[Python] [the Essentials]

Basic Syntax

- Indentation: Python uses indentation (usually 4 spaces) to define code blocks.
- Comments: Use # for single-line comments and """ or ''' for multi-line comments.
- Variables: Declared using variable_name = value. Python is dynamically typed.
- Print: Use print() to display output, e.g., print("Hello, World!").
- Input: Use input() to get user input as a string, e.g., name = input("Enter your name: ").

Data Types

- Numbers: int (e.g., 42), float (e.g., 3.14), complex (e.g., 2+3j).
- Strings: Enclosed in single quotes '' or double quotes "", e.g., "Hello", 'Python'.
- Lists: Ordered, mutable sequences enclosed in [], e.g., [1, 2, 3], ["a", "b", "c"].
- Tuples: Ordered, immutable sequences enclosed in (), e.g., (1, 2, 3), ("a", "b", "c").
- Sets: Unordered, unique elements enclosed in {}, e.g., {1, 2, 3}, {"a", "b", "c"}.
- Dictionaries: Key-value pairs enclosed in {}, e.g., {"name": "John", "age": 25}.
- Booleans: True and False.

Control Flow

- if, elif, else: Conditional statements, e.g., if x > 0: ... elif x < 0: ... else: ....
- for: Iteration over sequences, e.g., for item in list: ....
- while: Loops based on a condition, e.g., while condition: ....
- break, continue: Loop control statements, e.g., break to exit a loop, continue to skip an iteration.
- pass: Placeholder statement, used when no action is required.

By: Waleed Mousa

Functions

- Defined using `def function_name(parameters):`, e.g., `def greet(name): ...`.
- `return`: Used to return a value from a function, e.g., `return result`.
- `*args` and `**kwargs`: Used for variable-length arguments, e.g., `def func(*args, **kwargs): ...`.
- Lambda functions: Anonymous functions defined using `lambda arguments: expression`, e.g., `lambda x: x ** 2`.

Collections

- Lists:
  - `append()`: Add an element to the end of the list, e.g., `list.append(item)`.
  - `insert()`: Insert an element at a specific index, e.g., `list.insert(index, item)`.
  - `remove()`: Remove the first occurrence of an element, e.g., `list.remove(item)`.
  - `pop()`: Remove and return an element at a specific index, e.g., `item = list.pop(index)`.
  - `sort()`: Sort the list in-place, e.g., `list.sort()`.
  - `reverse()`: Reverse the order of elements in the list, e.g., `list.reverse()`.
- Tuples: Immutable, accessed using index, e.g., `tuple[index]`.
- Sets:
  - `add()`: Add an element to the set, e.g., `set.add(item)`.
  - `remove()`: Remove an element from the set, raises `KeyError` if not found, e.g., `set.remove(item)`.
  - `discard()`: Remove an element from the set if present, e.g., `set.discard(item)`.
  - `union()`: Return a new set with elements from both sets, e.g., `set1.union(set2)`.
  - `intersection()`: Return a new set with elements common to both sets, e.g., `set1.intersection(set2)`.
  - `difference()`: Return a new set with elements in the first set but not in the second, e.g., `set1.difference(set2)`.
- Dictionaries:
  - `keys()`: Return a view of dictionary keys, e.g., `dict.keys()`.
  - `values()`: Return a view of dictionary values, e.g., `dict.values()`.

By: Waleed Mousa

    - items(): Return a view of dictionary key-value pairs, e.g., dict.items().
    - get(): Return the value for a key if it exists, else return a default value, e.g., dict.get(key, default).
    - update(): Update the dictionary with key-value pairs from another dictionary or iterable, e.g., dict.update(other_dict).

File Handling

- Opening files: open(filename, mode), where mode can be 'r' (read), 'w' (write), 'a' (append), 'x' (exclusive creation), 'b' (binary), 't' (text), '+' (read/write).
- Reading files:
    - read(): Read the entire contents of a file as a string, e.g., content = file.read().
    - readline(): Read a single line from the file, e.g., line = file.readline().
    - readlines(): Read all lines from the file and return them as a list, e.g., lines = file.readlines().
- Writing to files:
    - write(): Write a string to the file, e.g., file.write("Hello, World!").
    - writelines(): Write a list of strings to the file, e.g., file.writelines(lines).
- Closing files: file.close() to close the file after reading/writing.
- Context managers: Use with statement to automatically close the file, e.g., with open(filename, mode) as file: ....

Exception Handling

- try, except, finally: Used to handle exceptions.
    - try: Contains code that may raise an exception.
    - except: Catches and handles specific exceptions.
    - finally: Executes code regardless of whether an exception occurred or not.
- raise: Used to raise an exception explicitly, e.g., raise ValueError("Invalid value").
- Common exceptions: ValueError, TypeError, IndexError, KeyError.

By: Waleed Mousa

Object-Oriented Programming (OOP)

- Classes: Defined using `class ClassName:`, e.g., `class Person: ...`.
- Objects: Instances of a class, created using `object_name = ClassName()`, e.g., `person = Person()`.
- Inheritance: Defined using `class ChildClass(ParentClass):`, e.g., `class Student(Person): ...`.
- Method overriding: Redefining methods in child classes, e.g., `def method(self): ...` in the child class.
- `self`: Reference to the instance of a class, used as the first parameter in instance methods.

Modules

- Importing modules: `import module_name`, e.g., `import math`, or `from module_name import function_name`, e.g., `from math import sqrt`.
- Creating modules: Save code in a `.py` file and import it using `import module_name`.
- Common modules:
  - `math`: Mathematical functions and constants, e.g., `math.pi`, `math.sqrt()`.
  - `random`: Generate pseudo-random numbers, e.g., `random.random()`, `random.randint(a, b)`.
  - `datetime`: Manipulate dates and times, e.g., `datetime.now()`, `datetime.timedelta()`.
  - `os`: Interact with the operating system, e.g., `os.getcwd()`, `os.listdir()`.
  - `sys`: Access system-specific parameters and functions, e.g., `sys.argv`, `sys.exit()`.

String Manipulation

- Concatenation: Using `+` operator, e.g., `"Hello, " + "World!"`.
- Formatting:
  - f-strings: `f"Hello, {name}!"`, e.g., `name = "John"; print(f"Hello, {name}!")`.
  - `str.format()`: `"Hello, {}!".format(name)`, e.g., `name = "John"; print("Hello, {}!".format(name))`.

By: Waleed Mousa

    - % operator: "Hello, %s!" % name, e.g., name = "John"; print("Hello, %s!" % name).
- Common methods:
    - lower(): Convert string to lowercase, e.g., "Hello".lower().
    - upper(): Convert string to uppercase, e.g., "hello".upper().
    - strip(): Remove leading and trailing whitespace, e.g., " hello ".strip().
    - split(): Split string into a list based on a delimiter, e.g., "a,b,c".split(",").
    - join(): Join elements of an iterable into a string using a delimiter, e.g., ",".join(["a", "b", "c"]).
    - replace(): Replace occurrences of a substring with another substring, e.g., "Hello".replace("l", "x").

List Comprehensions

- Syntax: [expression for item in iterable if condition].
- Used to create new lists based on existing iterables.
- Examples:
    - squares = [x ** 2 for x in range(1, 11)] creates a list of squares of numbers from 1 to 10.
    - even_numbers = [x for x in numbers if x % 2 == 0] creates a list of even numbers from the numbers list.

Decorators

- Syntax: @decorator_name above a function definition.
- Used to modify the behavior of functions or classes without directly modifying their code.
- Examples:
    - @timer decorator to measure the execution time of a function.
    - @cache decorator to cache the results of a function for specific inputs.

Regular Expressions

- Importing the re module: import re.
- Common methods:

    - search(): Search for the first occurrence of a pattern in a string, e.g., re.search(pattern, string).
    - match(): Check if the string starts with a pattern, e.g., re.match(pattern, string).
    - findall(): Find all occurrences of a pattern in a string, e.g., re.findall(pattern, string).
    - sub(): Replace occurrences of a pattern with a replacement string, e.g., re.sub(pattern, repl, string).
- Pattern syntax:
    - .: Matches any single character except a newline.
    - *: Matches zero or more occurrences of the preceding character or group.
    - +: Matches one or more occurrences of the preceding character or group.
    - ?: Matches zero or one occurrence of the preceding character or group.
    - ^: Matches the start of a string.
    - $: Matches the end of a string.
    - []: Defines a character set, e.g., [aeiou] matches any vowel.

Error Handling and Debugging

- Syntax errors: Occur when code violates Python's grammar rules, e.g., missing colons, incorrect indentation.
- Exceptions: Raised during the execution of a program when an error occurs, e.g., ZeroDivisionError, FileNotFoundError.
- Debugging techniques:
    - Use print() statements to display variable values and track program flow.
    - Use a debugger to set breakpoints, step through code, and inspect variables.
    - Use logging to record information during program execution, e.g., import logging; logging.debug(message).

Virtual Environments

- Creating virtual environments: python -m venv env_name creates a new virtual environment named env_name.
- Activating virtual environments:

    - Unix/Linux: source env_name/bin/activate.
    - Windows: env_name\Scripts\activate.
- Installing packages: pip install package_name installs a package within the active virtual environment.
- Deactivating virtual environments: deactivate deactivates the currently active virtual environment.

Iterators and Generators

- Iterators: Objects that can be iterated upon, e.g., lists, tuples, dictionaries.
- iter(): Creates an iterator object from an iterable, e.g., iter(list).
- next(): Retrieves the next item from an iterator, e.g., next(iterator).
- Generators: Functions that generate a sequence of values using yield instead of return.
- Generator expressions: Similar to list comprehensions but create generators, e.g., (x ** 2 for x in range(1, 11)).

Context Managers

- with statement: Used to wrap the execution of a block of code with methods defined by a context manager.
- Common use cases: File handling, database connections, acquiring and releasing locks.
- Example:

```
with open("file.txt", "r") as file:
     content = file.read()
```

Networking

- socket module: Provides low-level networking interface for creating socket connections.
- Creating a socket: socket.socket(socket.AF_INET, socket.SOCK_STREAM) creates a TCP/IP socket.
- Connecting to a server: socket.connect((host, port)) connects to a server.
- Sending data: socket.send(data) sends data through the socket.
- Receiving data: socket.recv(buffer_size) receives data from the socket.

By: Waleed Mousa

Web Development

- Flask framework: A lightweight web framework for building web applications.
    - Installing Flask: pip install flask.
    - Creating a Flask app:

```
from flask import Flask
app = Flask(__name__)

@app.route("/")
def hello():
      return "Hello, World!"

if __name__ == "__main__":
      app.run()
```

- Django framework: A high-level web framework for building complex web applications.
    - Installing Django: pip install django.
    - Creating a Django project: django-admin startproject project_name.
    - Creating a Django app: python manage.py startapp app_name.
    - Defining models, views, and templates to build web pages.

Data Analysis and Scientific Computing

- NumPy library: Provides support for large, multi-dimensional arrays and matrices, along with mathematical functions.
    - Installing NumPy: pip install numpy.
    - Creating arrays: np.array([1, 2, 3]), np.zeros((3, 3)), np.ones((2, 2)).
    - Array operations: np.sum(array), np.mean(array), np.dot(array1, array2).
- Pandas library: Provides data structures and data analysis tools for handling structured data.
    - Installing Pandas: pip install pandas.
    - Creating data frames: pd.DataFrame(data), pd.read_csv("file.csv").
    - Data manipulation: df.groupby(column), df.merge(other_df), df.fillna(value).
- Matplotlib library: Creates static, animated, and interactive visualizations.

    - Installing Matplotlib: `pip install matplotlib`.
    - Creating plots: `plt.plot(x, y)`, `plt.scatter(x, y)`, `plt.bar(x, height)`.
    - Customizing plots: `plt.title("Title")`, `plt.xlabel("X-axis")`, `plt.legend()`.

Concurrency and Parallelism

- `threading` module: Provides high-level threading interfaces for creating and managing threads.
    - Creating a thread: `threading.Thread(target=function, args=(arg1, arg2))`.
    - Starting a thread: `thread.start()`.
    - Waiting for a thread to finish: `thread.join()`.
- `multiprocessing` module: Provides support for spawning processes to leverage multiple CPUs.
    - Creating a process: `multiprocessing.Process(target=function, args=(arg1, arg2))`.
    - Starting a process: `process.start()`.
    - Waiting for a process to finish: `process.join()`.
- `asyncio` module: Provides infrastructure for writing single-threaded concurrent code using coroutines and event loops.
    - Defining a coroutine: `async def function(): ...`.
    - Running a coroutine: `asyncio.run(function())`.
    - Using `await` to wait for the completion of a coroutine.

Testing and Debugging

- `unittest` module: Provides a framework for writing and running tests.
    - Creating a test case: Define a class that inherits from `unittest.TestCase`.
    - Defining test methods: Write methods with names starting with `test_`.
    - Running tests: `unittest.main()` runs all the test methods in the test case.
- `pytest` framework: A popular testing framework with a simple and expressive syntax.
    - Installing pytest: `pip install pytest`.
    - Writing test functions: Define functions with names starting with `test_`.

By: Waleed Mousa

    - Running tests: `pytest` command runs all the test functions in the project.
- Debugging tools:
    - `pdb` module: Python's built-in debugger, allows stepping through code and inspecting variables.
    - IDEs with debugging support: PyCharm, Visual Studio Code, and others provide integrated debugging features.

Packaging and Distribution

- `setuptools` library: Provides tools for packaging and distributing Python projects.
    - Creating a `setup.py` file: Specify project metadata, dependencies, and entry points.
    - Building a distribution package: `python setup.py sdist` creates a source distribution package.
- `pip` package manager: Installs and manages Python packages.
    - Installing a package: `pip install package_name`.
    - Uninstalling a package: `pip uninstall package_name`.
    - Creating a requirements file: `pip freeze > requirements.txt` lists all installed packages and their versions.

Miscellaneous

- `os` module: Provides functions for interacting with the operating system.
    - `os.path`: Provides functions for working with file paths, e.g., `os.path.join()`, `os.path.exists()`.
    - `os.environ`: Provides access to environment variables, e.g., `os.environ.get("VARIABLE_NAME")`.
- `json` module: Provides functions for working with JSON data.
    - `json.dumps()`: Serializes a Python object to a JSON string.
    - `json.loads()`: Deserializes a JSON string to a Python object.
- `csv` module: Provides functions for reading and writing CSV files.
    - `csv.reader()`: Creates a reader object for reading CSV files.
    - `csv.writer()`: Creates a writer object for writing CSV files.
- `random` module: Provides functions for generating random numbers and sequences.
    - `random.random()`: Generates a random float between 0 and 1.

By: Waleed Mousa

- `random.randint(a, b)`: Generates a random integer between `a` and `b` (inclusive).
- `random.shuffle(sequence)`: Shuffles a sequence in-place.

Functional Programming

- `lambda` functions: Anonymous functions defined using the `lambda` keyword, e.g., `lambda x: x ** 2`.
- `map()`: Applies a function to each item in an iterable and returns an iterator of the results, e.g., `map(lambda x: x ** 2, [1, 2, 3])`.
- `filter()`: Filters an iterable based on a predicate function and returns an iterator of the filtered items, e.g., `filter(lambda x: x % 2 == 0, [1, 2, 3, 4])`.
- `reduce()`: Applies a function of two arguments cumulatively to the items of an iterable, reducing it to a single value, e.g., `reduce(lambda x, y: x + y, [1, 2, 3, 4])`.

Decorators with Arguments

- Decorators that accept arguments: Decorators that take arguments to customize their behavior.
- Example:
  def repeat(times):

```
    def decorator(func):
    def wrapper(*args, **kwargs):
          for _ in range(times):
                result = func(*args, **kwargs)
          return result
    return wrapper
    return decorator

@repeat(3)
def greet(name):
    print(f"Hello, {name}!")
```