Python Cheatsheet

This cheatsheet provides a quick reference for essential Python methods and functions, designed for easy printing and everyday use.

1. Basic Syntax and Data Types

Variables and Assignment

```
 x = 10          # Assigns an integer value to x
y = 3.14         # Assigns a float value to y
s = "Hello"      # Assigns a string value to s
b = True         # Assigns a boolean value to b
l = [1, 2, 3]    # Assigns a list (ordered, mutable collection) to l
t = (1, 2)       # Assigns a tuple (ordered, immutable collection) to t
d = {\'a\': 1}     # Assigns a dictionary (key-value pairs) to d
se = {1, 2}      # Assigns a set (unordered collection of unique items) to se
```

Basic Operations

```
 x = 10
y = 3
print(x + y)  # Addition: Output: 13
print(x / y)  # Division: Output: 3.333...
print(x // y) # Floor Division: Output: 3 (discards fractional part)
print(x % y)  # Modulo: Output: 1 (remainder of division)
print(x == y) # Equality Check: Output: False
print(x > y)  # Greater Than Check: Output: True
print(True and False) # Logical AND: Output: False
```

2. Control Flow

Conditional Statements

```
 x = 10
if x > 5:
    print("x is greater than 5") # Executes if x is greater than 5
elif x == 5:
    print("x is 5") # Executes if x is equal to 5 (and not greater than 5)
else:
    print("x is less than 5") # Executes if neither of the above conditions are
met
```

Loops

```
 # For loop: Iterates over a sequence (e.g., range, list, string)
for i in range(3):
    print(i) # Prints numbers from 0 to 2

fruits = ["apple", "banana", "cherry"]
for fruit in fruits:
    print(fruit) # Prints each fruit in the list

# While loop: Executes a block of code repeatedly as long as a condition is
true
count = 0
while count < 3:
    print(count)
    count += 1 # Increments count; prints numbers from 0 to 2

# break and continue: Control loop execution
for i in range(5):
    if i == 2:
        continue # Skips the rest of the current iteration if i is 2, moves to
next iteration
    if i == 4:
        break    # Terminates the loop entirely if i is 4
    print(i) # Prints 0, 1, 3
```

3. Functions

Defining a Function

```
def greet(name, greeting="Hello"):
    """Greets a person with a customizable greeting.""" # Docstring explains
function purpose
    return f"{greeting}, {name}!" # Returns a formatted string

print(greet("Alice"))            # Calls function with one argument, uses
default greeting
print(greet("Bob", "Hi"))        # Calls function with two arguments, overrides
default greeting
```

Lambda Functions

```
add = lambda a, b: a + b # Defines a small anonymous function that adds two
numbers
print(add(2, 3)) # Calls the lambda function; Output: 5
```

4. Data Structures

Lists

```
 my_list = [1, 2, 3]
my_list.append(4)         # Adds 4 to the end; my_list is now [1, 2, 3, 4]
my_list.insert(1, 5)      # Inserts 5 at index 1; my_list is now [1, 5, 2, 3,
4]
my_list.remove(2)         # Removes the first occurrence of 2; my_list is now
[1, 5, 3, 4]
popped = my_list.pop()    # Removes and returns the last item (4); popped is 4,
my_list is [1, 5, 3]
my_list.sort()            # Sorts the list in-place; my_list is now [1, 3, 5]
length = len(my_list)     # Returns the number of items (3); length is 3
item = my_list[0]         # Accesses item at index 0 (1); item is 1
sub_list = my_list[0:2]   # Slices from index 0 up to (but not including) 2;
sub_list is [1, 3]
sub_list_end = my_list[1:]  # Slices from index 1 to the end; sub_list_end is
[3, 5]
sub_list_start = my_list[:2] # Slices from the beginning up to (but not
including) 2; sub_list_start is [1, 3]
sub_list_drop_last = my_list[:-1] # Slices to drop the last element;
sub_list_drop_last is [1, 3]
sub_list_step = my_list[::2] # Slices with a step of 2; sub_list_step is [1, 5]
reversed_list = my_list[::-1] # Reverses the list using slicing; reversed_list
is [5, 3, 1]
# Slicing semantics: [start:stop:step]. Stop index is exclusive.

list2 = [6, 7]
my_list.extend(list2)     # Appends all items from list2; my_list is now [1, 3,
5, 6, 7]
my_list.clear()           # Removes all items; my_list is now []

original = [1, 2, 3]
copy = original.copy()    # Creates a shallow copy; copy is [1, 2, 3]
count = original.count(1) # Counts occurrences of 1 (1); count is 1
index = original.index(2) # Returns index of 2 (1); index is 1
original.reverse()        # Reverses the list in-place; original is now [3, 2,
1]

items = [('apple', 3), ('banana', 1), ('cherry', 2)]
sorted_by_name = sorted(items, key=lambda x: x[0]) # Sorts by item name;
[('apple', 3), ('banana', 1), ('cherry', 2)]
sorted_by_count_desc = sorted(items, key=lambda x: x[1], reverse=True) # Sorts
by count descending; [('apple', 3), ('cherry', 2), ('banana', 1)]

# list.sort() modifies in-place
mutable_list = [3, 1, 2]
mutable_list.sort(key=lambda x: x) # Sorts in-place; mutable_list is [1, 2, 3]
```

Tuples

- Immutable version of lists

```
my_tuple = (1, 2, 3)
item = my_tuple[0]        # Accesses item at index 0 (1); item is 1
sub_tuple = my_tuple[0:2] # Slices from index 0 up to (but not including) 2; sub_tuple is (1, 2)
reversed_tuple = my_tuple[::-1] # Reverses the tuple using slicing; reversed_tuple is (3, 2, 1)
# my_tuple[0] = 5         # Attempting to modify an item in a tuple raises a TypeError
```

Dictionaries

```
my_dict = {\'a\': 1, \'b\': 2}
my_dict[\'c\'] = 3          # Adds a new key-value pair; my_dict is {\'a\': 1, \'b\': 2, \'c\': 3}
value = my_dict.get(\'d\', 0) # Gets value for key \'d\', returns 0 if not found; value is 0
keys = my_dict.keys()     # Returns a view of all keys; keys is dict_keys([\'a\', \'b\', \'c\'])
values = my_dict.values() # Returns a view of all values; values is dict_values([1, 2, 3])
items = my_dict.items()   # Returns a view of all key-value pairs; items is dict_items([(\'a\', 1), (\'b\', 2), (\'c\', 3)])

my_dict.clear()           # Removes all items from the dictionary; my_dict is {}

original_dict = {\'x\': 10, \'y\': 20}
copy_dict = original_dict.copy() # Creates a shallow copy; copy_dict is {\'x\': 10, \'y\': 20}
popped_value = original_dict.pop(\'x\') # Removes \'x\' and returns its value (10); original_dict is {\'y\': 20}

original_dict[\'z\'] = 30
popped_item = original_dict.popitem() # Removes and returns an arbitrary (key, value) pair (e.g., (\'z\', 30)); original_dict is {\'y\': 20}

other_dict = {\'y\': 25, \'w\': 35}
original_dict.update(other_dict) # Updates with key-value pairs from other_dict; original_dict is {\'y\': 25, \'w\': 35}
```

Sets

```
my_set = {1, 2}
my_set.add(3)             # Adds an element to the set; my_set is {1, 2, 3}
my_set.remove(2)          # Removes an element from the set; my_set is {1, 3}

other_set = {3, 4}
union_set = my_set.union(other_set)         # Returns a new set with all elements from both sets; union_set is {1, 3, 4}
intersection_set = my_set.intersection(other_set) # Returns a new set with common elements; intersection_set is {3}
```

5. String Methods

```
 my_string = "  Hello World  "
print(my_string.lower())        # Converts string to lowercase
print(my_string.upper())        # Converts string to uppercase
print(my_string.strip())        # Removes leading/trailing whitespace

text = "apple,banana,cherry"
parts = text.split(",")         # Splits string by delimiter into a list; parts
is ["apple", "banana", "cherry"]
joined = "-".join(parts)        # Joins elements of an iterable with the string
as a delimiter; joined is "apple-banana-cherry"

replaced = my_string.replace("World", "Python") # Replaces all occurrences of a
substring; replaced is "  Hello Python  "
index = my_string.find("World") # Returns the lowest index of the substring if
found, -1 otherwise; index is 8
sub_string = my_string[3:8] # Slices from index 3 up to (but not including) 8;
sub_string is "Hello"
reversed_string = my_string[::-1] # Reverses the string using slicing;
reversed_string is "  dlroW olleH  "

print(my_string.startswith("  Hello")) # Checks if string starts with the
specified prefix; Output: True
print(my_string.endswith("World  "))   # Checks if string ends with the
specified suffix; Output: True
print("banana".count("a"))      # Returns the number of non-overlapping
occurrences of substring; Output: 3
print("123".isdigit())          # Returns True if all characters in the string
are digits; Output: True
print("abc".isalpha())          # Returns True if all characters in the string
are alphabetic; Output: True
print("abc123".isalnum())       # Returns True if all characters are
alphanumeric; Output: True
print("   ".isspace())          # Returns True if all characters are
whitespace; Output: True
print("hello world".title())    # Converts the first character of each word to
uppercase; Output: "Hello World"
print("hello world".capitalize()) # Converts the first character of the string
to uppercase; Output: "Hello world"
print("42".zfill(5))            # Pads the string on the left with zeros to
fill a specified width; Output: "00042"
```

6. File I/O

```
 # Writing to a file: Always specify encoding for text files
with open("example.txt", "w", encoding="utf-8") as f:
    f.write("Hello, world!\n") # Writes the string to the file

# Reading a file: Always specify encoding for text files
with open("example.txt", "r", encoding="utf-8") as f:
    content = f.read()       # Reads the entire content of the file into a
string
    print(f"Read content: {content.strip()}")

# Appending to a file
with open("example.txt", "a", encoding="utf-8") as f:
    f.write("New line.\n") # Appends a new line to the file

# Reading lines from a file
with open("example.txt", "r", encoding="utf-8") as f:
    lines = f.readlines()    # Reads all lines into a list of strings
    print(f"Read lines: {lines}")

import csv

# Handling newlines for CSV files (Windows compatibility) and writing
with open("data.csv", "w", newline="", encoding="utf-8") as csvfile:
    writer = csv.writer(csvfile)
    writer.writerow(["header1", "header2"])
    writer.writerow(["value1", "value2"])
print("CSV file written with proper newline handling.")
```

7. Error Handling

```
 try:
    result = 10 / 0 # Code that might raise an error
except ZeroDivisionError as e:
    print(f"Error: {e}") # Handles specific ZeroDivisionError
except TypeError as e:
    print(f"Type Error: {e}") # Handles specific TypeError
else:
    print("No error occurred") # Executes if no exception was raised in the try
block
finally:
    print("Execution finished") # Always executes, regardless of whether an
exception occurred

try:
    int("abc") # This will raise a ValueError
except ValueError as e:
    print(f"Value Error: {e}") # Handles ValueError, e.g., invalid literal for
int() with base 10: 'abc'

# Custom Exception
class CustomError(Exception):
    """A custom exception for demonstration."""
    pass

def validate_age(age):
    if not isinstance(age, int) or age < 0:
        raise ValueError("Age must be a non-negative integer.")
    if age > 120:
        raise CustomError("Age seems unusually high.")
    print(f"Age {age} is valid.")

try:
    validate_age(150)
except CustomError as e:
    print(f"Custom Error caught: {e}")
except ValueError as e:
    print(f"Validation Error: {e}")
```

8. Modules and Packages

```
 import math # Imports the entire math module
import os   # Imports the entire os module for interacting with the operating system
from datetime import datetime # Imports only the datetime class from the datetime module

# Using functions from modules
print(math.sqrt(16)) # Calculates the square root using a function from the math module; Output: 4.0
print(os.getcwd())   # Gets the current working directory using a function from the os module; Output: Current working directory
print(datetime.now())# Gets the current date and time using the datetime class; Output: Current date and time
```

9. Useful Built-in Functions

```
 print("Hello")            # Prints output to the console
name = input("Enter name: ") # Prompts the user for input and returns it as a
string

my_list = [1, 2, 3]
length = len(my_list)     # Returns the length (number of items) of an object;
length is 3

print(type(my_list))      # Returns the type of an object; Output: <class
'list'>

num_str = "10"
num_int = int(num_str)    # Converts a value to an integer; num_int is 10

for i in range(3):        # Generates a sequence of numbers (0, 1, 2) for
iteration
    print(i)

numbers = [1, 2, 3]
squared = list(map(lambda x: x*x, numbers)) # Applies a function to all items
in an input list; squared is [1, 4, 9]
even = list(filter(lambda x: x % 2 == 0, numbers)) # Constructs an iterator
from elements for which a function returns true; even is [2]

list1 = [1, 2]
list2 = ["a", "b"]
zipped = list(zip(list1, list2)) # Aggregates elements from multiple iterables;
zipped is [(1, 'a'), (2, 'b')]

maximum = max(my_list)    # Returns the largest item in an iterable; maximum is
3
minimum = min(my_list)    # Returns the smallest item in an iterable; minimum
is 1
total = sum(my_list)      # Sums the items of an iterable; total is 6

print(abs(-5))            # Returns the absolute value of a number; Output: 5
print(all([True, True]))  # Returns True if all items in an iterable are true;
Output: True
print(any([True, False])) # Returns True if any item in an iterable is true;
Output: True

# print(dir(my_list))     # Returns a list of the specified object's properties
and methods

for i, item in enumerate(my_list):
    print(f"{i}: {item}") # Adds counters to an iterable; Output: 0: 1, 1: 2,
2: 3

# WARNING: eval() is dangerous. Use ast.literal_eval for safe evaluation of
literals.
from ast import literal_eval
result = literal_eval("[1, 2, 3]") # Safely evaluates a literal structure like
[1, 2, 3]
print(f"Safely evaluated: {result}")
# help(str)               # Invokes the built-in help system for an object

print(isinstance(my_list, list)) # Checks if an object is an instance of a
```

```
class; Output: True
print(round(3.14159, 2))  # Rounds a number to a given precision; Output: 3.14

sorted_list = sorted([3, 1, 2]) # Returns a new sorted list from the items in
an iterable; sorted_list is [1, 2, 3]
```

10. List Comprehensions

```
 squares = [x**2 for x in range(10) if x % 2 == 0] # Creates a list of squares
for even numbers from 0 to 9; squares is [0, 4, 16, 36, 64]
print(squares)

even_numbers = [x for x in range(5) if x % 2 == 0] # Creates a list of even
numbers from 0 to 4; even_numbers is [0, 2, 4]
print(even_numbers)

matrix = [[1, 2], [3, 4]]
flattened = [num for row in matrix for num in row] # Flattens a 2D list into a
1D list; flattened is [1, 2, 3, 4]
print(flattened)
```

11. Object-Oriented Programming (OOP)

Class Definition

```
 class Dog: # Defines a new class named Dog
    def __init__(self, name, age): # Constructor method, initializes object
attributes
        self.name = name # Instance attribute for the dog's name
        self.age = age   # Instance attribute for the dog's age

    def bark(self): # Instance method, defines a behavior for Dog objects
        return f"{self.name} says Woof!"

my_dog = Dog("Buddy", 3) # Creates an instance of the Dog class
print(my_dog.bark()) # Calls the bark method on the my_dog object; Output:
Buddy says Woof!

# Inheritance: Labrador inherits from Dog, gaining its attributes and methods
class Labrador(Dog): # Defines a new class Labrador that inherits from Dog
    def __init__(self, name, age, color): # Constructor for Labrador
        super().__init__(name, age) # Calls the constructor of the parent class
(Dog)
        self.color = color # Adds a new attribute specific to Labrador

    def swim(self): # New method specific to Labrador
        return f"{self.name} is swimming."

my_lab = Labrador("Lucy", 5, "yellow") # Creates an instance of the Labrador
class
print(my_lab.bark())  # Calls inherited method; Output: Lucy says Woof!
print(my_lab.swim())  # Calls Labrador-specific method; Output: Lucy is
swimming.
```

References

[1] Python Cheat Sheet & Quick Reference. (n.d.). Retrieved from https://quickref.me/python.html [2] DataCamp. (2022, November 20). Python Cheat Sheet for Beginners. Retrieved from https://www.datacamp.com/cheat-sheet/getting-started-with-python-cheat-sheet [3] Python Cheatsheet. (n.d.). Retrieved from https://www.pythoncheatsheet.org/ [4] Zero To Mastery. (n.d.). The Best Python Cheat Sheet. Retrieved from https://zerotomastery.io/cheatsheets/python-cheat-sheet/ [5] Stackademic. (2024, January 29). Ultimate Python Cheat Sheet: Practical Python For Everyday Tasks. Retrieved from https://blog.stackademic.com/ultimate-python-cheat-sheet-practical-python-for-everyday-tasks-c267c1394ee8 [6] Dataquest. (n.d.). Python Cheat Sheet. Retrieved from https://www.dataquest.io/cheat-sheet/python-cheat-sheet/

- `str.startswith(prefix)`: Checks if string starts with prefix
- `str.endswith(suffix)`: Checks if string ends with suffix
- `str.count(substring)`: Returns number of occurrences of substring
- `str.isdigit()`: Checks if all characters are digits
- `str.isalpha()`: Checks if all characters are alphabetic
- `str.isalnum()`: Checks if all characters are alphanumeric
- `str.isspace()`: Checks if all characters are whitespace
- `str.title()`: Converts first letter of each word to uppercase
- `str.capitalize()`: Converts first letter of string to uppercase
- `str.zfill(width)`: Pads string with zeros on the left to fill width
- `list.extend(iterable)`: Extends list by appending elements from the iterable
- `list.clear()`: Removes all items from the list
- `list.copy()`: Returns a shallow copy of the list
- `list.count(value)`: Returns number of occurrences of value
- `list.index(value, start, end)`: Returns first index of value
- `list.reverse()`: Reverses the list in-place
- `dict.clear()`: Removes all items from the dictionary
- `dict.copy()`: Returns a shallow copy of the dictionary
- `dict.pop(key, default)`: Removes specified key and returns the corresponding value
- `dict.popitem()`: Removes and returns a (key, value) pair
- `dict.update(other)`: Updates the dictionary with the key/value pairs from other
- `abs()`: Returns the absolute value of a number
- `all()`: Returns True if all items in an iterable are true
- `any()`: Returns True if any item in an iterable is true

- `dir()`: Returns a list of the specified object's properties and methods
- `enumerate()`: Takes a collection (e.g. a tuple) and returns it as an enumerate object
- `eval()`: Evaluates and executes an expression
- `help()`: Invokes the built-in help system
- `isinstance()`: Checks if an object is an instance of a class
- `round()`: Rounds a number
- `sorted()`: Returns a new sorted list from the items in an iterable
- `sum()`: Sums the items of an iterable

12. Context Managers

Context managers simplify resource management (e.g., files, locks) by ensuring resources are properly acquired and released.

```
 # Using with statement for file handling
with open("my_file.txt", "w", encoding="utf-8") as f:
    f.write("Hello from context manager!")

# Custom context manager using contextlib
from contextlib import contextmanager

@contextmanager
def tag(name):
    print(f"<{name}>")
    yield
    print(f"</{name}>")

with tag("h1"):
    print("This is a heading")
```

13. Pathlib

`pathlib` offers an object-oriented way to handle filesystem paths, replacing `os.path`.

```python
from pathlib import Path

# Current working directory
current_dir = Path.cwd()
print(f"Current directory: {current_dir}")

# Creating a path object
file_path = Path("data") / "report.csv"
print(f"File path: {file_path}")

# Checking if path exists
if file_path.exists():
    print(f"{file_path} exists.")

# Reading and writing text files
# file_path.write_text("Hello, Pathlib!")
# content = file_path.read_text()

# Iterating over files
for p in Path(".").glob("*.md"):
    print(f"Markdown file: {p.name}")
```

14. Logging

Logging is crucial for tracking events that happen when software runs. It's preferred over `print()` for production applications.

```python
import logging

# Basic configuration
logging.basicConfig(level=logging.INFO, format=\"%(asctime)s - %(levelname)s - %(message)s\")

# Get a logger instance
log = logging.getLogger(__name__)

log.info("This is an informational message.")
log.warning("This is a warning message.")
log.error("This is an error message.")

# Example with variables
def divide(a, b):
    try:
        log.info(f"Attempting to divide {a} by {b}")
        result = a / b
        log.info(f"Division successful: {result}")
        return result
    except ZeroDivisionError:
        log.error("Attempted to divide by zero!")
        return None

divide(10, 2)
divide(10, 0)
```

15. Virtual Environments & Packaging

Virtual environments create isolated Python environments for projects, managing dependencies without conflicts. Packaging involves distributing your code.

```
# Create a virtual environment
python -m venv .venv

# Activate the virtual environment (Linux/macOS)
source .venv/bin/activate

# Activate the virtual environment (Windows PowerShell)
.venv\Scripts\Activate.ps1

# Install packages
pip install requests beautifulsoup4

# Save installed packages to a requirements file
pip freeze > requirements.txt

# Install packages from a requirements file
pip install -r requirements.txt

# Deactivate the virtual environment
deactivate
```

pipx for CLIs: For installing Python command-line applications in isolated environments, `pipx` is recommended to avoid polluting global site-packages.

```
pip install pipx
pipx install black
```

16. Testing with Pytest

Testing ensures your code works as expected and helps prevent regressions. Pytest is a popular testing framework.

```python
# test_example.py
def add(a, b):
    return a + b

def test_add():
    assert add(1, 2) == 3, "Should be 3"
    assert add(0, 0) == 0, "Should be 0"
    assert add(-1, 1) == 0, "Should be 0"

# To run tests from the terminal:
# pytest
# pytest -v (for verbose output)

# Assert patterns:
# assert value == expected
# assert value in collection
# assert "substring" in string
# assert isinstance(obj, type)
# with pytest.raises(ExceptionType): # For testing exceptions
#     do_something_that_raises_exception()
```

17. Typing & Dataclasses

Type hints improve code readability and help catch errors during development. Dataclasses provide a concise way to create classes primarily used to store data.

```python
from typing import List, Dict, Tuple, Optional
from dataclasses import dataclass

# Type hints for functions
def greet_person(name: str, age: int) -> str:
    return f"Hello {name}, you are {age} years old."

# Type hints for variables
users: List[str] = ["Alice", "Bob"]
user_data: Dict[str, int] = {"Alice": 30, "Bob": 25}

# Dataclasses
@dataclass
class Product:
    name: str
    price: float
    quantity: int = 0 # Default value
    is_available: bool = True

# Immutable dataclass (frozen=True)
@dataclass(frozen=True)
class Point:
    x: float
    y: float

# Usage
apple = Product("Apple", 1.0, 10)
print(apple) # Output: Product(name=\'Apple\', price=1.0, quantity=10,
is_available=True)

p = Point(1.0, 2.0)
# p.x = 3.0 # This would raise a FrozenInstanceError
```

18. Comprehensions & Generators

Comprehensions provide a concise way to create lists, dictionaries, and sets. Generators are iterators that generate items one by one, saving memory.

```python
# Dictionary Comprehension
sq_dict = {x: x*x for x in range(5)} # {0: 0, 1: 1, 2: 4, 3: 9, 4: 16}

# Set Comprehension
even_set = {x for x in range(10) if x % 2 == 0} # {0, 2, 4, 6, 8}

# Generator Expression (creates an iterator)
gen_exp = (x*x for x in range(5))
print(next(gen_exp)) # 0
print(next(gen_exp)) # 1

# Generator Function with yield
def count_up_to(max):
    count = 1
    while count <= max:
        yield count
        count += 1

counter = count_up_to(3)
print(next(counter)) # 1
print(next(counter)) # 2

# itertools essentials
from itertools import chain, groupby, product

# chain: combines iterables
chained = list(chain([1, 2], [3, 4])) # [1, 2, 3, 4]

# groupby: groups consecutive identical elements
data = [("A", 1), ("A", 2), ("B", 3)]
for key, group in groupby(data, lambda x: x[0]):
    print(f"{key}: {list(group)}") # A: [("A", 1), ("A", 2)], B: [("B", 3)]

# product: Cartesian product of input iterables
prod = list(product("AB", "12")) # [("A", "1"), ("A", "2"), ("B", "1"), ("B",
"2")]
```

19. Unpacking Patterns

Unpacking allows assigning elements from iterables to multiple variables in a single statement, including extended unpacking with `*`.

```python
# Basic unpacking
a, b, c = [1, 2, 3] # a=1, b=2, c=3

# Extended unpacking (Python 3+)
first, *middle, last = [1, 2, 3, 4, 5] # first=1, middle=[2, 3, 4], last=5

# Function arguments unpacking
def func(x, y, z):
    print(x, y, z)

args = [1, 2, 3]
func(*args) # Unpacks list into positional arguments

kwargs = {'x': 10, 'y': 20, 'z': 30}
func(**kwargs) # Unpacks dictionary into keyword arguments

# Merging dictionaries (Python 3.5+)
dict1 = {'a': 1, 'b': 2}
dict2 = {'b': 3, 'c': 4}
merged_dict = {**dict1, **dict2} # {'a': 1, 'b': 3, 'c': 4}

# Merging dictionaries (Python 3.9+)
merged_dict_39 = dict1 | dict2 # {'a': 1, 'b': 3, 'c': 4}
```

20. String Formatting Mini-Language

Python's f-strings (formatted string literals) support a powerful mini-language for formatting values.

```python
import datetime

value = 12345.6789
print(f"Number: {value:,.2f}") # Output: Number: 12,345.68 (comma as thousands separator, 2 decimal places)

percentage = 0.75
print(f"Percentage: {percentage:.1%}") # Output: Percentage: 75.0%

date_obj = datetime.datetime(2025, 9, 14, 10, 30, 0)
print(f"Date: {date_obj:%Y-%m-%d %H:%M}") # Output: Date: 2025-09-14 10:30 (formatted date)

text = "hello"
print(f"{text:>10}") # Output:      hello (right-aligned, width 10)
print(f"{text:<10}") # Output: hello      (left-aligned, width 10)
print(f"{text:^10}") # Output:   hello    (center-aligned, width 10)
```

21. Datetime & Timezones

Working with dates and times, especially across different timezones, requires careful handling.

```
from datetime import datetime, timedelta, timezone

# Current UTC time
now_utc = datetime.now(timezone.utc)
print(f"UTC Now: {now_utc}")

# Local timezone (requires `zoneinfo` module in Python 3.9+ or `pytz`)
# from zoneinfo import ZoneInfo
# local_tz = ZoneInfo("America/New_York")
# now_local = now_utc.astimezone(local_tz)
# print(f"Local Now: {now_local}")

# Creating a specific datetime
d = datetime(2025, 9, 14, 10, 30, 0, tzinfo=timezone.utc)
print(f"Specific UTC: {d}")

# Time differences
one_day = timedelta(days=1)
tomorrow = now_utc + one_day
print(f"Tomorrow (UTC): {tomorrow}")

# Formatting datetime objects
print(f"Formatted: {now_utc.strftime(\"%Y-%m-%d %H:%M:%S\")}")
```

22. HTTP & JSON

Interacting with web services often involves making HTTP requests and handling JSON data.

```
 import requests
import json

# Make an HTTP GET request and parse JSON response
try:
    response = requests.get("https://jsonplaceholder.typicode.com/todos/1")
    response.raise_for_status() # Raise an exception for HTTP errors
    todo_item = response.json() # Parses JSON response into a Python dictionary
    print(f"Fetched TODO: {todo_item[\"title\"]}")
except requests.exceptions.RequestException as e:
    print(f"HTTP Request failed: {e}")

# Convert Python dictionary to JSON string
data = {
    "name": "Alice",
    "age": 30,
    "isStudent": False
}
json_string = json.dumps(data, indent=2) # Converts dict to JSON string with 2-space indentation
print("\nPython dict to JSON:")
print(json_string)

# Convert JSON string to Python dictionary
json_data = '{"product": "Laptop", "price": 1200}'
python_dict = json.loads(json_data)

print(f"\nJSON string to Python dict: {python_dict[\"product\"]}")
```

23. Common Collections

The `collections` module provides specialized container datatypes offering alternatives to Python’s general purpose built-in containers.

```python
from collections import Counter, defaultdict, deque, namedtuple

# Counter: A dict subclass for counting hashable objects
counts = Counter(["apple", "banana", "apple", "orange", "banana", "apple"])
print(f"Counts: {counts}") # Output: Counter({\'apple\': 3, \'banana\': 2,
\'orange\': 1})
print(f"Most common: {counts.most_common(1)}") # Output: [(\'apple\', 3)]

# defaultdict: A dict subclass that calls a factory function to supply missing
values
dd = defaultdict(int) # Default value for missing keys will be 0
dd["a"] += 1
dd["b"] += 1
print(f"Defaultdict: {dd}") # Output: defaultdict(<class \'int\'>, {\'a\': 1,
\'b\': 1})

# deque: A list-like container with fast appends and pops on either end
d = deque([1, 2, 3])
d.appendleft(0) # d is now deque([0, 1, 2, 3])
d.append(4)     # d is now deque([0, 1, 2, 3, 4])
d.popleft()     # Returns 0, d is now deque([1, 2, 3, 4])
d.pop()         # Returns 4, d is now deque([1, 2, 3])
print(f"Deque: {d}") # Output: deque([1, 2, 3])

# namedtuple: Factory function for creating tuple subclasses with named fields
Point = namedtuple("Point", ["x", "y"])
p = Point(10, 20)
print(f"Namedtuple: {p.x}, {p.y}") # Output: Namedtuple: 10, 20
```

24. Copy Semantics

Understanding how Python copies objects is crucial, especially with mutable data structures, to avoid unexpected side effects.

```python
import copy

# Shallow Copy (copy.copy() or slicing for lists)
original_list = [1, [2, 3], 4]
shallow_copy = copy.copy(original_list) # or original_list[:]

shallow_copy[0] = 100 # Modifies only shallow_copy
print(f"Original after shallow change: {original_list}") # Output: [1, [2, 3], 4]
print(f"Shallow copy after shallow change: {shallow_copy}") # Output: [100, [2, 3], 4]

shallow_copy[1][0] = 200 # Modifies the nested list, affecting both original and copy
print(f"Original after nested change: {original_list}") # Output: [1, [200, 3], 4]
print(f"Shallow copy after nested change: {shallow_copy}") # Output: [100, [200, 3], 4]

# Deep Copy (copy.deepcopy())
original_list_deep = [1, [2, 3], 4]
deep_copy = copy.deepcopy(original_list_deep)

deep_copy[1][0] = 300 # Modifies only deep_copy
print(f"Original after deep change: {original_list_deep}") # Output: [1, [200, 3], 4] (Note: original_list_deep was affected by previous shallow_copy change)
print(f"Deep copy after deep change: {deep_copy}") # Output: [1, [300, 3], 4]
```

25. Regex Quick Hits

Regular expressions (`re` module) are powerful for pattern matching and manipulation of strings.

```python
import re

text = "The quick brown fox jumps over the lazy dog. The fox is quick."

# re.findall(pattern, string): Finds all non-overlapping matches of pattern
all_fox = re.findall(r"fox", text) # [\'fox\', \'fox\']
print(f"Found all \"fox\": {all_fox}")

# re.sub(pattern, repl, string): Replaces occurrences of pattern with repl
cleaned_text = re.sub(r"\s+", " ", text).strip() # Replaces multiple spaces
with single space and strips whitespace
print(f"Cleaned text: {cleaned_text}")

# re.search(pattern, string): Scans for the first location where the regex
pattern produces a match
match = re.search(r"quick", text)
if match:
    print(f"First match found at: {match.start()}")

# re.match(pattern, string): Checks for a match only at the beginning of the
string
match_start = re.match(r"The", text)
if match_start:
    print("String starts with \"The\"")
```

26. Decorators

Decorators provide a way to modify or enhance functions or methods without changing their source code.

```python
import functools
import time

# @functools.lru_cache: Caches function results
@functools.lru_cache(maxsize=None) # Caches results for faster access
def fibonacci(n):
    if n <= 1:
        return n
    return fibonacci(n-1) + fibonacci(n-2)

# Custom @retry decorator
def retry(max_attempts=3, delay=1):
    def decorator(func):
        @functools.wraps(func)
        def wrapper(*args, **kwargs):
            for attempt in range(1, max_attempts + 1):
                try:
                    return func(*args, **kwargs)
                except Exception as e:
                    print(f"Attempt {attempt} failed: {e}")
                    if attempt < max_attempts:
                        time.sleep(delay)
            raise Exception(f"Function {func.__name__} failed after
{max_attempts} attempts")
        return wrapper
    return decorator

@retry(max_attempts=2, delay=0.5)
def unstable_function():
    if time.time() % 2 < 1: # Simulate occasional failure
        raise ValueError("Temporary error")
    return "Success!"

# print(fibonacci(10)) # Will be fast after first call
# print(unstable_function())
```

27. Properties, Static Methods, & Class Methods

These are special types of methods in Python classes that offer different ways to interact with class and instance data.

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

    @property # Allows accessing a method like an attribute
    def radius(self):
        return self._radius

    @radius.setter # Defines a setter for the property
    def radius(self, value):
        if value < 0:
            raise ValueError("Radius cannot be negative")
        self._radius = value

    @classmethod # Operates on the class itself, not an instance
    def from_diameter(cls, diameter):
        return cls(diameter / 2)

    @staticmethod # A regular function inside a class, doesn't access instance
or class
    def pi():
        return 3.14159

c = Circle(5)
print(f"Radius: {c.radius}") # Access property like an attribute
c.radius = 10 # Use setter
print(f"New Radius: {c.radius}")

c2 = Circle.from_diameter(20) # Use class method to create instance
print(f"Radius from diameter: {c2.radius}")

print(f"Pi: {Circle.pi()}") # Call static method
```

28. Dunder Methods (Special Methods)

Dunder (double underscore) methods allow Python classes to implement special behaviors, enabling them to integrate with Python's built-in functions and operators.

```python
class Book:
    def __init__(self, title, author, pages):
        self.title = title
        self.author = author
        self.pages = pages

    def __repr__(self): # Official string representation for developers
        return f"Book(\"{self.title}\", \"{self.author}\", {self.pages})"

    def __str__(self): # Readable string representation for users
        return f"\"{self.title}\" by {self.author}"

    def __len__(self): # Implements len() for the object
        return self.pages

    def __eq__(self, other): # Defines equality comparison (==)
        if not isinstance(other, Book):
            return NotImplemented
        return self.title == other.title and self.author == other.author

    def __hash__(self): # Makes objects hashable (for use in sets/dict keys)
        return hash((self.title, self.author))

    def __iter__(self): # Makes the object iterable
        yield self.title
        yield self.author
        yield self.pages

book1 = Book("1984", "George Orwell", 328)
book2 = Book("1984", "George Orwell", 328)

print(repr(book1)) # Output: Book("1984", "George Orwell", 328)
print(str(book1))  # Output: "1984" by George Orwell
print(len(book1))  # Output: 328
print(book1 == book2) # Output: True

# Example of iteration
for attr in book1:
    print(attr)
```

29. CLI Arguments with `argparse`

`argparse` is the recommended module for parsing command-line arguments and options.

```python
import argparse

def main():
    parser = argparse.ArgumentParser(description="A simple command-line tool.")
    parser.add_argument("name", type=str, help="Your name")
    parser.add_argument("--age", type=int, default=30, help="Your age")
    parser.add_argument("-v", "--verbose", action="store_true", help="Enable
verbose output")

    args = parser.parse_args()

    if args.verbose:
        print("Verbose mode enabled.")
    print(f"Hello, {args.name}! You are {args.age} years old.")

if __name__ == "__main__":
    # To run this from terminal:
    # python your_script_name.py Alice --age 25
    # python your_script_name.py Bob -v
    main()
```

30. Concurrency

Concurrency allows multiple tasks to make progress seemingly at the same time. Python offers `asyncio` for asynchronous programming and `threading` / `multiprocessing` for parallel execution.

```python
import asyncio
import time
import threading
import multiprocessing

# Asyncio (Asynchronous I/O)
async def fetch_data(delay):
    await asyncio.sleep(delay) # Simulate I/O-bound operation
    return f"Data after {delay} seconds"

async def main_async():
    print("Starting async fetches...")
    results = await asyncio.gather(
        fetch_data(2),
        fetch_data(1)
    )
    print(f"Async results: {results}")

# To run: asyncio.run(main_async())

# Threads vs Processes (brief note)
# Threads: Good for I/O-bound tasks (e.g., network requests) due to GIL.
# Processes: Good for CPU-bound tasks (e.g., heavy computation) as they bypass GIL.

# Example of a simple thread
def task(name):
    print(f"Thread {name}: Starting")
    time.sleep(1)
    print(f"Thread {name}: Finishing")

# thread = threading.Thread(target=task, args=("Worker",))
# thread.start()
# thread.join()
```

31. Performance Tips

Small optimizations can significantly improve Python code performance, especially in critical sections.

```
# String Concatenation vs. Join
words = ["Hello", "world", "this", "is", "a", "test"]
# Bad: string_concat = ""
# for word in words:
#     string_concat += word + " " # Inefficient for many strings

# Good: Use join for efficient string concatenation
string_join = " ".join(words) # Much faster
print(f"Joined string: {string_join}")

# Deque for efficient appends/pops at both ends
from collections import deque
my_queue = deque()
my_queue.append(1)
my_queue.appendleft(0)
my_queue.pop()
my_queue.popleft()

# Sum with start parameter for large iterables
numbers = range(1_000_000)
# total = sum(numbers) # Default start is 0
total_with_start = sum(numbers, 100) # Adds 100 to the sum
print(f"Sum with start: {total_with_start}")
```

32. Contextlib

The `contextlib` module provides utilities for common tasks involving the `with` statement, making it easier to create and manage context managers.

```
from contextlib import suppress, ExitStack

# suppress: Suppress specified exceptions
try:
    with suppress(FileNotFoundError):
        with open("non_existent_file.txt", "r") as f:
            content = f.read()
    print("File not found, but suppressed.")
except Exception as e:
    print(f"An unexpected error occurred: {e}") # This won't be printed for
FileNotFoundError

# ExitStack: Combine multiple context managers
# Useful when you don't know how many context managers you need until runtime
with ExitStack() as stack:
    files = [stack.enter_context(open(f"file_{i}.txt", "w")) for i in range(3)]
    for i, f in enumerate(files):
        f.write(f"Hello from file {i}\n")
    print("All files written and will be closed automatically.")
# Files are automatically closed here
```

Composition vs. Inheritance: - **Inheritance (Is-A relationship):** A `Labrador` *is a* `Dog`. Useful for extending functionality. - **Composition (Has-A relationship):** A `Car` *has an*

`Engine`. Useful for building complex objects from simpler ones, often preferred for flexibility and avoiding deep inheritance hierarchies.

33. Mutable Default Arguments Pitfall

Be cautious with mutable default arguments in function definitions, as they retain their state across calls.

```
def add_item_bad(item, item_list=[]): # ❌ Pitfall: default list is created once
    item_list.append(item)
    return item_list

# print(add_item_bad(1)) # Output: [1]
# print(add_item_bad(2)) # Output: [1, 2] - unexpected!

def add_item_good(item, item_list=None): # ✅ Correct way
    if item_list is None:
        item_list = []
    item_list.append(item)
    return item_list

# print(add_item_good(1)) # Output: [1]
# print(add_item_good(2)) # Output: [2]
```

34. Match / Structural Pattern Matching (Python 3.10+)

Structural pattern matching allows you to match values against patterns, similar to `switch` statements in other languages.

```
def handle_command(command):
    match command:
        case {"action": "greet", "name": name}:
            print(f"Hello, {name}!")
        case {"action": "add", "item": item, "quantity": q} if q > 0:
            print(f"Adding {q} of {item}.")
        case {"action": "quit"}:
            print("Exiting.")
        case _:
            print("Unknown command.")

handle_command({"action": "greet", "name": "Alice"})
handle_command({"action": "add", "item": "apple", "quantity": 5})
handle_command({"action": "add", "item": "orange", "quantity": 0})
handle_command({"action": "quit"})
handle_command({"action": "unknown"})
```

35. Subprocess.run (Real-world Scripting)

The `subprocess` module allows you to run new applications and commands from your Python code, and get their input/output streams.

```
import subprocess as sp

# Run a simple command
try:
    result = sp.run(["echo", "Hello from subprocess!"], capture_output=True,
text=True, check=True)
    print(f"Stdout: {result.stdout.strip()}")
except sp.CalledProcessError as e:
    print(f"Command failed with error: {e}")

# Run a command with arguments
try:
    result = sp.run(["ls", "-l"], capture_output=True, text=True, check=True)
    print("\nls -l output:")
    print(result.stdout)
except sp.CalledProcessError as e:
    print(f"Command failed with error: {e}")
```

36. `heapq` & `bisect` (DSA in Standard Library)

Python's standard library provides efficient implementations for common data structures and algorithms.

```
import heapq
import bisect

# heapq: Implements the heap queue algorithm (priority queue)
pq = []
heapq.heappush(pq, (3, "write code"))
heapq.heappush(pq, (1, "eat breakfast"))
heapq.heappush(pq, (2, "review PR"))

while pq:
    priority, item = heapq.heappop(pq)
    print(f"Processing (Priority: {priority}): {item}")
# Output will be in priority order: eat breakfast, review PR, write code

# bisect: Maintains a list in sorted order without having to sort the list
after each insertion
sorted_list = [10, 20, 30, 40, 50]
x = 25
i = bisect.bisect_left(sorted_list, x) # Finds insertion point for x to
maintain sorted order
bisect.insort_left(sorted_list, x) # Inserts x into sorted_list at position i
print(f"List after bisect: {sorted_list}") # Output: [10, 20, 25, 30, 40, 50]
```

37. Statistics & Decimal (Practical Data/Money)

Python offers modules for statistical calculations and precise decimal arithmetic, crucial for financial applications.

```
 from statistics import mean, median, stdev
from decimal import Decimal, ROUND_HALF_UP

# Statistics module
data_points = [10, 20, 30, 40, 50]
print(f"Mean: {mean(data_points)}")     # Output: 30
print(f"Median: {median(data_points)}") # Output: 30
print(f"Standard Deviation: {stdev(data_points):.2f}") # Output: 15.81

# Decimal for precise arithmetic (e.g., money calculations)
price = Decimal("19.99")
tax_rate = Decimal("0.05")
tax = price * tax_rate
total = price + tax
print(f"Price: {price}, Tax: {tax}, Total: {total}") # Avoids floating point
inaccuracies

# Rounding with Decimal
rounded_total = total.quantize(Decimal("0.01"), rounding=ROUND_HALF_UP)
print(f"Rounded Total: {rounded_total}")
```

38. Typing Extras (Modern Python)

Advanced typing features for more robust and readable code.

```python
from typing import TypedDict, Literal, Optional

# TypedDict: Define dictionary with specific keys and value types
class Price(TypedDict):
    amount: float
    currency: str

def get_price_info(price: Price) -> str:
    return f"Amount: {price[\"amount\"]} {price[\"currency\"]}"

# Literal: Specify that a parameter can only take certain literal values
def set_mode(mode: Literal["fast", "safe"]) -> None:
    print(f"Setting mode to: {mode}")

# Optional: Indicates a value can be of a certain type or None
def get_user_id(username: str) -> Optional[int]:
    if username == "admin":
        return 1
    return None

# Usage
product_price: Price = {"amount": 29.99, "currency": "USD"}
print(get_price_info(product_price))
set_mode("fast")
print(f"Admin ID: {get_user_id("admin")}")
print(f"Guest ID: {get_user_id("guest")}")
```

39. F-string Debugging & Formatting

F-strings (formatted string literals) are powerful for string interpolation and offer convenient debugging features.

```python
name = "Alice"
age = 30

# Debugging with f-strings (Python 3.8+)
print(f"{name=}, {age=}") # Output: name=\'Alice\', age=30

# Formatting numbers
value = 12345.6789
print(f"Formatted: {value:,.2f}") # Output: Formatted: 12,345.68 (thousands
separator, 2 decimal places)

# Padding and alignment
item = "Python"
print(f"Left aligned: {item:<10}") # Output: Left aligned: Python
print(f"Right aligned: {item:>10}") # Output: Right aligned:     Python
print(f"Centered: {item:^10}") # Output: Centered:   Python
```

40. `functools` Gems

The `functools` module provides higher-order functions and operations on callable objects.

```
 from functools import partial, cached_property

# partial: Create a new function with some arguments pre-filled
def multiply(a, b):
    return a * b

double = partial(multiply, 2) # double is now a function that multiplies its
argument by 2
print(f"Double 5: {double(5)}") # Output: 10

# cached_property (Python 3.8+): Cache the result of a method call
class ExpensiveComputation:
    def __init__(self, value):
        self._value = value

    @cached_property
    def computed_result(self):
        print("Performing expensive computation...")
        return self._value * 100

comp = ExpensiveComputation(10)
print(f"First access: {comp.computed_result}") # Computes and caches
print(f"Second access: {comp.computed_result}") # Uses cached value
```

41. Context Manager Protocol

To create your own context managers (objects that can be used with the `with` statement), you implement the `__enter__` and `__exit__` dunder methods.

```
class MyContextManager:
    def __init__(self, resource):
        self.resource = resource

    def __enter__(self):
        print(f"Entering context for {self.resource}")
        # Acquire the resource
        return self.resource

    def __exit__(self, exc_type, exc_val, exc_tb):
        print(f"Exiting context for {self.resource}")
        # Release the resource
        if exc_type:
            print(f"An exception occurred: {exc_val}")
        return False # Propagate exception if True, suppress if False

with MyContextManager("Database Connection") as db:
    print(f"Using {db}")
    # raise ValueError("Something went wrong!") # Uncomment to test exception
handling
```

Composition vs. Inheritance: - **Inheritance (Is-A relationship):** A `Labrador` *is a* `Dog`. Useful for extending functionality. - **Composition (Has-A relationship):** A `Car` *has an* `Engine`. Useful for building complex objects from simpler ones, often preferred for flexibility and avoiding deep inheritance hierarchies.

Composition vs. Inheritance: - **Inheritance (Is-A relationship):** A `Labrador` *is a* `Dog`. Useful for extending functionality. - **Composition (Has-A relationship):** A `Car` *has an* `Engine`. Useful for building complex objects from simpler ones, often preferred for flexibility and avoiding deep inheritance hierarchies.

33. Mutable Default Arguments Pitfall

Be cautious with mutable default arguments in function definitions, as they retain their state across calls.

```python
def add_item_bad(item, item_list=[]): # ❌ Pitfall: default list is created once
    item_list.append(item)
    return item_list

# print(add_item_bad(1)) # Output: [1]
# print(add_item_bad(2)) # Output: [1, 2] - unexpected!

def add_item_good(item, item_list=None): # ✅ Correct way
    if item_list is None:
        item_list = []
    item_list.append(item)
    return item_list

# print(add_item_good(1)) # Output: [1]
# print(add_item_good(2)) # Output: [2]
```

34. Match / Structural Pattern Matching (Python 3.10+)

Structural pattern matching allows you to match values against patterns, similar to `switch` statements in other languages.

```python
def handle_command(command):
    match command:
        case {"action": "greet", "name": name}:
            print(f"Hello, {name}!")
        case {"action": "add", "item": item, "quantity": q} if q > 0:
            print(f"Adding {q} of {item}.")
        case {"action": "quit"}:
            print("Exiting.")
        case _:
            print("Unknown command.")

handle_command({"action": "greet", "name": "Alice"})
handle_command({"action": "add", "item": "apple", "quantity": 5})
handle_command({"action": "add", "item": "orange", "quantity": 0})
handle_command({"action": "quit"})
handle_command({"action": "unknown"})
```

35. `subprocess.run` (Real-world Scripting)

The `subprocess` module allows you to run new applications and commands from your Python code, and get their input/output streams.

```python
import subprocess as sp

# Run a simple command
try:
    result = sp.run(["echo", "Hello from subprocess!"], capture_output=True,
text=True, check=True)
    print(f"Stdout: {result.stdout.strip()}")
except sp.CalledProcessError as e:
    print(f"Command failed with error: {e}")

# Run a command with arguments
try:
    result = sp.run(["ls", "-l"], capture_output=True, text=True, check=True)
    print("\nls -l output:")
    print(result.stdout)
except sp.CalledProcessError as e:
    print(f"Command failed with error: {e}")
```

36. `heapq` & `bisect` (DSA in Standard Library)

Python's standard library provides efficient implementations for common data structures and algorithms.

```python
import heapq
import bisect

# heapq: Implements the heap queue algorithm (priority queue)
pq = []
heapq.heappush(pq, (3, "write code"))
heapq.heappush(pq, (1, "eat breakfast"))
heapq.heappush(pq, (2, "review PR"))

while pq:
    priority, item = heapq.heappop(pq)
    print(f"Processing (Priority: {priority}): {item}")
# Output will be in priority order: eat breakfast, review PR, write code

# bisect: Maintains a list in sorted order without having to sort the list
after each insertion
sorted_list = [10, 20, 30, 40, 50]
x = 25
i = bisect.bisect_left(sorted_list, x) # Finds insertion point for x to
maintain sorted order
bisect.insort_left(sorted_list, x) # Inserts x into sorted_list at position i
print(f"List after bisect: {sorted_list}") # Output: [10, 20, 25, 30, 40, 50]
```

37. Statistics & Decimal (Practical Data/Money)

Python offers modules for statistical calculations and precise decimal arithmetic, crucial for financial applications.

```
 from statistics import mean, median, stdev
from decimal import Decimal, ROUND_HALF_UP

# Statistics module
data_points = [10, 20, 30, 40, 50]
print(f"Mean: {mean(data_points)}")     # Output: 30
print(f"Median: {median(data_points)}") # Output: 30
print(f"Standard Deviation: {stdev(data_points):.2f}") # Output: 15.81

# Decimal for precise arithmetic (e.g., money calculations)
price = Decimal("19.99")
tax_rate = Decimal("0.05")
tax = price * tax_rate
total = price + tax
print(f"Price: {price}, Tax: {tax}, Total: {total}") # Avoids floating point
inaccuracies

# Rounding with Decimal
rounded_total = total.quantize(Decimal("0.01"), rounding=ROUND_HALF_UP)
print(f"Rounded Total: {rounded_total}")
```

38. Typing Extras (Modern Python)

Advanced typing features for more robust and readable code.

```python
from typing import TypedDict, Literal, Optional

# TypedDict: Define dictionary with specific keys and value types
class Price(TypedDict):
    amount: float
    currency: str

def get_price_info(price: Price) -> str:
    return f"Amount: {price[\"amount\"]} {price[\"currency\"]}"

# Literal: Specify that a parameter can only take certain literal values
def set_mode(mode: Literal["fast", "safe"]) -> None:
    print(f"Setting mode to: {mode}")

# Optional: Indicates a value can be of a certain type or None
def get_user_id(username: str) -> Optional[int]:
    if username == "admin":
        return 1
    return None

# Usage
product_price: Price = {"amount": 29.99, "currency": "USD"}
print(get_price_info(product_price))
set_mode("fast")
print(f"Admin ID: {get_user_id("admin")}")
print(f"Guest ID: {get_user_id("guest")}")
```

39. F-string Debugging & Formatting

F-strings (formatted string literals) are powerful for string interpolation and offer convenient debugging features.

```python
name = "Alice"
age = 30

# Debugging with f-strings (Python 3.8+)
print(f"{name=}, {age=}") # Output: name=\'Alice\', age=30

# Formatting numbers
value = 12345.6789
print(f"Formatted: {value:,.2f}") # Output: Formatted: 12,345.68 (thousands
separator, 2 decimal places)

# Padding and alignment
item = "Python"
print(f"Left aligned: {item:<10}") # Output: Left aligned: Python
print(f"Right aligned: {item:>10}") # Output: Right aligned:     Python
print(f"Centered: {item:^10}") # Output: Centered:   Python
```

40. `functools` Gems

The `functools` module provides higher-order functions and operations on callable objects.

```
 from functools import partial, cached_property

# partial: Create a new function with some arguments pre-filled
def multiply(a, b):
    return a * b

double = partial(multiply, 2) # double is now a function that multiplies its
argument by 2
print(f"Double 5: {double(5)}") # Output: 10

# cached_property (Python 3.8+): Cache the result of a method call
class ExpensiveComputation:
    def __init__(self, value):
        self._value = value

    @cached_property
    def computed_result(self):
        print("Performing expensive computation...")
        return self._value * 100

comp = ExpensiveComputation(10)
print(f"First access: {comp.computed_result}") # Computes and caches
print(f"Second access: {comp.computed_result}") # Uses cached value
```

41. Context Manager Protocol

To create your own context managers (objects that can be used with the `with` statement), you implement the `__enter__` and `__exit__` dunder methods.

```python
class MyContextManager:
    def __init__(self, resource):
        self.resource = resource

    def __enter__(self):
        print(f"Entering context for {self.resource}")
        # Acquire the resource
        return self.resource

    def __exit__(self, exc_type, exc_val, exc_tb):
        print(f"Exiting context for {self.resource}")
        # Release the resource
        if exc_type:
            print(f"An exception occurred: {exc_val}")
        return False # Propagate exception if True, suppress if False

with MyContextManager("Database Connection") as db:
    print(f"Using {db}")
    # raise ValueError("Something went wrong!") # Uncomment to test exception handling
```

Composition vs. Inheritance: - **Inheritance (Is-A relationship):** A `Labrador` *is a* `Dog`. Useful for extending functionality. - **Composition (Has-A relationship):** A `Car` *has an* `Engine`. Useful for building complex objects from simpler ones, often preferred for flexibility and avoiding deep inheritance hierarchies.

33. Mutable Default Arguments Pitfall

Be cautious with mutable default arguments in function definitions, as they retain their state across calls.

```python
def add_item_bad(item, item_list=[]): # ❌ Pitfall: default list is created once
    item_list.append(item)
    return item_list

# print(add_item_bad(1)) # Output: [1]
# print(add_item_bad(2)) # Output: [1, 2] - unexpected!

def add_item_good(item, item_list=None): # ✅ Correct way
    if item_list is None:
        item_list = []
    item_list.append(item)
    return item_list

# print(add_item_good(1)) # Output: [1]
# print(add_item_good(2)) # Output: [2]
```

33. Mutable Default Arguments Pitfall

Be cautious with mutable default arguments in function definitions, as they retain their state across calls.

```
 def add_item_bad(item, item_list=[]): # ❌ Pitfall: default list is created
once
    item_list.append(item)
    return item_list

# print(add_item_bad(1)) # Output: [1]
# print(add_item_bad(2)) # Output: [1, 2] - unexpected!

def add_item_good(item, item_list=None): # ✅ Correct way
    if item_list is None:
        item_list = []
    item_list.append(item)
    return item_item_list

# print(add_item_good(1)) # Output: [1]
# print(add_item_good(2)) # Output: [2]
```

36. `heapq` & `bisect` (DSA in Standard Library)

Python's standard library provides efficient implementations for common data structures and algorithms.

```
 import heapq
import bisect

# heapq: Implements the heap queue algorithm (priority queue)
pq = []
heapq.heappush(pq, (3, "write code"))
heapq.heappush(pq, (1, "eat breakfast"))
heapq.heappush(pq, (2, "review PR"))

while pq:
    priority, item = heapq.heappop(pq)
    print(f"Processing (Priority: {priority}): {item}")
# Output will be in priority order: eat breakfast, review PR, write code

# bisect: Maintains a list in sorted order without having to sort the list
after each insertion
sorted_list = [10, 20, 30, 40, 50]
x = 25
i = bisect.bisect_left(sorted_list, x) # Finds insertion point for x to
maintain sorted order
bisect.insort_left(sorted_list, x) # Inserts x into sorted_list at position i
print(f"List after bisect: {sorted_list}") # Output: [10, 20, 25, 30, 40, 50]
```

12. Context Managers

Context managers simplify resource management by ensuring that resources are properly acquired and released. The `with` statement is used to work with context managers.

```
 # Using with open for file handling
with open("my_file.txt", "w", encoding="utf-8") as f:
    f.write("This is a test.\n")
    f.write("Another line.\n")
# File is automatically closed after the 'with' block, even if errors occur

# Custom context manager using contextlib.contextmanager
from contextlib import contextmanager

@contextmanager
def managed_resource(name):
    print(f"Acquiring resource: {name}")
    yield name # The value yielded is assigned to the 'as' variable
    print(f"Releasing resource: {name}")

with managed_resource("Database Connection") as db:
    print(f"Using resource: {db}")
# Output:
# Acquiring resource: Database Connection
# Using resource: Database Connection
# Releasing resource: Database Connection
```

13. Pathlib (Modern File Paths)

`pathlib` offers an object-oriented approach to filesystem paths, making path manipulation more intuitive and less error-prone than `os.path`.

```python
from pathlib import Path

# Current working directory
current_dir = Path.cwd()
print(f"Current directory: {current_dir}")

# Creating paths
data_dir = current_dir / "data"
print(f"Data directory path: {data_dir}")

# Creating directories (if they don't exist)
data_dir.mkdir(exist_ok=True)

# Listing files with glob patterns
# Create some dummy files for demonstration
(data_dir / "report1.csv").write_text("header\nvalue1")
(data_dir / "report2.txt").write_text("some text")

csv_files = list(data_dir.glob("*.csv"))
print(f"CSV files in data directory: {csv_files}")

# Reading and writing text files directly
file_path = data_dir / "my_document.txt"
file_path.write_text("Hello, Pathlib!")
content = file_path.read_text()
print(f"Content of {file_path}: {content}")

# Checking if path exists and is a file/directory
print(f"Is data_dir a directory? {data_dir.is_dir()}")
print(f"Is file_path a file? {file_path.is_file()}")

# Deleting files (use with caution)
# file_path.unlink() # Deletes the file
# data_dir.rmdir() # Deletes empty directory
```

14. Logging (Not Print)

For robust applications, use the `logging` module instead of `print` statements for better control over message severity, output destination, and formatting.

```
import logging

# Basic configuration
logging.basicConfig(level=logging.INFO, format="%(asctime)s - %(levelname)s - %
(message)s")

# Get a logger instance (good practice for modules)
log = logging.getLogger(__name__)

log.debug("This is a debug message") # Won't show with INFO level
log.info("This is an info message")
log.warning("This is a warning message")
log.error("This is an error message")
log.critical("This is a critical message")

# Example with variable
def divide(a, b):
    try:
        result = a / b
        log.info(f"Division successful: {a} / {b} = {result}")
        return result
    except ZeroDivisionError:
        log.error(f"Attempted to divide by zero: {a} / {b}")
        return None

divide(10, 2)
divide(10, 0)
```

15. Virtual Environments & Packaging

Virtual environments isolate project dependencies, and packaging tools manage these dependencies.

```
# Create a virtual environment
python3 -m venv .venv

# Activate the virtual environment (Linux/macOS)
source .venv/bin/activate

# Activate the virtual environment (Windows PowerShell)
.venv\Scripts\Activate.ps1

# Install packages
pip install requests beautifulsoup4

# Generate requirements file
pip freeze > requirements.txt

# Install from requirements file
pip install -r requirements.txt

# Deactivate virtual environment
deactivate

# pipx for CLI tools: Install and run Python applications in isolated environments
# pip install pipx
# pipx install black
# pipx run cowsay "Hello!"
```

16. Testing with Pytest

Writing tests is crucial for maintaining code quality and ensuring correctness. `pytest` is a popular and powerful testing framework.

```python
# test_example.py

def add(a, b):
    return a + b

def test_add():
    assert add(1, 2) == 3
    assert add(-1, 1) == 0
    assert add(0, 0) == 0

def subtract(a, b):
    return a - b

def test_subtract():
    assert subtract(5, 2) == 3
    assert subtract(2, 5) == -3

# To run tests from the terminal:
# pip install pytest
# pytest

# Test discovery: pytest automatically finds files named test_*.py or *_test.py
# and functions/methods starting with `test_`.
```

17. Typing & Dataclasses

Type hints improve code readability and maintainability, allowing static analysis tools to catch errors. `dataclasses` simplify creating classes primarily used for storing data.

```python
from typing import List, Dict, Tuple, Optional
from dataclasses import dataclass

# Type hints for function arguments and return values
def greet(name: str) -> str:
    return f"Hello, {name}"

def calculate_sum(numbers: List[int]) -> int:
    return sum(numbers)

# Using Optional for values that might be None
def get_user_email(user_id: int) -> Optional[str]:
    if user_id == 1:
        return "user1@example.com"
    return None

# Dataclasses: Automatically generate __init__, __repr__, __eq__ methods
@dataclass(frozen=True) # frozen=True makes instances immutable
class Point:
    x: int
    y: int
    z: float = 0.0 # Default value

@dataclass
class User:
    id: int
    name: str
    email: Optional[str] = None
    is_active: bool = True

# Usage
my_point = Point(10, 20)
print(f"Point: {my_point}")

user1 = User(id=1, name="Alice")
user2 = User(id=2, name="Bob", email="bob@example.com")
print(f"User 1: {user1}")
print(f"User 2: {user2}")

# my_point.x = 5 # This would raise an error if Point was frozen
```

18. Comprehensions & Generators

Concise ways to create sequences and iterators.

```python
# Dictionary Comprehension
squares_dict = {x: x**2 for x in range(5)}
print(f"Squares Dictionary: {squares_dict}") # Output: {0: 0, 1: 1, 2: 4, 3: 9, 4: 16}

# Set Comprehension
even_set = {x for x in range(10) if x % 2 == 0}
print(f"Even Set: {even_set}") # Output: {0, 2, 4, 6, 8}

# Generator Expression (lazy evaluation)
gen_exp = (x**2 for x in range(5))
print(f"Generator Expression: {list(gen_exp)}") # Output: [0, 1, 4, 9, 16]

# Generator Function with yield
def fibonacci_sequence(n):
    a, b = 0, 1
    for _ in range(n):
        yield a
        a, b = b, a + b

fib_gen = fibonacci_sequence(5)
print(f"Fibonacci Sequence: {list(fib_gen)}") # Output: [0, 1, 1, 2, 3]

# itertools essentials
import itertools

# chain: Iterate over several iterables as if they were one sequence
chained = list(itertools.chain([1, 2], [3, 4]))
print(f"Chained: {chained}") # Output: [1, 2, 3, 4]

# groupby: Group consecutive identical items
for key, group in itertools.groupby("AAABBCDAA"):
    print(f"{key}: {list(group)}")
# Output:
# A: [\'A\', \'A\', \'A\']
# B: [\'B\', \'B\']
# C: [\'C\']
# D: [\'D\']
# A: [\'A\', \'A\']

# product: Cartesian product of input iterables
prod = list(itertools.product("AB", "12"))
print(f"Product: {prod}") # Output: [(\'A\', \'1\'), (\'A\', \'2\'), (\'B\', \'1\'), (\'B\', \'2\')]
```

19. Unpacking Patterns

Python offers flexible ways to unpack iterables into variables.

```python
# Extended unpacking (Python 3.0+)
seq = [1, 2, 3, 4, 5]
a, *mid, b = seq
print(f"a: {a}, mid: {mid}, b: {b}") # Output: a: 1, mid: [2, 3, 4], b: 5

# Unpacking function arguments
def func(x, y, z):
    print(f"x: {x}, y: {y}, z: {z}")

args = [1, 2, 3]
func(*args) # Unpacks list into positional arguments

kwargs = {"x": 10, "y": 20, "z": 30}
func(**kwargs) # Unpacks dictionary into keyword arguments

# Merging dictionaries
dict1 = {"a": 1, "b": 2}
dict2 = {"b": 3, "c": 4}

# Python 3.5+ using **
merged_dict_35 = {**dict1, **dict2}
print(f"Merged (3.5+): {merged_dict_35}") # Output: {"a": 1, "b": 3, "c": 4}

# Python 3.9+ using |
merged_dict_39 = dict1 | dict2
print(f"Merged (3.9+): {merged_dict_39}") # Output: {"a": 1, "b": 3, "c": 4}
```

20. Datetime and Timezones

Working with dates and times, especially across different timezones, requires careful handling.

```python
from datetime import datetime, timedelta, timezone

# Current UTC time
now_utc = datetime.now(timezone.utc)
print(f"Current UTC time: {now_utc}")

# Local timezone (requires `zoneinfo` module in Python 3.9+ or `pytz` for older versions)
# from zoneinfo import ZoneInfo # Python 3.9+
# local_tz = ZoneInfo("America/New_York")
# now_local = now_utc.astimezone(local_tz)
# print(f"Current local time (NYC): {now_local}")

# Creating a specific datetime object
date_obj = datetime(2023, 9, 14, 10, 30, 0, tzinfo=timezone.utc)
print(f"Specific datetime: {date_obj}")

# Formatting datetime to string (strftime)
formatted_date = date_obj.strftime("%Y-%m-%d %H:%M:%S %Z%z")
print(f"Formatted date: {formatted_date}")

# Parsing string to datetime (strptime)
date_string = "2023-09-14 10:30:00 UTC+0000"
parsed_date = datetime.strptime(date_string, "%Y-%m-%d %H:%M:%S %Z%z")
print(f"Parsed date: {parsed_date}")

# Timedelta for date arithmetic
two_hours_later = now_utc + timedelta(hours=2)
print(f"Two hours later: {two_hours_later}")

yesterday = now_utc - timedelta(days=1)
print(f"Yesterday: {yesterday}")
```

21. HTTP & JSON

Interacting with web services often involves making HTTP requests and handling JSON data.

```
 import requests
import json

# Making an HTTP GET request and parsing JSON response
try:
    # Example: Fetch data from a public API
    response = requests.get("https://jsonplaceholder.typicode.com/todos/1")
    response.raise_for_status() # Raise an exception for HTTP errors
    todo_item = response.json() # Automatically parses JSON response into a
Python dict
    print(f"Fetched Todo: {todo_item[\"title\"]}")
except requests.exceptions.RequestException as e:
    print(f"HTTP Request failed: {e}")

# Converting Python dictionary to JSON string
python_data = {"name": "Manus", "type": "AI Agent", "version": 1.0}
json_string = json.dumps(python_data, indent=2) # indent for pretty printing
print("\nPython dict to JSON:")
print(json_string)

# Converting JSON string to Python dictionary
json_data = '{"product": "Laptop", "price": 1200}'
python_dict = json.loads(json_data)
print(f"\nJSON string to Python dict: {python_dict[\"product\"]}")
```

22. Common Collections

The `collections` module provides specialized container datatypes offering alternatives to Python’s general purpose built-in containers.

```python
from collections import Counter, defaultdict, deque, namedtuple

# Counter: A dict subclass for counting hashable objects
words = ["apple", "banana", "apple", "orange", "banana", "apple"]
word_counts = Counter(words)
print(f"Word counts: {word_counts}") # Output: Counter({"apple": 3, "banana":
2, "orange": 1})
print(f"Most common: {word_counts.most_common(1)}") # Output: [("apple", 3)]

# defaultdict: A dict subclass that calls a factory function to supply missing
values
s = [("yellow", 1), ("blue", 2), ("yellow", 3), ("blue", 4), ("red", 1)]
d = defaultdict(list)
for k, v in s:
    d[k].append(v)
print(f"Grouped by color: {d}") # Output: defaultdict(<class 'list'>,
{'yellow': [1, 3], 'blue': [2, 4], 'red': [1]})

# deque (double-ended queue): A list-like container with fast appends and pops
on either end
dq = deque([1, 2, 3])
dq.appendleft(0) # Adds to the left; dq is now deque([0, 1, 2, 3])
dq.append(4)     # Adds to the right; dq is now deque([0, 1, 2, 3, 4])
popped_right = dq.pop() # Removes from right; popped_right is 4, dq is
deque([0, 1, 2, 3])
popped_left = dq.popleft() # Removes from left; popped_left is 0, dq is
deque([1, 2, 3])
print(f"Deque: {dq}")

# namedtuple: Factory function for creating tuple subclasses with named fields
Point = namedtuple("Point", ["x", "y"])
p = Point(11, y=22)
print(f"Named tuple: {p.x}, {p.y}") # Access by name
print(f"Named tuple as tuple: {p[0]}, {p[1]}") # Access by index
```

23. Copy Semantics (`copy.copy` vs `copy.deepcopy`)

Understanding how Python handles object copying is crucial, especially with mutable objects, to avoid unexpected side effects.

```python
import copy

# Shallow copy: Creates a new compound object but then inserts references into
it
# to the objects found in the original. Changes to nested mutable objects
affect both.
original_list = [[1, 2], [3, 4]]
shallow_copy = copy.copy(original_list)

shallow_copy[0][0] = 99 # Modifies nested list
print(f"Original after shallow copy change: {original_list}") # Output: [[99,
2], [3, 4]]
print(f"Shallow copy: {shallow_copy}") # Output: [[99, 2], [3, 4]]

# Deep copy: Creates a new compound object and then recursively inserts copies
# of the objects found in the original. Changes to nested objects do not affect
the original.
original_list_deep = [[1, 2], [3, 4]]
deep_copy = copy.deepcopy(original_list_deep)

deep_copy[0][0] = 88 # Modifies nested list
print(f"Original after deep copy change: {original_list_deep}") # Output: [[1,
2], [3, 4]]
print(f"Deep copy: {deep_copy}") # Output: [[88, 2], [3, 4]]
```

24. Regex Quick Hits

Regular expressions (`re` module) are powerful for pattern matching and manipulation of strings.

```python
import re

text = "Hello 123 World 456 Python"

# re.findall: Find all occurrences of a pattern
numbers = re.findall(r"\d+", text) # Finds one or more digits
print(f"Numbers found: {numbers}") # Output: [\"123\", \"456\"]

# re.sub: Replace occurrences of a pattern
cleaned_text = re.sub(r"\s+", " ", text).strip() # Replaces multiple spaces
with single space and strips
print(f"Cleaned text: {cleaned_text}") # Output: Hello 123 World 456 Python

# re.search: Search for the first occurrence of a pattern
match = re.search(r"World", text)
if match:
    print(f"Found \"World\" at index {match.start()}")

# re.match: Check for pattern at the beginning of the string
match_start = re.match(r"Hello", text)
if match_start:
    print("String starts with Hello")

# Compile regex for efficiency if used multiple times
compiled_pattern = re.compile(r"\d+")
more_numbers = compiled_pattern.findall("Value 789 and 007")
print(f"More numbers: {more_numbers}")
```

25. Decorators

Decorators provide a way to modify or enhance functions or methods without changing their source code. They are often used for logging, access control, memoization, and more.

```python
import time
from functools import wraps, lru_cache

# Simple decorator for timing function execution
def timing_decorator(func):
    @wraps(func) # Preserves original function metadata
    def wrapper(*args, **kwargs):
        start_time = time.time()
        result = func(*args, **kwargs)
        end_time = time.time()
        print(f"{func.__name__} took {end_time - start_time:.4f} seconds")
        return result
    return wrapper

@timing_decorator
def long_running_function(n):
    time.sleep(n)
    return f"Slept for {n} seconds"

print(long_running_function(1))

# @lru_cache: Memoization decorator (caches function results)
@lru_cache(maxsize=None) # maxsize=None means unlimited cache size
def fibonacci(n):
    if n <= 1:
        return n
    return fibonacci(n-1) + fibonacci(n-2)

print(f"Fibonacci(10): {fibonacci(10)}") # Computes and caches
print(f"Fibonacci(10): {fibonacci(10)}") # Uses cached result (faster)

# Custom @retry shape (conceptual example)
def retry(max_attempts=3, delay=1):
    def decorator(func):
        @wraps(func)
        def wrapper(*args, **kwargs):
            for attempt in range(1, max_attempts + 1):
                try:
                    return func(*args, **kwargs)
                except Exception as e:
                    print(f"Attempt {attempt} failed: {e}")
                    if attempt < max_attempts:
                        time.sleep(delay)
            raise Exception(f"Function failed after {max_attempts} attempts")
        return wrapper
    return decorator

# @retry(max_attempts=2, delay=0.5)
# def flaky_function():
#     if random.random() < 0.7: # 70% chance of failure
#         raise ValueError("Simulated network error")
#     return "Success!"

# print(flaky_function())
```

26. Properties / `staticmethod` / `classmethod`

These are special types of methods in classes that provide enhanced functionality or different ways of interacting with class and instance data.

```
class Circle:
    def __init__(self, radius):
        self._radius = radius # Convention for 'protected' attribute

    @property # Allows method to be accessed like an attribute
    def radius(self):
        return self._radius

    @radius.setter # Allows setting the property with validation
    def radius(self, value):
        if value < 0:
            raise ValueError("Radius cannot be negative")
        self._radius = value

    @property
    def diameter(self):
        return self._radius * 2

    @staticmethod # Belongs to the class, but doesn't access instance or class
data
    def pi_value():
        return 3.14159

    @classmethod # Operates on the class itself, often used for alternative
constructors
    def from_diameter(cls, diameter):
        return cls(diameter / 2)

# Usage
c = Circle(5)
print(f"Radius: {c.radius}") # Access as attribute
c.radius = 7 # Set as attribute
print(f"Diameter: {c.diameter}") # Computed attribute
print(f"Pi value: {Circle.pi_value()}") # Call static method via class

c2 = Circle.from_diameter(10) # Use class method as alternative constructor
print(f"Circle from diameter: {c2.radius}")
```

27. Dunder Methods (Special Methods)

"Dunder" (double underscore) methods allow Python classes to implement behavior that interacts with built-in functions and operators (e.g., `len()`, `+`, `==`).

```python
class MyVector:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __repr__(self): # Official string representation for developers
        return f"MyVector(x={self.x}, y={self.y})"

    def __str__(self): # Informal string representation for users
        return f"({self.x}, {self.y})"

    def __add__(self, other): # Defines behavior for the + operator
        return MyVector(self.x + other.x, self.y + other.y)

    def __eq__(self, other): # Defines behavior for the == operator
        return self.x == other.x and self.y == other.y

    def __len__(self): # Defines behavior for len() function
        return 2 # For a 2D vector

    def __hash__(self): # Makes objects hashable (e.g., for use in sets or dict keys)
        return hash((self.x, self.y))

    def __iter__(self): # Makes object iterable
        yield self.x
        yield self.y

v1 = MyVector(1, 2)
v2 = MyVector(3, 4)

print(repr(v1)) # Output: MyVector(x=1, y=2)
print(str(v1))  # Output: (1, 2)
print(v1 + v2)  # Output: (4, 6)
print(v1 == MyVector(1, 2)) # Output: True
print(len(v1))  # Output: 2

my_set = {v1, MyVector(5, 6)}
print(my_set) # Requires __hash__

for coord in v1:
    print(coord) # Requires __iter__
```

28. CLI Arguments with `argparse`

The `argparse` module makes it easy to write user-friendly command-line interfaces.

```python
import argparse

def main():
    parser = argparse.ArgumentParser(description="A simple command-line tool.")
    parser.add_argument("name", type=str, help="Your name")
    parser.add_argument("--greeting", "-g", type=str, default="Hello",
                        help="An optional greeting message")
    parser.add_argument("--verbose", "-v", action="store_true",
                        help="Enable verbose output")

    args = parser.parse_args()

    if args.verbose:
        print("Verbose mode enabled.")

    print(f"{args.greeting}, {args.name}!")

# To run from command line:
# python your_script_name.py Alice
# python your_script_name.py Bob --greeting Hi -v
```

29. Concurrency (`asyncio`, Threads vs. Processes)

Python offers several ways to handle concurrency, allowing programs to do multiple things seemingly at once.

```python
import asyncio
import threading
import multiprocessing
import time

# asyncio (asynchronous I/O): For I/O-bound tasks
async def fetch_data(delay):
    await asyncio.sleep(delay) # Simulate I/O operation
    return f"Data fetched after {delay} seconds"

async def main_async():
    print("Starting async fetches...")
    results = await asyncio.gather(
        fetch_data(2),
        fetch_data(1)
    )
    print(f"Async results: {results}")

# asyncio.run(main_async()) # Uncomment to run async example

# Threads vs. Processes:
# Threads: Good for I/O-bound tasks (e.g., network requests, file I/O) due to GIL.
# Processes: Good for CPU-bound tasks (e.g., heavy computation) as they bypass GIL.

# Example with Threads (I/O-bound simulation)
def task_thread(name, delay):
    time.sleep(delay)
    print(f"Thread {name}: finished after {delay}s")

# thread1 = threading.Thread(target=task_thread, args=("One", 2))
# thread2 = threading.Thread(target=task_thread, args=("Two", 1))
# thread1.start()
# thread2.start()
# thread1.join()
# thread2.join()
# print("All threads finished.")

# Example with Processes (CPU-bound simulation)
def task_process(name, count):
    sum_val = 0
    for _ in range(count):
        sum_val += 1
    print(f"Process {name}: finished computation.")

# process1 = multiprocessing.Process(target=task_process, args=("A", 10**7))
# process2 = multiprocessing.Process(target=task_process, args=("B", 10**7))
# process1.start()
# process2.start()
# process1.join()
# process2.join()
# print("All processes finished.")
```

30. Performance Tips

Small optimizations can significantly impact performance in Python.

```
 import timeit
from collections import deque

# String concatenation vs. join
# Concatenation (slower for many strings)
print(timeit.timeit("s = ''
for i in range(1000): s += str(i)", number=100))

# Join (faster for many strings)
print(timeit.timeit("s_list = []
for i in range(1000): s_list.append(str(i))
s = ''.join(s_list)", number=100))

# deque for efficient appends/pops from both ends (queues)
my_queue = deque()
my_queue.append(1)
my_queue.append(2)
my_queue.popleft() # O(1) operation

# sum() with start parameter for efficiency with iterables
numbers = range(1000000)
# sum(numbers) # Default start=0
# sum(numbers, 100) # Start sum from 100
```

31. `contextlib` (Utilities for `with` Statement)

The `contextlib` module provides utilities for common tasks with the `with` statement, making it easier to create context managers.

```python
from contextlib import suppress, ExitStack

# suppress: Suppress specified exceptions
try:
    with suppress(FileNotFoundError):
        with open("non_existent_file.txt", "r") as f:
            content = f.read()
            print(content)
    print("File not found error suppressed.")
except Exception as e:
    print(f"An unexpected error occurred: {e}")

# ExitStack: Combine multiple context managers
# Useful when the number of context managers is not known beforehand
files = ["file1.txt", "file2.txt"]
# Create dummy files for demonstration
with open("file1.txt", "w") as f: f.write("Content of file1")
with open("file2.txt", "w") as f: f.write("Content of file2")

with ExitStack() as stack:
    open_files = [stack.enter_context(open(fname, "r")) for fname in files]
    for i, f in enumerate(open_files):
        print(f"Content of {files[i]}: {f.read()}")
# All files are automatically closed when exiting the ExitStack context
```

12. Context Managers

Context managers simplify resource management by ensuring that resources are properly acquired and released. The `with` statement is used to work with context managers.

```python
# Using with open for file handling
with open("my_file.txt", "w", encoding="utf-8") as f:
    f.write("This is a test.\n")
    f.write("Another line.\n")
# File is automatically closed after the 'with' block, even if errors occur

# Custom context manager via contextlib.contextmanager
from contextlib import contextmanager

@contextmanager
def managed_resource(name):
    print(f"Acquiring resource: {name}")
    yield name # The value yielded is assigned to the 'as' variable
    print(f"Releasing resource: {name}")

with managed_resource("Database Connection") as db:
    print(f"Using resource: {db}")
# Output:
# Acquiring resource: Database Connection
# Using resource: Database Connection
# Releasing resource: Database Connection
```

13. Pathlib (Modern File Paths)

`pathlib` offers an object-oriented approach to filesystem paths, making path manipulation more intuitive and less error-prone than `os.path`.

```python
from pathlib import Path

# Current working directory
current_dir = Path.cwd()
print(f"Current directory: {current_dir}")

# Creating paths
data_dir = current_dir / "data"
print(f"Data directory path: {data_dir}")

# Creating directories (if they don't exist)
data_dir.mkdir(exist_ok=True)

# Listing files with glob patterns
# Create some dummy files for demonstration
(data_dir / "report1.csv").write_text("header\nvalue1")
(data_dir / "report2.txt").write_text("some text")

csv_files = list(data_dir.glob("*.csv"))
print(f"CSV files in data directory: {csv_files}")

# Reading and writing text files directly
file_path = data_dir / "my_document.txt"
file_path.write_text("Hello, Pathlib!")
content = file_path.read_text()
print(f"Content of {file_path}: {content}")

# Checking if path exists and is a file/directory
print(f"Is data_dir a directory? {data_dir.is_dir()}")
print(f"Is file_path a file? {file_path.is_file()}")

# Deleting files (use with caution)
# file_path.unlink() # Deletes the file
# data_dir.rmdir() # Deletes empty directory
```

14. Logging (Not Print)

For robust applications, use the `logging` module instead of `print` statements for better control over message severity, output destination, and formatting.

```
import logging

# Basic configuration
logging.basicConfig(level=logging.INFO, format="%(asctime)s - %(levelname)s - %
(message)s")

# Get a logger instance (good practice for modules)
log = logging.getLogger(__name__)

log.debug("This is a debug message") # Won't show with INFO level
log.info("This is an info message")
log.warning("This is a warning message")
log.error("This is an error message")
log.critical("This is a critical message")

# Example with variable
def divide(a, b):
    try:
        result = a / b
        log.info(f"Division successful: {a} / {b} = {result}")
        return result
    except ZeroDivisionError:
        log.error(f"Attempted to divide by zero: {a} / {b}")
        return None

divide(10, 2)
divide(10, 0)
```

15. Virtual Environments & Packaging

Virtual environments isolate project dependencies, and packaging tools manage these dependencies.

```
# Create a virtual environment
python3 -m venv .venv

# Activate the virtual environment (Linux/macOS)
source .venv/bin/activate

# Activate the virtual environment (Windows PowerShell)
.venv\Scripts\Activate.ps1

# Install packages
pip install requests beautifulsoup4

# Generate requirements file
pip freeze > requirements.txt

# Install from requirements file
pip install -r requirements.txt

# Deactivate virtual environment
deactivate

# pipx for CLI tools: Install and run Python applications in isolated environments
# pip install pipx
# pipx install black
# pipx run cowsay "Hello!"
```

16. Testing with Pytest

Writing tests is crucial for maintaining code quality and ensuring correctness. `pytest` is a popular and powerful testing framework.

```python
# test_example.py

def add(a, b):
    return a + b

def test_add():
    assert add(1, 2) == 3
    assert add(-1, 1) == 0
    assert add(0, 0) == 0

def subtract(a, b):
    return a - b

def test_subtract():
    assert subtract(5, 2) == 3
    assert subtract(2, 5) == -3

# To run tests from the terminal:
# pip install pytest
# pytest

# Test discovery: pytest automatically finds files named test_*.py or *_test.py
# and functions/methods starting with `test_`.
```

17. Typing & Dataclasses

Type hints improve code readability and maintainability, allowing static analysis tools to catch errors. `dataclasses` simplify creating classes primarily used for storing data.

```python
from typing import List, Dict, Tuple, Optional
from dataclasses import dataclass

# Type hints for function arguments and return values
def greet(name: str) -> str:
    return f"Hello, {name}"

def calculate_sum(numbers: List[int]) -> int:
    return sum(numbers)

# Using Optional for values that might be None
def get_user_email(user_id: int) -> Optional[str]:
    if user_id == 1:
        return "user1@example.com"
    return None

# Dataclasses: Automatically generate __init__, __repr__, __eq__ methods
@dataclass(frozen=True) # frozen=True makes instances immutable
class Point:
    x: int
    y: int
    z: float = 0.0 # Default value

@dataclass
class User:
    id: int
    name: str
    email: Optional[str] = None
    is_active: bool = True

# Usage
my_point = Point(10, 20)
print(f"Point: {my_point}")

user1 = User(id=1, name="Alice")
user2 = User(id=2, name="Bob", email="bob@example.com")
print(f"User 1: {user1}")
print(f"User 2: {user2}")

# my_point.x = 5 # This would raise an error if Point was frozen
```

18. Comprehensions & Generators

Concise ways to create sequences and iterators.

```python
# Dictionary Comprehension
squares_dict = {x: x**2 for x in range(5)}
print(f"Squares Dictionary: {squares_dict}") # Output: {0: 0, 1: 1, 2: 4, 3: 9,
4: 16}

# Set Comprehension
even_set = {x for x in range(10) if x % 2 == 0}
print(f"Even Set: {even_set}") # Output: {0, 2, 4, 6, 8}

# Generator Expression (lazy evaluation)
gen_exp = (x**2 for x in range(5))
print(f"Generator Expression: {list(gen_exp)}") # Output: [0, 1, 4, 9, 16]

# Generator Function with yield
def fibonacci_sequence(n):
    a, b = 0, 1
    for _ in range(n):
        yield a
        a, b = b, a + b

fib_gen = fibonacci_sequence(5)
print(f"Fibonacci Sequence: {list(fib_gen)}") # Output: [0, 1, 1, 2, 3]

# itertools essentials
import itertools

# chain: Iterate over several iterables as if they were one sequence
chained = list(itertools.chain([1, 2], [3, 4]))
print(f"Chained: {chained}") # Output: [1, 2, 3, 4]

# groupby: Group consecutive identical items
for key, group in itertools.groupby("AAABBCDAA"):
    print(f"{key}: {list(group)}")
# Output:
# A: [\"A\", \"A\", \"A\"]
# B: [\"B\", \"B\"]
# C: [\"C\"]
# D: [\"D\"]
# A: [\"A\", \"A\"]

# product: Cartesian product of input iterables
prod = list(itertools.product("AB", "12"))
print(f"Product: {prod}") # Output: [("A", "1"), ("A", "2"), ("B", "1"), ("B",
"2")]
```

19. Unpacking Patterns

Python offers flexible ways to unpack iterables into variables.

```python
# Extended unpacking (Python 3.0+)
seq = [1, 2, 3, 4, 5]
a, *mid, b = seq
print(f"a: {a}, mid: {mid}, b: {b}") # Output: a: 1, mid: [2, 3, 4], b: 5

# Unpacking function arguments
def func(x, y, z):
    print(f"x: {x}, y: {y}, z: {z}")

args = [1, 2, 3]
func(*args) # Unpacks list into positional arguments

kwargs = {"x": 10, "y": 20, "z": 30}
func(**kwargs) # Unpacks dictionary into keyword arguments

# Merging dictionaries
dict1 = {"a": 1, "b": 2}
dict2 = {"b": 3, "c": 4}

# Python 3.5+ using **
merged_dict_35 = {**dict1, **dict2}
print(f"Merged (3.5+): {merged_dict_35}") # Output: {"a": 1, "b": 3, "c": 4}

# Python 3.9+ using |
merged_dict_39 = dict1 | dict2
print(f"Merged (3.9+): {merged_dict_39}") # Output: {"a": 1, "b": 3, "c": 4}
```

20. Datetime and Timezones

Working with dates and times, especially across different timezones, requires careful handling.

```python
from datetime import datetime, timedelta, timezone

# Current UTC time
now_utc = datetime.now(timezone.utc)
print(f"Current UTC time: {now_utc}")

# Local timezone (requires `zoneinfo` module in Python 3.9+ or `pytz` for older
versions)
# from zoneinfo import ZoneInfo # Python 3.9+
# local_tz = ZoneInfo("America/New_York")
# now_local = now_utc.astimezone(local_tz)
# print(f"Current local time (NYC): {now_local}")

# Creating a specific datetime object
date_obj = datetime(2023, 9, 14, 10, 30, 0, tzinfo=timezone.utc)
print(f"Specific datetime: {date_obj}")

# Formatting datetime to string (strftime)
formatted_date = date_obj.strftime("%Y-%m-%d %H:%M:%S %Z%z")
print(f"Formatted date: {formatted_date}")

# Parsing string to datetime (strptime)
date_string = "2023-09-14 10:30:00 UTC+0000"
parsed_date = datetime.strptime(date_string, "%Y-%m-%d %H:%M:%S %Z%z")
print(f"Parsed date: {parsed_date}")

# Timedelta for date arithmetic
two_hours_later = now_utc + timedelta(hours=2)
print(f"Two hours later: {two_hours_later}")

yesterday = now_utc - timedelta(days=1)
print(f"Yesterday: {yesterday}")
```

21. HTTP & JSON

Interacting with web services often involves making HTTP requests and handling JSON data.

```
 import requests
import json

# Making an HTTP GET request and parsing JSON response
try:
    # Example: Fetch data from a public API
    response = requests.get("https://jsonplaceholder.typicode.com/todos/1")
    response.raise_for_status() # Raise an exception for HTTP errors
    todo_item = response.json() # Automatically parses JSON response into a
Python dict
    print(f"Fetched Todo: {todo_item[\"title\"]}")
except requests.exceptions.RequestException as e:
    print(f"HTTP Request failed: {e}")

# Converting Python dictionary to JSON string
python_data = {"name": "Manus", "type": "AI Agent", "version": 1.0}
json_string = json.dumps(python_data, indent=2) # indent for pretty printing
print("\nPython dict to JSON:")
print(json_string)

# Converting JSON string to Python dictionary
json_data = '{"product": "Laptop", "price": 1200}'
python_dict = json.loads(json_data)
print(f"\nJSON string to Python dict: {python_dict[\"product\"]}")
```

33. Mutable Default Arguments Pitfall

Be cautious with mutable default arguments in function definitions, as they retain their state across calls.

```
 def add_item_bad(item, item_list=[]): # ❌ Pitfall: default list is created
once
    item_list.append(item)
    return item_list

# print(add_item_bad(1)) # Output: [1]
# print(add_item_bad(2)) # Output: [1, 2] - unexpected!

def add_item_good(item, item_list=None): # ✅ Correct way
    if item_list is None:
        item_list = []
    item_list.append(item)
    return item_list

# print(add_item_good(1)) # Output: [1]
# print(add_item_good(2)) # Output: [2]
```

22. Common Collections

The `collections` module provides specialized container datatypes offering alternatives to Python's general purpose built-in containers.

```
from collections import Counter, defaultdict, deque, namedtuple

# Counter: A dict subclass for counting hashable objects
words = ["apple", "banana", "apple", "orange", "banana", "apple"]
word_counts = Counter(words)
print(f"Word counts: {word_counts}") # Output: Counter({"apple": 3, "banana":
2, "orange": 1})
print(f"Most common: {word_counts.most_common(1)}") # Output: [("apple", 3)]

# defaultdict: A dict subclass that calls a factory function to supply missing
values
s = [("yellow", 1), ("blue", 2), ("yellow", 3), ("blue", 4), ("red", 1)]
d = defaultdict(list)
for k, v in s:
    d[k].append(v)
print(f"Grouped by color: {d}") # Output: defaultdict(<class 'list'>,
{'yellow': [1, 3], 'blue': [2, 4], 'red': [1]})

# deque (double-ended queue): A list-like container with fast appends and pops
on either end
dq = deque([1, 2, 3])
dq.appendleft(0) # Adds to the left; dq is now deque([0, 1, 2, 3])
dq.append(4)     # Adds to the right; dq is now deque([0, 1, 2, 3, 4])
popped_right = dq.pop() # Removes from right; popped_right is 4, dq is
deque([0, 1, 2, 3])
popped_left = dq.popleft() # Removes from left; popped_left is 0, dq is
deque([1, 2, 3])
print(f"Deque: {dq}")

# namedtuple: Factory function for creating tuple subclasses with named fields
Point = namedtuple("Point", ["x", "y"])
p = Point(11, y=22)
print(f"Named tuple: {p.x}, {p.y}") # Access by name
print(f"Named tuple as tuple: {p[0]}, {p[1]}") # Access by index
```

34. Match / Structural Pattern Matching (Python 3.10+)

Structural pattern matching allows you to match values against patterns, similar to `switch` statements in other languages.

```python
def handle_command(command):
    match command:
        case {"action": "greet", "name": name}:
            print(f"Hello, {name}!")
        case {"action": "add", "item": item, "quantity": q} if q > 0:
            print(f"Adding {q} of {item}.")
        case {"action": "quit"}:
            print("Exiting.")
        case _:
            print("Unknown command.")

handle_command({"action": "greet", "name": "Alice"})
handle_command({"action": "add", "item": "apple", "quantity": 5})
handle_command({"action": "add", "item": "orange", "quantity": 0})
handle_command({"action": "quit"})
handle_command({"action": "unknown"})
```

35. `subprocess.run` (Real-world Scripting)

The `subprocess` module allows you to run new applications and commands from your Python code, and get their input/output streams.

```python
import subprocess as sp

# Run a simple command
try:
    result = sp.run(["echo", "Hello from subprocess!"], capture_output=True,
text=True, check=True)
    print(f"Stdout: {result.stdout.strip()}")
except sp.CalledProcessError as e:
    print(f"Command failed with error: {e}")

# Run a command with arguments
try:
    result = sp.run(["ls", "-l"], capture_output=True, text=True, check=True)
    print("\nls -l output:")
    print(result.stdout)
except sp.CalledProcessError as e:
    print(f"Command failed with error: {e}")
```

36. `heapq` & `bisect` (DSA in Standard Library)

Python's standard library provides efficient implementations for common data structures and algorithms.

```python
import heapq
import bisect

# heapq: Implements the heap queue algorithm (priority queue)
pq = []
heapq.heappush(pq, (3, "write code"))
heapq.heappush(pq, (1, "eat breakfast"))
heapq.heappush(pq, (2, "review PR"))

while pq:
    priority, item = heapq.heappop(pq)
    print(f"Processing (Priority: {priority}): {item}")
# Output will be in priority order: eat breakfast, review PR, write code

# bisect: Maintains a list in sorted order without having to sort the list
after each insertion
sorted_list = [10, 20, 30, 40, 50]
x = 25
i = bisect.bisect_left(sorted_list, x) # Finds insertion point for x to
maintain sorted order
bisect.insort_left(sorted_list, x) # Inserts x into sorted_list at position i
print(f"List after bisect: {sorted_list}") # Output: [10, 20, 25, 30, 40, 50]
```

37. `statistics` & `decimal` (Practical Data/Money)

Python's standard library provides modules for common statistical operations and precise decimal arithmetic.

```python
from statistics import mean, median, stdev
from decimal import Decimal, ROUND_HALF_UP, getcontext

# statistics: Basic statistical functions
data = [1.0, 2.5, 3.0, 2.5, 4.0]
print(f"Mean: {mean(data)}")    # Output: 2.6
print(f"Median: {median(data)}")  # Output: 2.5
print(f"Standard Deviation: {stdev(data):.2f}") # Output: 1.12

# decimal: For precise floating-point arithmetic (e.g., financial calculations)
getcontext().prec = 10 # Set precision for Decimal operations

price = Decimal('19.99')
tax_rate = Decimal('0.075')
tax = price * tax_rate
total = price + tax

print(f"Price: {price}")
print(f"Tax: {tax.quantize(Decimal('0.01'), rounding=ROUND_HALF_UP)}") # Round
to 2 decimal places
print(f"Total: {total.quantize(Decimal('0.01'), rounding=ROUND_HALF_UP)}")

# Avoid floating point inaccuracies
# print(0.1 + 0.2) # Output: 0.30000000000000004
# print(Decimal('0.1') + Decimal('0.2')) # Output: 0.3
```

38. Typing Extras (Modern Python)

Advanced type hints for more expressive and robust code.

```
 from typing import TypedDict, Literal, Optional, Union, Any

# TypedDict: Define dictionary with specific keys and value types
class Price(TypedDict):
    amount: float
    currency: str

def get_price_info(price: Price) -> str:
    return f"Amount: {price[\"amount\"]} {price[\"currency\"]}"

my_price: Price = {"amount": 29.99, "currency": "USD"}
print(get_price_info(my_price))

# Literal: Specify that a parameter can only take specific literal values
def set_mode(mode: Literal["fast", "safe"]) -> None:
    print(f"Setting mode to: {mode}")

set_mode("fast")
# set_mode("unsafe") # Type checker would flag this as an error

# Optional: Indicates a value can be either a type or None
def get_optional_value(value: Optional[str]) -> str:
    return value if value is not None else "No value"

print(get_optional_value("hello"))
print(get_optional_value(None))

# Union: Indicates a value can be one of several types
def process_id(id: Union[int, str]) -> None:
    print(f"Processing ID: {id}, Type: {type(id)}")

process_id(123)
process_id("abc")

# Any: Opt-out of type checking for a specific variable or function parameter
def process_anything(data: Any) -> Any:
    print(f"Processing anything: {data}")
    return data

process_anything([1, 2, 3])
```

39. F-string Debugging & Formatting

F-strings (formatted string literals) provide a concise and convenient way to embed Python expressions inside string literals for formatting and debugging.

```
 name = "Alice"
age = 30

# Debugging with f-strings (Python 3.8+)
print(f"{name=}") # Output: name=\'Alice\'
print(f"{age=}")  # Output: age=30

# Numeric formatting
price = 12345.6789
print(f"Price: {price:,.2f}") # Thousands separator, 2 decimal places. Output:
Price: 12,345.68

# Date formatting with strftime mini-language
from datetime import datetime
now = datetime.now()
print(f"Current date: {now:%Y-%m-%d}") # Output: Current date: 2023-09-14
print(f"Current time: {now:%H:%M:%S}") # Output: Current time: 10:30:00

# Alignment and padding
text = "Python"
print(f"{text:>10}")  # Right-align in 10 spaces. Output:     Python
print(f"{text:<10}")  # Left-align in 10 spaces. Output: Python
print(f"{text:^10}")  # Center-align in 10 spaces. Output:   Python
print(f"{text:=^10}") # Center-align with fill character. Output: ==Python==

# Percentage formatting
ratio = 0.75
print(f"Completion: {ratio:.2%}") # Output: Completion: 75.00%
```

40. `functools` Gems

The `functools` module provides higher-order functions and operations on callable objects.

```python
from functools import partial, cached_property, wraps

# partial: Create a new function with some arguments pre-filled
def multiply(a, b):
    return a * b

double = partial(multiply, 2) # New function that always multiplies by 2
print(f"Double 5: {double(5)}") # Output: 10

# cached_property (Python 3.8+): Cache the result of a method call
class MyClass:
    def __init__(self, name):
        self.name = name
        self._heavy_computation_count = 0

    @cached_property
    def heavy_computation(self):
        self._heavy_computation_count += 1
        print("Performing heavy computation...")
        return f"Result of heavy computation for {self.name} (call
{self._heavy_computation_count})"

obj = MyClass("Example")
print(obj.heavy_computation) # First call, computes and caches
print(obj.heavy_computation) # Second call, uses cached result
print(obj.heavy_computation) # Third call, uses cached result

# wraps: Decorator to preserve metadata of the original function when using
decorators
# (Already demonstrated in the Decorators section)
```

41. Context Manager Protocol (When Rolling Your Own)

To create your own context manager without `contextlib.contextmanager`, you implement the `__enter__` and `__exit__` dunder methods.

```python
class MyContextManager:
    def __init__(self, resource_name):
        self.resource_name = resource_name

    def __enter__(self):
        print(f"Entering context for {self.resource_name}")
        # Acquire the resource here
        return self.resource_name # Value returned by 'as' in 'with' statement

    def __exit__(self, exc_type, exc_val, exc_tb):
        print(f"Exiting context for {self.resource_name}")
        # Release the resource here
        if exc_type:
            print(f"An exception occurred: {exc_val}")
        return False # Propagate exception if False, suppress if True

with MyContextManager("Custom Resource") as res:
    print(f"Inside context, using: {res}")
    # raise ValueError("Something went wrong!") # Uncomment to test exception handling

print("Outside context.")
```

37. `statistics` & `decimal` (Practical Data/Money)

Python's standard library provides modules for common statistical operations and precise decimal arithmetic.

```python
from statistics import mean, median, stdev
from decimal import Decimal, ROUND_HALF_UP, getcontext

# statistics: Basic statistical functions
data = [1.0, 2.5, 3.0, 2.5, 4.0]
print(f"Mean: {mean(data)}")    # Output: 2.6
print(f"Median: {median(data)}")  # Output: 2.5
print(f"Standard Deviation: {stdev(data):.2f}") # Output: 1.12

# decimal: For precise floating-point arithmetic (e.g., financial calculations)
getcontext().prec = 10 # Set precision for Decimal operations

price = Decimal('19.99')
tax_rate = Decimal('0.075')
tax = price * tax_rate
total = price + tax

print(f"Price: {price}")
print(f"Tax: {tax.quantize(Decimal('0.01'), rounding=ROUND_HALF_UP)}") # Round to 2 decimal places
print(f"Total: {total.quantize(Decimal('0.01'), rounding=ROUND_HALF_UP)}")

# Avoid floating point inaccuracies
# print(0.1 + 0.2) # Output: 0.30000000000000004
# print(Decimal('0.1') + Decimal('0.2')) # Output: 0.3
```

38. Typing Extras (Modern Python)

Advanced type hints for more expressive and robust code.

```
 from typing import TypedDict, Literal, Optional, Union, Any

# TypedDict: Define dictionary with specific keys and value types
class Price(TypedDict):
    amount: float
    currency: str

def get_price_info(price: Price) -> str:
    return f"Amount: {price[\"amount\"]} {price[\"currency\"]}"

my_price: Price = {"amount": 29.99, "currency": "USD"}
print(get_price_info(my_price))

# Literal: Specify that a parameter can only take specific literal values
def set_mode(mode: Literal["fast", "safe"]) -> None:
    print(f"Setting mode to: {mode}")

set_mode("fast")
# set_mode("unsafe") # Type checker would flag this as an error

# Optional: Indicates a value can be either a type or None
def get_optional_value(value: Optional[str]) -> str:
    return value if value is not None else "No value"

print(get_optional_value("hello"))
print(get_optional_value(None))

# Union: Indicates a value can be one of several types
def process_id(id: Union[int, str]) -> None:
    print(f"Processing ID: {id}, Type: {type(id)}")

process_id(123)
process_id("abc")

# Any: Opt-out of type checking for a specific variable or function parameter
def process_anything(data: Any) -> Any:
    print(f"Processing anything: {data}")
    return data

process_anything([1, 2, 3])
```

39. F-string Debugging & Formatting

F-strings (formatted string literals) provide a concise and convenient way to embed Python expressions inside string literals for formatting and debugging.

```
 name = "Alice"
age = 30

# Debugging with f-strings (Python 3.8+)
print(f"{name=}") # Output: name=\'Alice\'
print(f"{age=}")  # Output: age=30

# Numeric formatting
price = 12345.6789
print(f"Price: {price:,.2f}") # Thousands separator, 2 decimal places. Output:
Price: 12,345.68

# Date formatting with strftime mini-language
from datetime import datetime
now = datetime.now()
print(f"Current date: {now:%Y-%m-%d}") # Output: Current date: 2023-09-14
print(f"Current time: {now:%H:%M:%S}") # Output: Current time: 10:30:00

# Alignment and padding
text = "Python"
print(f"{text:>10}")  # Right-align in 10 spaces. Output:     Python
print(f"{text:<10}")  # Left-align in 10 spaces. Output: Python
print(f"{text:^10}")  # Center-align in 10 spaces. Output:   Python
print(f"{text:=^10}") # Center-align with fill character. Output: ==Python==

# Percentage formatting
ratio = 0.75
print(f"Completion: {ratio:.2%}") # Output: Completion: 75.00%
```

40. `functools` Gems

The `functools` module provides higher-order functions and operations on callable objects.

```python
from functools import partial, cached_property, wraps

# partial: Create a new function with some arguments pre-filled
def multiply(a, b):
    return a * b

double = partial(multiply, 2) # New function that always multiplies by 2
print(f"Double 5: {double(5)}") # Output: 10

# cached_property (Python 3.8+): Cache the result of a method call
class MyClass:
    def __init__(self, name):
        self.name = name
        self._heavy_computation_count = 0

    @cached_property
    def heavy_computation(self):
        self._heavy_computation_count += 1
        print("Performing heavy computation...")
        return f"Result of heavy computation for {self.name} (call
{self._heavy_computation_count})"

obj = MyClass("Example")
print(obj.heavy_computation) # First call, computes and caches
print(obj.heavy_computation) # Second call, uses cached result
print(obj.heavy_computation) # Third call, uses cached result

# wraps: Decorator to preserve metadata of the original function when using
decorators
# (Already demonstrated in the Decorators section)
```

41. Context Manager Protocol (When Rolling Your Own)

To create your own context manager without `contextlib.contextmanager`, you implement the `__enter__` and `__exit__` dunder methods.

```
class MyContextManager:
    def __init__(self, resource_name):
        self.resource_name = resource_name

    def __enter__(self):
        print(f"Entering context for {self.resource_name}")
        # Acquire the resource here
        return self.resource_name # Value returned by 'as' in 'with' statement

    def __exit__(self, exc_type, exc_val, exc_tb):
        print(f"Exiting context for {self.resource_name}")
        # Release the resource here
        if exc_type:
            print(f"An exception occurred: {exc_val}")
        return False # Propagate exception if False, suppress if True

with MyContextManager("Custom Resource") as res:
    print(f"Inside context, using: {res}")
    # raise ValueError("Something went wrong!") # Uncomment to test exception
handling

print("Outside context.")
```

Composition vs. Inheritance

- **Inheritance** (`is-a` relationship): Use when a class is a more specific version of another class (e.g., a `Labrador` is a `Dog`).
- **Composition** (`has-a` relationship): Use when a class is composed of other objects (e.g., a `Car` has an `Engine`). Prefer composition over inheritance for greater flexibility.

```
# Composition Example
class Engine:
    def start(self):
        return "Engine started."

class Car:
    def __init__(self):
        self.engine = Engine() # Car has an Engine

    def start(self):
        return self.engine.start()

my_car = Car()
print(my_car.start()) # Output: Engine started.
```

Corrections & Addendum (v12 Fixes)

■ Fixed Issues:
- Logging: ensure correct format string: logging.basicConfig(level=logging.INFO, format='%(asctime)s - %(levelname)s - %
- Typing Extras: use single quotes inside f-strings: print(f"Admin ID: {get_user_id('admin')}")
- Mutable default args example: corrected return variable to 'item_list'.
- Clean up duplicates: remove repeated Context Managers, Pathlib, Logging sections.
- Match/Pattern Matching: corrected header to Python 3.10+.

■ Recommended Improvements:
- HTTP requests: always use timeout and response.raise_for_status().
- Subprocess: note difference between Unix ('ls -l') and Windows ('dir').
- Regex: example to safely extract group or None.
- __name__ == '__main__': add example for script vs module usage.