TIME Module in Python:

The `time` module in Python provides various functions to work with time-related tasks, such as getting the current time, pausing execution, converting between different time representations, and measuring elapsed time. Here's an overview of some commonly used functions and constants in the `time` module:

Getting Current Time:

- **`time()`: Returns the current time in seconds since the epoch (January 1, 1970, 00:00:00 UTC) as a floating-point number.**

```python
import time

current_time = time.time()
print("Current time:", current_time)
```

Converting Time:

- **`ctime()`: Converts a timestamp (seconds since the epoch) to a human-readable string representation of the local time.**

python

```python
import time

current_time = time.time()
print("Current time:", time.ctime(current_time))
```

Pausing Execution:

- **`sleep(seconds)`: Suspends execution of the current thread for the given number of seconds.**


```python
import time

print("Start")
time.sleep(2)  # Pause execution for 2 seconds
print("End")
```

Formatting Time:

- `strftime(format, time_tuple)`: **Converts a time tuple or a struct_time object to a string representation based on the format specified.**

```python
import time

current_time = time.localtime()
formatted_time = time.strftime("%Y-%m-%d %H:%M:%S", current_time)
print("Formatted time:", formatted_time)
```

Measure Elapsed Time:

- `perf_counter()`: **Returns a high-resolution time value (in fractional seconds) intended for use in measuring short durations.**

```python
import time

start_time = time.perf_counter()
# Code to measure elapsed time
end_time = time.perf_counter()
elapsed_time = end_time - start_time
print("Elapsed time:", elapsed_time)
```

Constants:

- `time.localtime()`: **Returns the current time as a struct_time object, representing the local time.**
- `time.gmtime()`: **Returns the current time as a struct_time object, representing the UTC time.**


```
import time

local_time = time.localtime()
print("Local time:", local_time)

utc_time = time.gmtime()
print("UTC time:", utc_time)
```

These are just a few examples of what you can do with the `time` module in Python. It's a versatile module that's useful for a wide range of time-related tasks in Python programming.

CODE:

```
import time
initial = time.time()
k=0
while(k<45):
    print("This is pentest diaries")
    time.sleep(2)
    k+=1
print("While loop ran in ",time.time() - initial,"seconds")
initial2 = time.time()
for i in range(45):
    print("This is pentest diaries")
print("For loop ran in",time.time() - initial2, "seconds")
```

Output:

```
/root/PycharmProjects/pythonProject/.venv/bin/python /root/PycharmProjects/pythonProject/26.py
This is pentest diaries
This is pentest diaries
This is pentest diaries
This is pentest diaries
This is pentest diaries
This is pentest diaries
This is pentest diaries
This is pentest diaries
This is pentest diaries
This is pentest diaries
```

VIRTUAL ENVIRONMENT & REQUIREMENT:

In Python, virtual environments and requirements files are essential tools for managing project dependencies and isolating project environments. Here's a brief overview of each:

1. **Virtual Environments:**

A virtual environment is a self-contained directory tree that contains a Python installation for a particular version of Python, as well as a number of additional packages. It allows you to work on a Python project without affecting the system-wide Python installation or other projects.

To create a virtual environment, you can use the built-in `venv` module in Python 3:

```
python3 -m venv myenv
```

This command creates a new virtual environment named `myenv`. You can activate it by running:

On Windows:

```
myenv\Scripts\activate
```

On Unix or MacOS:

```
source myenv/bin/activate
```

1. **Once activated, any Python commands you run will use the Python interpreter and packages installed within the virtual environment.**

2. **Requirements Files:**

A requirements file is a simple text file that lists all the Python packages that your project depends on, along with their versions. This file allows you to specify exactly which packages and versions are required for your project, making it easier to reproduce the environment on another machine.

You can generate a requirements file that captures the current state of your virtual environment using the `pip freeze` command:

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

This will create a file named `requirements.txt` containing a list of all installed packages and their versions. You can then share this file with others, and they can use it to install the exact same set of dependencies by running:

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

This ensures that everyone working on the project is using the same versions of the dependencies, reducing the chances of compatibility issues.

By combining virtual environments with requirements files, you can create isolated and reproducible development environments for your Python projects.

CODE:

Numpy==1.15.4

Scikit-learn==0.20.1

Scipy==1.1.0

Sklearn==0.0

Commands:

PS C:\> pip install virtualenv

PS C:\> virtualenv

PS C:\> virtualenv test

PS C:\> .\test\scripts\activate

PS C:\> set-executionpolicy remotesigned

PS C:\> pip install package

PS C:\> pip uinstall package

PS C:\> pip freeze > requirements.txt

PS C:\> pip install –r .\requirements.txt

PS C:\> deactive

PS C:\> virtualenv –system-site-packages test

ENUMERATE FUNCTION:

The `enumerate()` function in Python is a built-in function that allows you to loop over an iterable (such as a list, tuple, or string) while also keeping track of the index of the current item. It returns an enumerate object, which yields pairs of indices and corresponding values from the iterable.

Here's the syntax of the `enumerate()` function:

```
enumerate(iterable, start=0)
```

- **`iterable`: The iterable (list, tuple, string, etc.) that you want to loop over.**
- **`start` (optional): The index at which to start counting. By default, it is 0.**

Here's a simple example to demonstrate how `enumerate()` works:

```
my_list = ['apple', 'banana', 'cherry', 'date']

for index, fruit in enumerate(my_list):
    print(f"Index: {index}, Fruit: {fruit}")
```

Output:

```
Index: 0, Fruit: apple
Index: 1, Fruit: banana
Index: 2, Fruit: cherry
Index: 3, Fruit: date
```

In this example, the `enumerate()` function is used to iterate over the `my_list` list. In each iteration, it returns a tuple containing the index of the current item and the item itself. This tuple is then unpacked into the variables `index` and `fruit`, which are then used in the loop body.

You can also specify a starting index other than 0 if needed:

```
for index, fruit in enumerate(my_list, start=1):
    print(f"Index: {index}, Fruit: {fruit}")
```

Output:

```
Index: 1, Fruit: apple
Index: 2, Fruit: banana
Index: 3, Fruit: cherry
Index: 4, Fruit: date
```

In this case, the enumeration starts from 1 instead of the default 0.

CODE:

```
l1 = ["tea","coffee","milk","bund"]

for index,item in enumerate(l1):
    if index%2==0:
        print(f"Jarvis please buy {item}")
```

Output:

```
/root/PycharmProjects/pythonProject/.venv/bin/python /root/PycharmProjects/pythonProject/26_2.py
Jarvis please buy tea
Jarvis please buy milk
```