Formatted date and time: {formatted}")

4. Parsing Dates and Times from Strings

Parsing Dates and Times from Strings:

```
date_string = "2023-01-01 15:00:00"
```

parsed_date = datetime.strptime(date_string, "%Y-%m-%d %H:%M:%S")

print(f"Parsed date and time: {parsed_date}")

5. Working with Time Deltas

To traverse the distances between moments, leaping forward or backward through time:

from datetime import timedelta

delta = timedelta(days=7)

future_date = now + delta

print(f"Date after 7 days: {future_date}")

6. Comparing Dates and Times

Date and Times comparisons:

```
if specific_time > now:
    print("Specific time is in the future.")
else:
    print("Specific time has passed.")
```

7. Extracting Components from a Date/Time

To extract dates year, month, day, and more:

```
year = now.year
month = now.month
day = now.day
hour = now.hour
minute = now.minute
second = now.second
print(f"Year: {year}, Month: {month}, Day: {day}, Hour: {hour}, Minute: {minute}, Seco
```

8. Working with Time Zones

To work with time zones honoring the local time:

from datetime import timezone, timedelta

utc_time = datetime.now(timezone.utc)

print(f"Current UTC time: {utc_time}")

Adjusting to a specific timezone (e.g., EST)

est_time = utc_time - timedelta(hours=5)

print(f"Current EST time: {est_time}")

9. Getting the Weekday

To identify the day of the week:

```
weekday = now.strftime("%A")
print(f"Today is: {weekday}")
```

10. Working with Unix Timestamps

To converse with the ancient epochs, translating their count from the dawn of Unix:

timestamp = datetime.timestamp(now)

print(f"Current timestamp: {timestamp}")

Converting a timestamp back to a datetime

date_from_timestamp = datetime.fromtimestamp(timestamp)

print(f"Date from timestamp: {date_from_timestamp}")

Working With More Advanced List Comprehensions and Lambda Functions

1. Nested List Comprehensions

To work with nested list Comprehensions:

```
matrix = [[j for j in range(5)] for i in range(3)]
print(matrix)  # Creates a 3x5 matrix
```

2. Conditional List Comprehensions

To filter elements that meet your criteria:

```
filtered = [x for x in range(10) if x % 2 == 0]
print(filtered)  # Even numbers from 0 to 9
```

3. List Comprehensions with Multiple Iterables

To merge and transform elements from multiple sources in a single dance:

```
pairs = [(x, y) for x in [1, 2, 3] for y in [3, 1, 4] if x != y]
print(pairs)  # Pairs of non-equal elements
```

4. Using Lambda Functions

To summon anonymous functions, ephemeral and concise, for a single act of magic:

```
square = lambda x: x**2
print(square(5))  # Returns 25
```

5. Lambda Functions in List Comprehensions

To employ lambda functions within your list comprehensions:

```
squared = [(lambda x: x**2)(x) for x in range(5)]
```

Ultimate Python Cheat Sheet: Practical Python For Everyday Tasks

```
print(squared)  # Squares of numbers from 0 to 4
```

6. List Comprehensions for Flattening Lists

To flatten a nested list, spreading its elements into a single dimension:

```
nested = [[1, 2, 3], [4, 5], [6, 7]]
flattened = [x for sublist in nested for x in sublist]
print(flattened)
```

7. Applying Functions to Elements

To apply a transformation function to each element:

```
import math
transformed = [math.sqrt(x) for x in range(1, 6)]
print(transformed)  # Square roots of numbers from 1 to 5
```

8. Using Lambda with Map and Filter

To map and filter lists:

```
mapped = list(map(lambda x: x**2, range(5)))
filtered = list(filter(lambda x: x > 5, mapped))
print(mapped)    # Squares of numbers from 0 to 4
print(filtered)  # Elements greater than 5
```

9. List Comprehensions with Conditional Expressions

List Comprehensions with Condidtional Expressions:

```
conditional = [x if x > 2 else x**2 for x in range(5)]
print(conditional)  # Squares numbers less than or equal to 2, passes others unchanged
```

10. Complex Transformations with Lambda

To conduct intricate transformations, using lambda functions:

```
complex_transformation = list(map(lambda x: x**2 if x % 2 == 0 else x + 5, range(5)))
print(complex_transformation)  # Applies different transformations based on even-odd
```

Working With Object Oriented Programming

1. Defining a Class

Creating a class:

class Wizard:

```
    def __init__(self, name, power):
        self.name = name
        self.power = power
```

def cast_spell(self):

```
        print(f"{self.name} casts a spell with power {self.power}!")
```

2. Creating an Instance

Ultimate Python Cheat Sheet: Practical Python For Everyday Tasks

To create an instance of your class:

```
merlin = Wizard("Merlin", 100)
```

3. Invoking Methods

To call methods on instance of class:

```
merlin.cast_spell()
```

4. Inheritance

Subclassing:

class ArchWizard(Wizard):

```
    def __init__(self, name, power, realm):
        super().__init__(name, power)
        self.realm = realm
    def summon_familiar(self):
        print(f"{self.name} summons a familiar from the {self.realm} realm.")
```

5. Overriding Methods

To overide base classes:

class Sorcerer(Wizard):

```
    def cast_spell(self):
        print(f"{self.name} casts a powerful dark spell!")
```

6. Polymorphism

To interact with different forms through a common interface:

```
def unleash_magic(wizard):
    wizard.cast_spell()
unleash_magic(merlin)
unleash_magic(Sorcerer("Voldemort", 90))
```

7. Encapsulation

To use information hiding:

class Alchemist:

```
    def __init__(self, secret_ingredient):
        self.__secret = secret_ingredient
    def reveal_secret(self):
        print(f"The secret ingredient is {self.__secret}")
```

8. Composition

To assemble Objects from simpler ones:

class Spellbook:

```
    def __init__(self, spells):
```

```
        self.spells = spells
```

class Mage:

```
    def __init__(self, name, spellbook):
        self.name = name
        self.spellbook = spellbook
```

9. Class Methods and Static Methods

To bind actions to the class itself or liberate them from the instance, serving broader purposes:

class Enchanter:

```
    @staticmethod
    def enchant(item):
        print(f"{item} is enchanted!")
    @classmethod
    def summon(cls):
        print("A new enchanter is summoned.")
```

10. Properties and Setters

To elegantly manage access to an entity's attributes, guiding their use and protection:

class Elementalist:

```
    def __init__(self, element):
        self._element = element
    @property
    def element(self):
        return self._element
```

@element.setter

```
    def element(self, value):
        if value in ["Fire", "Water", "Earth", "Air"]:
            self._element = value
        else:
            print("Invalid element!")
```

Working With Decorators

1. Basic Decorator

To create a simple decorator that wraps a function:

```
def my_decorator(func):
    def wrapper():
        print("Something is happening before the function is called.")
        func()
```

Ultimate Python Cheat Sheet: Practical Python For Everyday Tasks

```
            print("Something is happening after the function is called.")
        return wrapper
    @my_decorator
    def say_hello():
        print("Hello!")
    say_hello()
```

2. Decorator with Arguments

To pass arguments to the function within a decorator:

```
    def my_decorator(func):
        def wrapper(*args, **kwargs):
            print("Before call")
            result = func(*args, **kwargs)
            print("After call")
            return result
        return wrapper
    @my_decorator
    def greet(name):
        print(f"Hello {name}")
    greet("Alice")
```

3. Using functools.wraps

To preserve the metadata of the original function when decorating:

```
    from functools import wraps
    def my_decorator(func):
        @wraps(func)
        def wrapper(*args, **kwargs):
            """Wrapper function"""
            return func(*args, **kwargs)
        return wrapper
    @my_decorator
    def greet(name):
        """Greet someone"""
        print(f"Hello {name}")
```

print(greet.__name__) # Outputs: 'greet'

print(greet.__doc__) # Outputs: 'Greet someone'

4. Class Decorator

Ultimate Python Cheat Sheet: Practical Python For Everyday Tasks

To create a decorator using a class:

```
class MyDecorator:
    def __init__(self, func):
        self.func = func

    def __call__(self, *args, **kwargs):
        print("Before call")
        self.func(*args, **kwargs)
        print("After call")

@MyDecorator
def greet(name):
    print(f"Hello {name}")
greet("Alice")
```

5. Decorator with Arguments

To create a decorator that accepts its own arguments:

```
def repeat(times):
    def decorator(func):
        @wraps(func)
        def wrapper(*args, **kwargs):
            for _ in range(times):
                func(*args, **kwargs)
        return wrapper
    return decorator
@repeat(3)
def say_hello():
    print("Hello")
say_hello()
```

6. Method Decorator

To apply a decorator to a method within a class:

```
def method_decorator(func):
    @wraps(func)
    def wrapper(self, *args, **kwargs):
        print("Method Decorator")
        return func(self, *args, **kwargs)
    return wrapper
```

class MyClass:

```
        @method_decorator
        def greet(self, name):
            print(f"Hello {name}")
obj = MyClass()
obj.greet("Alice")
```

7. Stacking Decorators

To apply multiple decorators to a single function:

```
@my_decorator
@repeat(2)
def greet(name):
    print(f"Hello {name}")
greet("Alice")
```

8. Decorator with Optional Arguments

Creating a decorator that works with or without arguments:

def smart_decorator(arg=None):

```
    def decorator(func):
        @wraps(func)
        def wrapper(*args, **kwargs):
            if arg:
                print(f"Argument: {arg}")
            return func(*args, **kwargs)
        return wrapper
    if callable(arg):
        return decorator(arg)
    return decorator
@smart_decorator
def no_args():
    print("No args")
@smart_decorator("With args")
def with_args():
    print("With args")
no_args()
with_args()
```

9. Class Method Decorator

To decorate a class method:

class MyClass:

```
    @classmethod
    @my_decorator
    def class_method(cls):
        print("Class method called")
MyClass.class_method()
```

10. Decorator for Static Method

To decorate a static method:

class MyClass:

```
    @staticmethod
    @my_decorator
    def static_method():
        print("Static method called")
MyClass.static_method()
```

Working With GraphQL

1. Setting Up a GraphQL Client

To work with GraphQL:

```
from gql import gql, Client
from gql.transport.requests import RequestsHTTPTransport
```

transport = RequestsHTTPTransport(url='https://your-graphql-endpoint.com/graphql')

client = Client(transport=transport, fetch_schema_from_transport=True)

2. Executing a Simple Query

Executing a Query:

```
query = gql('''
{
```

allWizards {

```
    id
    name
    power
  }
}
''')
result = client.execute(query)
print(result)
```

3. Executing a Query with Variables

Query with Variables:

```
query = gql('''
query GetWizards($element: String!) {
  wizards(element: $element) {
    id
    name
  }
}
''')
params = {"element": "Fire"}
result = client.execute(query, variable_values=params)
print(result)
```

4. Mutations

To create and execute a mutation:

```
mutation = gql('''
mutation CreateWizard($name: String!, $element: String!) {
  createWizard(name: $name, element: $element) {
    wizard {
      id
      name
    }
  }
}
''')
params = {"name": "Gandalf", "element": "Light"}
result = client.execute(mutation, variable_values=params)
print(result)
```

5. Handling Errors

Error handling:

```
from gql import gql, Client
from gql.transport.exceptions import TransportQueryError
try:
    result = client.execute(query)
except TransportQueryError as e:
    print(f"GraphQL Query Error: {e}")
```

Ultimate Python Cheat Sheet: Practical Python For Everyday Tasks

6. Subscriptions

Working with Subscriptions:

```
subscription = gql('''
subscription {
  wizardUpdated {
                    id
                    name
                    power
                  }
                }
                ''')
for result in client.subscribe(subscription):
                    print(result)
```

7. Fragments

Working with Fragments:

```
                query = gql('''
fragment WizardDetails on Wizard {
  name
  power
                }
query {
  allWizards {
                    ...WizardDetails
                  }
                }
                ''')
                result = client.execute(query)
                print(result)
```

8. Inline Fragments

To tailor the response based on the type of the object returned:

```
                query = gql('''
                {
  search(text: "magic") {
                    __typename
                    ... on Wizard {
```

```
            name
            power
          }
          ... on Spell {
            name
            effect
          }
        }
      }
      ''')
      result = client.execute(query)
      print(result)
```

9. Using Directives

To dynamically include or skip fields in your queries based on conditions:

```
      query = gql('''
```

query GetWizards($withPower: Boolean!) {

allWizards {

```
          name
          power @include(if: $withPower)
        }
      }
      ''')
```

params = {"withPower": True}

```
      result = client.execute(query, variable_values=params)
      print(result)
```

10. Batching Requests

To combine multiple operations into a single request, reducing network overhead:

```
      from gql import gql, Client
      from gql.transport.requests import RequestsHTTPTransport
```

transport = RequestsHTTPTransport(url='https://your-graphql-endpoint.com/graphql', use_json=True)

client = Client(transport=transport, fetch_schema_from_transport=True)

```
      query1 = gql('query { wizard(id: "1") { name } }')
      query2 = gql('query { allSpells { name } }')
      results = client.execute([query1, query2])
      print(results)
```

Ultimate Python Cheat Sheet: Practical Python For Everyday Tasks

Working With Regular Expressions

1. Basic Pattern Matching

To find a match for a pattern within a string:

```
import re
text = "Search this string for patterns."
match = re.search(r"patterns", text)
if match:
    print("Pattern found!")
```

2. Compiling Regular Expressions

To compile a regular expression for repeated use:

```
pattern = re.compile(r"patterns")
match = pattern.search(text)
```

3. Matching at the Beginning or End

To check if a string starts or ends with a pattern:

if re.match(r"^Search", text):

```
    print("Starts with 'Search'")
if re.search(r"patterns.$", text):
    print("Ends with 'patterns.'")
```

4. Finding All Matches

To find all occurrences of a pattern in a string:

```
all_matches = re.findall(r"t\w+", text)  # Finds words starting with 't'
print(all_matches)
```

5. Search and Replace (Substitution)

To replace occurrences of a pattern within a string:

```
replaced_text = re.sub(r"string", "sentence", text)
print(replaced_text)
```

6. Splitting a String

To split a string by occurrences of a pattern:

```
words = re.split(r"\s+", text)  # Split on one or more spaces
print(words)
```

7. Escaping Special Characters

To match special characters literally, escape them:

```
escaped = re.search(r"\bfor\b", text)  # \b is a word boundary
```

8. Grouping and Capturing

To group parts of a pattern and extract their values:

```
match = re.search(r"(\w+) (\w+)", text)
if match:
    print(match.group())  # The whole match
    print(match.group(1)) # The first group
```

9. Non-Capturing Groups

To define groups without capturing them:

```
match = re.search(r"(?:\w+) (\w+)", text)
if match:
    print(match.group(1))  # The first (and only) group
```

10. Lookahead and Lookbehind Assertions

To match a pattern based on what comes before or after it without including it in the result:

```
lookahead = re.search(r"\b\w+(?= string)", text)  # Word before ' string'
lookbehind = re.search(r"(?<=Search )\w+", text)  # Word after 'Search '
if lookahead:
    print(lookahead.group())
if lookbehind:
    print(lookbehind.group())
```

11. Flags to Modify Pattern Matching Behavior

To use flags like re.IGNORECASE to change how patterns are matched:

```
case_insensitive = re.findall(r"search", text, re.IGNORECASE)
print(case_insensitive)
```

12. Using Named Groups

To assign names to groups and reference them by name:

```
match = re.search(r"(?P<first>\w+) (?P<second>\w+)", text)
if match:
    print(match.group('first'))
    print(match.group('second'))
```

13. Matching Across Multiple Lines

To match patterns over multiple lines using the re.MULTILINE flag:

```
multi_line_text = "Start\nmiddle end"
matches = re.findall(r"^m\w+", multi_line_text, re.MULTILINE)
print(matches)
```

14. Lazy Quantifiers

To match as few characters as possible using lazy quantifiers (*?, +?, ??):

```
html = "<body><h1>Title</h1></body>"
```

Ultimate Python Cheat Sheet: Practical Python For Everyday Tasks

```
match = re.search(r"<.*?>", html)
if match:
    print(match.group())  # Matches '<body>'
```

15. Verbose Regular Expressions

To use re.VERBOSE for more readable regular expressions:

```
pattern = re.compile(r"""
    \b      # Word boundary
    \w+     # One or more word characters
    \s      # Space
    """, re.VERBOSE)
match = pattern.search(text)
```

Working With Strings

1. Concatenating Strings

To join strings together:

```
greeting = "Hello"
name = "Alice"
message = greeting + ", " + name + "!"
print(message)
```

2. String Formatting with str.format

To insert values into a string template:

```
message = "{}, {}. Welcome!".format(greeting, name)
print(message)
```

3. Formatted String Literals (f-strings)

To embed expressions inside string literals (Python 3.6+):

```
message = f"{greeting}, {name}. Welcome!"
print(message)
```

4. String Methods - Case Conversion

To change the case of a string:

```
s = "Python"
print(s.upper())  # Uppercase
print(s.lower())  # Lowercase
print(s.title())  # Title Case
```

5. String Methods - strip, rstrip, lstrip

To remove whitespace or specific characters from the ends of a string:

```
s = "   trim me   "
```

Ultimate Python Cheat Sheet: Practical Python For Everyday Tasks

```
print(s.strip())   # Both ends
print(s.rstrip())  # Right end
print(s.lstrip())  # Left end
```

6. String Methods - startswith, endswith

To check the start or end of a string for specific text:

```
s = "filename.txt"
print(s.startswith("file"))  # True
print(s.endswith(".txt"))    # True
```

7. String Methods - split, join

To split a string into a list or join a list into a string:

```
s = "split,this,string"
words = s.split(",")        # Split string into list
joined = " ".join(words)    # Join list into string
print(words)
print(joined)
```

8. String Methods - replace

To replace parts of a string with another string:

```
s = "Hello world"
new_s = s.replace("world", "Python")
print(new_s)
```

9. String Methods - find, index

To find the position of a substring within a string:

```
s = "look for a substring"
position = s.find("substring")  # Returns -1 if not found
index = s.index("substring")    # Raises ValueError if not found
print(position)
print(index)
```

10. String Methods - Working with Characters

To process individual characters in a string:

s = "characters"

for char in s:

```
    print(char)  # Prints each character on a new line
```

11. String Methods - isdigit, isalpha, isalnum

To check if a string contains only digits, alphabetic characters, or alphanumeric characters:

```
print("123".isdigit())   # True
```

Ultimate Python Cheat Sheet: Practical Python For Everyday Tasks

```
print("abc".isalpha())   # True
print("abc123".isalnum())# True
```

12. String Slicing

To extract a substring using slicing:

```
s = "slice me"
sub = s[2:7]  # From 3rd to 7th character
print(sub)
```

13. String Length with len

To get the length of a string:

```
s = "length"
print(len(s))  # 6
```

14. Multiline Strings

To work with strings spanning multiple lines:

```
multi = """Line one
Line two
Line three"""
print(multi)
```

15. Raw Strings

To treat backslashes as literal characters, useful for regex patterns and file paths:

path = r"C:\User\name\folder"

print(path)

Working With Web Scraping

1. Fetching Web Pages with requests

To retrieve the content of a web page:

```
import requests
url = 'https://example.com'
response = requests.get(url)
html = response.text
```

2. Parsing HTML with BeautifulSoup

To parse HTML and extract data:

```
from bs4 import BeautifulSoup
soup = BeautifulSoup(html, 'html.parser')
print(soup.prettify())  # Pretty-print the HTML
```

3. Navigating the HTML Tree

To find elements using tags:

Ultimate Python Cheat Sheet: Practical Python For Everyday Tasks

```
title = soup.title.text  # Get the page title
headings = soup.find_all('h1')  # List of all <h1> tags
```

4. Using CSS Selectors

To select elements using CSS selectors:

```
articles = soup.select('div.article')  # All elements with class 'article' inside a <d
```

5. Extracting Data from Tags

To extract text and attributes from HTML elements:

```
for article in articles:
    title = article.h2.text  # Text inside the <h2> tag
    link = article.a['href']  # 'href' attribute of the <a> tag
    print(title, link)
```

6. Handling Relative URLs

To convert relative URLs to absolute URLs:

```
from urllib.parse import urljoin
absolute_urls = [urljoin(url, link) for link in relative_urls]
```

7. Dealing with Pagination

To scrape content across multiple pages:

```
base_url = "https://example.com/page/"
for page in range(1, 6):  # For 5 pages
    page_url = base_url + str(page)
    response = requests.get(page_url)
    # Process each page's content
```

8. Handling AJAX Requests

To scrape data loaded by AJAX requests:

```
# Find the URL of the AJAX request (using browser's developer tools) and fetch it
ajax_url = 'https://example.com/ajax_endpoint'
data = requests.get(ajax_url).json()  # Assuming the response is JSON
```

9. Using Regular Expressions in Web Scraping

To extract data using regular expressions:

```
import re
emails = re.findall(r'\b[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Z|a-z]{2,}\b', html)
```

10. Respecting robots.txt

To check robots.txt for scraping permissions:

```
from urllib.robotparser import RobotFileParser
rp = RobotFileParser()
```

```
rp.set_url('https://example.com/robots.txt')
rp.read()
can_scrape = rp.can_fetch('*', url)
```

11. Using Sessions and Cookies

To maintain sessions and handle cookies:

```
session = requests.Session()
session.get('https://example.com/login')
session.cookies.set('key', 'value')  # Set cookies, if needed
response = session.get('https://example.com/protected_page')
```

12. Scraping with Browser Automation (selenium Library)

To scrape dynamic content rendered by JavaScript:

```
from selenium import webdriver
browser = webdriver.Chrome()
browser.get('https://example.com')
content = browser.page_source
# Parse and extract data using BeautifulSoup, etc.
browser.quit()
```

13. Error Handling in Web Scraping

To handle errors and exceptions:

```
try:
    response = requests.get(url, timeout=5)
    response.raise_for_status()  # Raises an error for bad status codes
```

except requests.exceptions.RequestException as e:

```
    print(f"Error: {e}")
```

14. Asynchronous Web Scraping

To scrape websites asynchronously for faster data retrieval:

```
import aiohttp
import asyncio
async def fetch(url):
    async with aiohttp.ClientSession() as session:
        async with session.get(url) as response:
            return await response.text()
urls = ['https://example.com/page1', 'https://example.com/page2']
loop = asyncio.get_event_loop()
pages = loop.run_until_complete(asyncio.gather(*(fetch(url) for url in urls)))
```

Ultimate Python Cheat Sheet: Practical Python For Everyday Tasks

15. Data Storage (CSV, Database)

To store scraped data in a CSV file or a database:

```
import csv
with open('output.csv', 'w', newline='') as file:
    writer = csv.writer(file)
    writer.writerow(['Title', 'URL'])
    for article in articles:
        writer.writerow([article['title'], article['url']])
```

Working With pip (Package Management)

1. Installing a Package

To summon a library from the vast repositories, incorporating its power into your environment:

```
pip install numpy
```

2. Listing Installed Packages

To survey the compendium of libraries that reside within your realm, noting their versions and lineage:

```
pip list
```

3. Upgrading a Package

To imbue an installed library with enhanced powers and capabilities, elevating it to its latest form:

```
pip install --upgrade numpy
```

4. Uninstalling a Package

To uninstall a package:

```
pip uninstall numpy
```

5. Searching for Packages

Searching packages:

```
pip search "data visualization"
```

6. Installing Specific Versions of a Package

To install a specific version:

```
pip install numpy==1.18.5
```

7. Generating a Requirements File

Requirements file:

```
pip freeze > requirements.txt
```

8. Installing Packages from a Requirements File

To conjure a symphony of libraries in unison, each aligned with the notations in your tome of requirements:

```
pip install -r requirements.txt
```

9. Using Virtual Environments

Create virtual Environments to manage package conflicts:

```
# Create a virtual environment named 'venv'
python -m venv venv
# Activate the virtual environment
# On Windows
.\venv\Scripts\activate
# On Unix or MacOS
source venv/bin/activate
```

10. Checking Package Dependencies

Understanding Dependencies:

```
pip show numpy
```

Working With Common Built-in Functions and Packages

1. os - Operating System Interface

To interact with the operating system:

```
import os
current_directory = os.getcwd()  # Get the current working directory
```

2. sys - System-specific Parameters and Functions

To access system-specific parameters and functions:

```
import sys
sys.exit()  # Exit the script
```

3. datetime - Basic Date and Time Types

To work with dates and times:

```
from datetime import datetime
now = datetime.now()  # Current date and time
```

4. math - Mathematical Functions

To perform mathematical operations:

```
import math
result = math.sqrt(16)  # Square root
```

5. random - Generate Pseudo-random Numbers

To generate pseudo-random numbers:

```
import random
number = random.randint(1, 10)  # Random integer between 1 and 10
```

6. json - JSON Encoder and Decoder

To parse and generate JSON data:

```
import json
json_string = json.dumps({'name': 'Alice', 'age': 30})  # Dictionary to JSON string
```

Ultimate Python Cheat Sheet: Practical Python For Everyday Tasks

7. re - Regular Expressions

To work with regular expressions:

```
import re
match = re.search('Hello', 'Hello, world!')  # Search for 'Hello' in the string
```

8. urllib - URL Handling Modules

To work with URLs:

```
from urllib.request import urlopen
content = urlopen('http://example.com').read()  # Fetch the content of a webpage
```

9. http - HTTP Modules

To create HTTP servers and work with HTTP requests:

```
from http.server import HTTPServer, BaseHTTPRequestHandler
```

class SimpleHTTPRequestHandler(BaseHTTPRequestHandler):

```
    def do_GET(self):
        self.send_response(200)
        self.send_header('Content-type', 'text/html')
        self.end_headers()
        self.wfile.write(b'<html><head><title>Python HTTP Server</title></head>')
        self.wfile.write(b'<body><h1>Hello from a simple Python HTTP server!</h1></bo
```

def run(server_class=HTTPServer, handler_class=SimpleHTTPRequestHandler):

```
    server_address = ('', 8000)  # Serve on all addresses, port 8000
    httpd = server_class(server_address, handler_class)
    print("Server starting on port 8000...")
    httpd.serve_forever()
if __name__ == '__main__':
    run()
```

10. subprocess - Subprocess Management

To spawn new processes and connect to their input/output/error pipes:

```
import subprocess
subprocess.run(['ls', '-l'])  # Run the 'ls -l' command
```

11. socket - Low-level Networking Interface

To create network clients and servers:

```
import socket
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)  # Create a TCP/IP socket
```

12. threading - Thread-based Parallelism

To manage concurrent execution of code:

Ultimate Python Cheat Sheet: Practical Python For Everyday Tasks

```
import threading
def worker():
    print("Worker thread executing")
thread = threading.Thread(target=worker)
thread.start()
```

13. multiprocessing - Process-based Parallelism

To manage concurrent processes:

```
from multiprocessing import Process
def worker():
    print("Worker process")
p = Process(target=worker)
p.start()
```

14. argparse - Parser for Command-line Options, Arguments, and Sub-commands

To parse command-line arguments:

```
import argparse
parser = argparse.ArgumentParser(description="Process some integers.")
args = parser.parse_args()
```

15. logging - Logging Facility

To log messages (debug, info, warning, error, and critical):

```
import logging
logging.warning('This is a warning message')
```

16. unittest - Unit Testing Framework

To create and run unit tests:

```
import unittest
```

class TestStringMethods(unittest.TestCase):

```
    def test_upper(self):
        self.assertEqual('foo'.upper(), 'FOO')
```

17. pathlib - Object-oriented Filesystem Paths

To work with filesystem paths in an object-oriented way:

```
from pathlib import Path
p = Path('.')
```

18. functools - Higher-order Functions and Operations on Callable Objects

To use higher-order functions and operations on callable objects:

```
from functools import lru_cache
@lru_cache(maxsize=None)
```

```
def fib(n):
    if n < 2:
        return n
    return fib(n-1) + fib(n-2)
```

19. collections - Container Data Types

To use specialized container data types (deque, Counter, OrderedDict, etc.):

```
from collections import Counter
c = Counter('hello world')
```

20. itertools - Functions Creating Iterators for Efficient Looping

To construct and use iterators for efficient looping:

```
import itertools
```

for combination in itertools.combinations('ABCD', 2):

```
    print(combination)
```

21. hashlib - Secure Hash and Message Digest Algorithms

To hash data:

```
import hashlib
hash_object = hashlib.sha256(b'Hello World')
hex_dig = hash_object.hexdigest()
```

22. csv - CSV File Reading and Writing

To read from and write to CSV files:

```
import csv
with open('file.csv', mode='r') as infile:
    reader = csv.reader(infile)
```

23. xml.etree.ElementTree - The ElementTree XML API

To parse and create XML data:

```
import xml.etree.ElementTree as ET
tree = ET.parse('file.xml')
root = tree.getroot()
```

24. sqlite3 - DB-API 2.0 Interface for SQLite Databases

To interact with SQLite databases:

```
import sqlite3
conn = sqlite3.connect('example.db')
```

25. tkinter - GUI Toolkit

To create GUI applications:

```
import tkinter as tk
```

Ultimate Python Cheat Sheet: Practical Python For Everyday Tasks

```
root = tk.Tk()
```

26. pickle - Python Object Serialization

To serialize and deserialize Python object structures:

```
import pickle
serialized_obj = pickle.dumps(obj)
```

27. io - Core Tools for Working with Streams

To handle streams (file-like objects):

from io import StringIO

f = StringIO("some initial text data")

28. time - Time Access and Conversions

To access time-related functions:

```
import time
time.sleep(1)  # Sleep for 1 second
```

29. calendar - General Calendar-related Functions

To work with calendars:

```
import calendar
print(calendar.month(2023, 1))  # Print the calendar for January 2023
```

30. queue - A Synchronized Queue Class

To manage a queue, useful in multithreaded programming:

```
from queue import Queue
q = Queue()
```

31. shutil - High-level File Operations

To perform high-level file operations, like copying and archiving:

```
import shutil
shutil.copyfile('source.txt', 'dest.txt')
```

32. glob - Unix Style Pathname Pattern Expansion

To find files matching a specified pattern:

```
import glob
for file in glob.glob("*.txt"):
    print(file)
```

33. tempfile - Generate Temporary Files and Directories

To create temporary files and directories:

```
import tempfile
temp = tempfile.TemporaryFile()
```

34. bz2 - Support for Bzip2 Compression

To compress and decompress data using bzip2 compression:

```
import bz2
compressed = bz2.compress(b'your data here')
```

35. gzip - Support for Gzip Compression

To compress and decompress data using gzip compression:

```
import gzip
with gzip.open('file.txt.gz', 'wt') as f:
    f.write('your data here')
```

36. ssl - TLS/SSL Wrapper for Socket Objects

To handle TLS/SSL encryption and peer authentication for network sockets:

```
import ssl
ssl.wrap_socket(sock)
```

37. imaplib - IMAP4 Protocol Client

To access and manipulate mail over IMAP4:

```
import imaplib
mail = imaplib.IMAP4_SSL('imap.example.com')
```

38. smtplib - SMTP Protocol Client

To send mail using the Simple Mail Transfer Protocol (SMTP):

```
import smtplib
server = smtplib.SMTP('smtp.example.com', 587)
```

39. email - Managing Email Messages

To manage email messages, including MIME and other RFC 2822-based message documents:

```
from email.message import EmailMessage
msg = EmailMessage()
```

40. base64 - Base16, Base32, Base64, Base85 Data Encodings

To encode and decode data using Base64:

```
import base64
encoded_data = base64.b64encode(b'data to encode')
```

41. difflib - Helpers for Computing Deltas

To compare sequences and produce human-readable diffs:

```
import difflib
diff = difflib.ndiff('one\ntwo\nthree\n'.splitlines(keepends=True),
                     'ore\ntree\nemu\n'.splitlines(keepends=True))
print(''.join(diff))
```

42. gettext - Multilingual Internationalization Services

Ultimate Python Cheat Sheet: Practical Python For Everyday Tasks

To internationalize your Python programs:

```
import gettext
gettext.install('myapp')
```

43. locale - Internationalization Services

To access a database of culture-specific data formats:

```
import locale
locale.setlocale(locale.LC_ALL, '')
```

44. secrets - Generate Secure Random Numbers for Managing Secrets

To generate secure random numbers for managing secrets, such as tokens or passwords:

```
import secrets
secure_token = secrets.token_hex(16)
```

45. uuid - UUID Objects According to RFC 4122

To generate universally unique identifiers (UUIDs):

```
import uuid
unique_id = uuid.uuid4()
```

46. html - HyperText Markup Language Support

To handle and manipulate HTML entities:

```
import html
escaped = html.escape('<a href="https://example.com">link</a>')
```

47. ftplib - FTP Protocol Client

To interact with and transfer files over the FTP protocol:

```
from ftplib import FTP
ftp = FTP('ftp.example.com')
```

48. tarfile - Read and Write Tar Archive Files

To work with tar archive files, allowing you to archive and compress/decompress:

import tarfile

with tarfile.open('sample.tar.gz', 'w:gz') as tar:

```
    tar.add('sample.txt')
```