powerpyro

Energy Consumption Monitoring
for Code Execution

Technical Documentation

API Reference

Módulo Monitor

`Monitor`

Class responsible for monitoring the energy consumption of selected hardware components (CPU, GPU, and memory) in a system.

Attributes:

Name	Type	Description
`__operating_system`	`OsType`	The current operating system.
`__components`	`Dict[str, HardwareComponent]`	Dictionary of initialized
`__stop_sign`	`bool`	Flag to stop the monitoring loop.
`__thread`	`Thread`	Thread in which monitoring occurs.
`__WATT_TO_KWH`	`float`	Constant to convert energy from

Source code in `power_pyro\monitor.py`

~~~python
class Monitor():
    """
    Class responsible for monitoring the energy consumption of
    selected hardware components
    (CPU, GPU, and memory) in a system.

    Attributes:
        __operating_system (OsType): The current operating system.
        __components (Dict[str, HardwareComponent]): Dictionary of
initialized
        hardware components.
        __stop_sign (bool): Flag to stop the monitoring loop.
        __thread (Thread): Thread in which monitoring occurs.
        __WATT_TO_KWH (float): Constant to convert energy from
        watts to kilowatt-hours.
    """
    def __init__(self, required_components: Dict[str, bool]):
        """
        Initializes the Monitor class by setting up the operating system,
        validating required components, creating component instances,
        and preparing the monitoring thread.

        Args:
            required_components (Dict[str, bool]): Dictionary specifying
which
            components ('cpu', 'gpu', 'memory') should be monitored.

        Raises:
            InvalidKeysErrorException: If any invalid keys are found in
            the provided dictionary.

        Example:
            Basic usage monitoring only CPU:

            ```python
            from power_pyro import Monitor

            monitor = Monitor({'cpu': True, 'gpu': False, 'memory':
False})
            ~~~

            Monitoring CPU and GPU:

            ```python
            from power_pyro import Monitor

            monitor = Monitor({'cpu': True, 'gpu': True, 'memory':
False})
            ```

            Monitoring all components:

            ```python
            from power_pyro import Monitor

            monitor = Monitor({'cpu': True, 'gpu': True, 'memory': True})
            ```
```
```

~~~python
        """
        self.__operating_system: OsType = self.__get_operating_system()
        self.__components: Dict[str, HardwareComponent] =
self.__create_components(required_components)
        self.__stop_sign: bool = False
        self.__thread:Thread = Thread(target=self.__monitor)
        self.__WATT_TO_KWH:float = 3_600_000

    def __get_operating_system(self) -> OsType:
        """Determines the operating system type.

        Returns:
            The operating system type as OsType.

        Raises:
            OSError: If the OS cannot be identified.
        """
        if os.name == 'nt':
            return OsType.WINDOWS
        elif os.name == 'posix':
            return OsType.LINUX
        else:
            raise OSError("Unable to identify operating system")

    def get_monitored_components(self) -> Dict[str, Any]:
        """Retrieves the components to be monitored.

        Returns:
            A dictionary with the monitoring status of the components
            and the components themselves.

        Example:
            Example retrieving monitored components:

            ```python
            from power_pyro import Monitor

            # Monitor only CPU and GPU
            monitor = Monitor({'cpu': True, 'gpu': True, 'memory':
False})
            components = monitor.get_monitored_components()

            print(components)
            # Output: {'cpu': True, 'gpu': True, 'memory': False}
            ~~~
        """
        monitored_components:Dict[str, Any] = {'cpu': {'component': None,
'monitored': False},
                                               'gpu': {'component': None,
'monitored': False},
                                               'memory': {'component':
None, 'monitored': False}}

        for key in self.__components.keys():
            monitored_components[key]['component'] =
self.__components[key]
            monitored_components[key]['monitored'] = True

        return monitored_components

    def __check_components(self, required_components: Dict[str, bool]) ->
```
```

```python
bool:
        """Validates the required components keys.

        Args:
            required_components: Dictionary of required components.

        Returns:
            bool:
                - 'True' if all the dictionary keys are in the list,
                - 'False' otherwise.
        """
        required_keys = ['cpu', 'gpu', 'memory']

        return len(required_components.keys()) <= len(required_keys) and
all(key in required_keys for key in required_components)

    def __create_components(self, required_components: Dict[str, bool]) -
> Dict[str, HardwareComponent]:
        """Creates the required hardware components using the appropriate
factories.

        Args:
            required_components: Dictionary indicating which components
should be created.

        Returns:
            components: Dictionary containing the created hardware
components.

        Raises:
            InvalidKeysErrorException: If the required components contain
invalid keys.
        """
        if not self.__check_components(required_components):
            raise InvalidKeysErrorException()

        factories: Dict[str, HardwareComponentFactory] = {
            'cpu': CpuComponentFactory(),
            'gpu': GpuComponentFactory(),
            'memory': MemoryComponentFactory()
        }

        components: Dict[str, HardwareComponent] = {}

        for component in required_components:
            components[component] =
factories[component].create_component(self.__operating_system)

            if hasattr(components[component], 'open'):
                components[component].open()

        return components

    def __close_resources(self) -> None:
        """Closes resources allocated by the components."""
        for component in self.__components:

            if hasattr(self.__components[component], 'close'):
                self.__components[component].close()

    def get_energy_consumed_by_components(self) -> Dict[str, float]:
```

~~~
        """Retrieves the total energy consumed by each hardware
component.

        Returns:
            energy_consumed_by_components: A dictionary where the keys
are component names ('cpu', 'gpu', 'memory') and the values are the
energy consumed by each component.

        Example:
            ```python
            monitor = Monitor({'cpu': True, 'gpu': False})
            result = monitor.get_energy_consumed_by_components()
            print(result)  # {'cpu': 2.5}
            ~~~
        """
        energy_consumed_by_components: Dict[str, float] = {}

        if 'cpu' in self.__components:
            energy_consumed_by_components['cpu'] =
self.__components['cpu'].total_energy_consumed

            if 'gpu' in self.__components:
                energy_consumed_by_components['gpu'] =
self.__components['gpu'].total_energy_consumed

                if 'memory' in self.__components:
                    energy_consumed_by_components['memory'] =
self.__components['memory'].total_energy_consumed

            if 'memory' in self.__components:
                energy_consumed_by_components['memory'] =
self.__components['memory'].total_energy_consumed

        return energy_consumed_by_components

    def total_energy_consumed(self) -> float:
        """Retrieves the total energy consumed by all components
monitored.

        Example:
            Get total energy consumption after monitoring:

            ```python
            from power_pyro import Monitor

            monitor = Monitor({'cpu': True, 'gpu': True, 'memory':
False})
            monitor.start()
            # ... perform operations ...
            monitor.end()

            total_energy = monitor.total_energy_consumed()
            print(f"Total energy consumed: {total_energy:.2f} Wh")
            ```

        """

        total_energy_consumed: float = 0.0

        for component in self.__components:
            total_energy_consumed +=
```
```

~~~python
self.__components[component].total_energy_consumed

        return total_energy_consumed

    def __monitor(self) -> None:
        """Monitors energy consumption of components at regular
intervals."""

        while not self.__stop_sign:
            start = time.time()

            time.sleep(10)

            end = time.time()

            period = end - start

            if 'cpu' in self.__components:
                cpu = self.__components['cpu']
                cpu.update_energy_consumed((cpu.get_power() *
cpu.get_cpu_percent_for_process() * period)/self.__WATT_TO_KWH)

            if 'gpu' in self.__components:
                gpu = self.__components['gpu']
                gpu.update_energy_consumed((gpu.get_power() *
period)/self.__WATT_TO_KWH)

            if 'memory' in self.__components:
                memory = self.__components['memory']
                memory.update_energy_consumed((memory.get_power() *
period)/self.__WATT_TO_KWH)

        if self.__operating_system == OsType.WINDOWS:
            self.__close_resources()

    def start(self) -> None:
        """
        Starts the monitoring process in a separate thread.

        Example:
            Start the monitoring process:

            ```python
            from power_pyro import Monitor

            monitor = Monitor({'cpu': True, 'gpu': False, 'memory':
True})
            monitor.start()
            ~~~
        """
        self.__thread.start()

    def is_running(self) -> bool:
        """
        Checks if the monitoring process is currently running.

        Returns:
            bool: True if the monitoring thread is alive (running), False
otherwise.

        Example:
```
```

~~~
                Verify if the monitoring process is still running:

                ```python
                from power_pyro import Monitor

                monitor = Monitor({'cpu': True, 'gpu': False, 'memory':
True})
                monitor.start()
                print(monitor.is_running())  # True, since monitoring has
started
                monitor.end()
                print(monitor.is_running())  # False, since monitoring has
ended
                ~~~
            """
            return self.__thread.is_alive()

        def end(self) -> None:
            """
            Stops the monitoring process and waits for the monitoring thread
to finish.

            Example:
                Stop the monitoring process:

                ```python
                from power_pyro import Monitor

                monitor = Monitor({'cpu': True, 'gpu': False, 'memory':
True})
                monitor.start()
                # ... perform operations to monitor ...
                monitor.end()  # Stops monitoring and waits for thread to
finish
                ```
            """
            self.__stop_sign = True
            self.__thread.join()
```

`__check_components(required_components)`

Validates the required components keys.

Parameters:

Name	Type	Description	Default
`required_components`	`Dict[str, bool]`	Dictionary of required components.	*required*

Returns:
```

| Name | Type | Description |
|---|---|---|
| `bool` | `bool` | • 'True' if all the dictionary keys are in the list,<br>• 'False' otherwise. |

**Source code in `power_pyro\monitor.py`**

```python
def __check_components(self, required_components: Dict[str, bool]) ->
bool:
    """Validates the required components keys.

    Args:
        required_components: Dictionary of required components.

    Returns:
        bool:
            - 'True' if all the dictionary keys are in the list,
            - 'False' otherwise.
    """
    required_keys = ['cpu', 'gpu', 'memory']

    return len(required_components.keys()) <= len(required_keys) and
all(key in required_keys for key in required_components)
```

`__close_resources()`

Closes resources allocated by the components.

Source code in `power_pyro\monitor.py`

```python
def __close_resources(self) -> None:
    """Closes resources allocated by the components."""
    for component in self.__components:

        if hasattr(self.__components[component], 'close'):
            self.__components[component].close()
```

`__create_components(required_components)`

Creates the required hardware components using the appropriate factories.

Parameters:

Name	Type	Description	Default
`required_components`	`Dict[str, bool]`	Dictionary indicating which components should be created.	*required*

Returns:

Name	Type	Description
`components`	`Dict[str, HardwareComponent]`	Dictionary containing the created hardware components.

Raises:

Type	Description
`InvalidKeysErrorException`	If the required components contain invalid keys.

Source code in `power_pyro\monitor.py`

```python
def __create_components(self, required_components: Dict[str, bool]) ->
Dict[str, HardwareComponent]:
    """Creates the required hardware components using the appropriate
factories.

    Args:
        required_components: Dictionary indicating which components
should be created.

    Returns:
        components: Dictionary containing the created hardware
components.

    Raises:
        InvalidKeysErrorException: If the required components contain
invalid keys.
    """
    if not self.__check_components(required_components):
        raise InvalidKeysErrorException()

    factories: Dict[str, HardwareComponentFactory] = {
        'cpu': CpuComponentFactory(),
        'gpu': GpuComponentFactory(),
        'memory': MemoryComponentFactory()
    }

    components: Dict[str, HardwareComponent] = {}

    for component in required_components:
        components[component] =
factories[component].create_component(self.__operating_system)

        if hasattr(components[component], 'open'):
            components[component].open()

    return components
```

`__get_operating_system()`

Determines the operating system type.

Returns:

Type	Description
`OsType`	The operating system type as OsType.

Raises:

Type	Description
`OSError`	If the OS cannot be identified.

Source code in `power_pyro\monitor.py`

```python
def __get_operating_system(self) -> OsType:
    """Determines the operating system type.

    Returns:
        The operating system type as OsType.

    Raises:
        OSError: If the OS cannot be identified.
    """
    if os.name == 'nt':
        return OsType.WINDOWS
    elif os.name == 'posix':
        return OsType.LINUX
    else:
        raise OSError("Unable to identify operating system")
```

`__init__(required_components)`

Initializes the Monitor class by setting up the operating system, validating required components, creating component instances, and preparing the monitoring thread.

Parameters:

Name	Type	Description	Default
`required_components`	`Dict[str, bool]`	Dictionary specifying which	*required*

Raises:

Type	Description
`InvalidKeysErrorException`	If any invalid keys are found in

Example

Basic usage monitoring only CPU:

```
from power_pyro import Monitor

monitor = Monitor({'cpu': True, 'gpu': False, 'memory': False})
```

Monitoring CPU and GPU:

```
from power_pyro import Monitor

monitor = Monitor({'cpu': True, 'gpu': True, 'memory': False})
```

Monitoring all components:

```
from power_pyro import Monitor

monitor = Monitor({'cpu': True, 'gpu': True, 'memory': True})
```

Source code in `power_pyro\monitor.py`

~~~python
def __init__(self, required_components: Dict[str, bool]):
    """
    Initializes the Monitor class by setting up the operating system,
    validating required components, creating component instances,
    and preparing the monitoring thread.

    Args:
        required_components (Dict[str, bool]): Dictionary specifying which
        components ('cpu', 'gpu', 'memory') should be monitored.

    Raises:
        InvalidKeysErrorException: If any invalid keys are found in
        the provided dictionary.

    Example:
        Basic usage monitoring only CPU:

        ```python
        from power_pyro import Monitor

        monitor = Monitor({'cpu': True, 'gpu': False, 'memory': False})
        ~~~

        Monitoring CPU and GPU:

        ```python
        from power_pyro import Monitor

        monitor = Monitor({'cpu': True, 'gpu': True, 'memory': False})
        ```

        Monitoring all components:

        ```python
        from power_pyro import Monitor

        monitor = Monitor({'cpu': True, 'gpu': True, 'memory': True})
        ```
    """
    self.__operating_system: OsType = self.__get_operating_system()
    self.__components: Dict[str, HardwareComponent] =
self.__create_components(required_components)
    self.__stop_sign: bool = False
    self.__thread:Thread = Thread(target=self.__monitor)
    self.__WATT_TO_KWH:float = 3_600_000
```

`__monitor()`

Monitors energy consumption of components at regular intervals.
```

Source code in `power_pyro\monitor.py`

```python
def __monitor(self) -> None:
    """Monitors energy consumption of components at regular intervals."""

    while not self.__stop_sign:
        start = time.time()

        time.sleep(10)

        end = time.time()

        period = end - start

        if 'cpu' in self.__components:
            cpu = self.__components['cpu']
            cpu.update_energy_consumed((cpu.get_power() *
cpu.get_cpu_percent_for_process() * period)/self.__WATT_TO_KWH)

        if 'gpu' in self.__components:
            gpu = self.__components['gpu']
            gpu.update_energy_consumed((gpu.get_power() *
period)/self.__WATT_TO_KWH)

        if 'memory' in self.__components:
            memory = self.__components['memory']
            memory.update_energy_consumed((memory.get_power() *
period)/self.__WATT_TO_KWH)

    if self.__operating_system == OsType.WINDOWS:
        self.__close_resources()
```

`end()`

Stops the monitoring process and waits for the monitoring thread to finish.

Example

Stop the monitoring process:

```python
from power_pyro import Monitor

monitor = Monitor({'cpu': True, 'gpu': False, 'memory': True})
monitor.start()
# ... perform operations to monitor ...
monitor.end()  # Stops monitoring and waits for thread to finish
```

Source code in `power_pyro\monitor.py`

~~~python
def end(self) -> None:
    """
    Stops the monitoring process and waits for the monitoring thread to
finish.

    Example:
        Stop the monitoring process:

        ```python
        from power_pyro import Monitor

        monitor = Monitor({'cpu': True, 'gpu': False, 'memory': True})
        monitor.start()
        # ... perform operations to monitor ...
        monitor.end()  # Stops monitoring and waits for thread to finish
        ~~~
    """
    self.__stop_sign = True
    self.__thread.join()

`get_energy_consumed_by_components()`

Retrieves the total energy consumed by each hardware component.

Returns:

Name	Type	Description
`energy_consumed_by_components`	`Dict[str, float]`	A dictionary where the keys are component names ('cpu', 'gpu', 'memory') and the values are the energy consumed by each component.

Example

```python
monitor = Monitor({'cpu': True, 'gpu': False})
result = monitor.get_energy_consumed_by_components()
print(result)  # {'cpu': 2.5}
```

Source code in `power_pyro\monitor.py`

~~~python
def get_energy_consumed_by_components(self) -> Dict[str, float]:
    """Retrieves the total energy consumed by each hardware component.

    Returns:
        energy_consumed_by_components: A dictionary where the keys are
component names ('cpu', 'gpu', 'memory') and the values are the energy
consumed by each component.

    Example:
        ```python
        monitor = Monitor({'cpu': True, 'gpu': False})
        result = monitor.get_energy_consumed_by_components()
        print(result)  # {'cpu': 2.5}
        ~~~
    """
    energy_consumed_by_components: Dict[str, float] = {}

    if 'cpu' in self.__components:
        energy_consumed_by_components['cpu'] =
self.__components['cpu'].total_energy_consumed

        if 'gpu' in self.__components:
            energy_consumed_by_components['gpu'] =
self.__components['gpu'].total_energy_consumed

            if 'memory' in self.__components:
                energy_consumed_by_components['memory'] =
self.__components['memory'].total_energy_consumed

        if 'memory' in self.__components:
            energy_consumed_by_components['memory'] =
self.__components['memory'].total_energy_consumed

    return energy_consumed_by_components
```

## `get_monitored_components()`

Retrieves the components to be monitored.

**Returns:**

| Type | Description |
|---|---|
| `Dict[str, Any]` | A dictionary with the monitoring status of the components |
| `Dict[str, Any]` | and the components themselves. |
```

Example

Example retrieving monitored components:

```
from power_pyro import Monitor

# Monitor only CPU and GPU
monitor = Monitor({'cpu': True, 'gpu': True, 'memory': False})
components = monitor.get_monitored_components()

print(components)
# Output: {'cpu': True, 'gpu': True, 'memory': False}
```

Source code in `power_pyro\monitor.py`

~~~
def get_monitored_components(self) -> Dict[str, Any]:
    """Retrieves the components to be monitored.

    Returns:
        A dictionary with the monitoring status of the components
        and the components themselves.

    Example:
        Example retrieving monitored components:

        ```python
        from power_pyro import Monitor

        # Monitor only CPU and GPU
        monitor = Monitor({'cpu': True, 'gpu': True, 'memory': False})
        components = monitor.get_monitored_components()

        print(components)
        # Output: {'cpu': True, 'gpu': True, 'memory': False}
        ```
    """
    monitored_components:Dict[str, Any] = {'cpu': {'component': None,
'monitored': False},
                                            'gpu': {'component': None,
'monitored': False},
                                            'memory': {'component': None,
'monitored': False}}

    for key in self.__components.keys():
        monitored_components[key]['component'] = self.__components[key]
        monitored_components[key]['monitored'] = True

    return monitored_components
~~~

`is_running()`

Checks if the monitoring process is currently running.

Returns:

Name	Type	Description
`bool`	`bool`	True if the monitoring thread is alive (running), False otherwise.

Example

Verify if the monitoring process is still running:

```
from power_pyro import Monitor

monitor = Monitor({'cpu': True, 'gpu': False, 'memory': True})
monitor.start()
print(monitor.is_running())  # True, since monitoring has started
monitor.end()
print(monitor.is_running())  # False, since monitoring has ended
```

Source code in `power_pyro\monitor.py`

~~~
def is_running(self) -> bool:
    """
    Checks if the monitoring process is currently running.

    Returns:
        bool: True if the monitoring thread is alive (running), False
otherwise.

    Example:
        Verify if the monitoring process is still running:

        ```python
        from power_pyro import Monitor

        monitor = Monitor({'cpu': True, 'gpu': False, 'memory': True})
        monitor.start()
        print(monitor.is_running())  # True, since monitoring has started
        monitor.end()
        print(monitor.is_running())  # False, since monitoring has ended
        ```
    """
    return self.__thread.is_alive()
~~~

`start()`

Starts the monitoring process in a separate thread.

Example

Start the monitoring process:

```
from power_pyro import Monitor

monitor = Monitor({'cpu': True, 'gpu': False, 'memory': True})
monitor.start()
```

Source code in `power_pyro\monitor.py`

~~~
def start(self) -> None:
    """
    Starts the monitoring process in a separate thread.

    Example:
        Start the monitoring process:

        ```python
        from power_pyro import Monitor

        monitor = Monitor({'cpu': True, 'gpu': False, 'memory': True})
        monitor.start()
        ```
    """
    self.__thread.start()
~~~

`total_energy_consumed()`

Retrieves the total energy consumed by all components monitored.

Example

Get total energy consumption after monitoring:

```
from power_pyro import Monitor

monitor = Monitor({'cpu': True, 'gpu': True, 'memory': False})
monitor.start()
# ... perform operations ...
monitor.end()

total_energy = monitor.total_energy_consumed()
print(f"Total energy consumed: {total_energy:.2f} Wh")
```

Source code in `power_pyro\monitor.py`

~~~python
def total_energy_consumed(self) -> float:
    """Retrieves the total energy consumed by all components monitored.

    Example:
        Get total energy consumption after monitoring:

        ```python
        from power_pyro import Monitor

        monitor = Monitor({'cpu': True, 'gpu': True, 'memory': False})
        monitor.start()
        # ... perform operations ...
        monitor.end()

        total_energy = monitor.total_energy_consumed()
        print(f"Total energy consumed: {total_energy:.2f} Wh")
        ~~~

    """

    total_energy_consumed: float = 0.0

    for component in self.__components:
        total_energy_consumed +=
self.__components[component].total_energy_consumed

    return total_energy_consumed
```

Módulo CPU

`Cpu`

Bases: `ProcessingUnit`

Represents a Central Processing Unit (CPU) responsible for accessing and retrieving power consumption valuesfrom hardware sensors.

Attributes:

Name	Type	Description
`__manufacturer`	`CpuType`	CPU type.
```

**Source code in `power_pyro\cpu.py`**

~~~python
class Cpu(ProcessingUnit):
    """Represents a Central Processing Unit (CPU) responsible
       for accessing and retrieving power consumption values
       from hardware sensors.

    Attributes:
        __manufacturer (CpuType): CPU  type.
    """
    def __init__(self, operating_system: OsType):
        super().__init__(operating_system)
        self.__manufacturer: CpuType

        if operating_system == OsType.WINDOWS:
            self.computer.IsCpuEnabled = True

        self._update_manufacture()
        self._update_hardware_name()

    @property
    def get_manufacturer(self) -> CpuType:
        """ The hardware manufacturer.

        Returns:
            CpuType: A type of CPU.

        Example:
            ```python

            from monitor import Monitor

            monitor = Monitor({'cpu': True})

            components = monitor.get_monitored_components()
            print(components['cpu']['component'].get_manufacturer)
#CpuType.INTEL

            ~~~
        """
        return self.__manufacturer

    def _update_manufacture(self) -> None:
        if self.operating_system == OsType.WINDOWS:
            self.__update_manufacture_windows()

        elif self.operating_system == OsType.LINUX:
            self.__update_manufacture_linux()
        else:
            raise OSError("Unable to identify operating system")

    def __update_manufacture_windows(self) -> None:
        """Update hardware manufacturer when running on Windows OS.

        Raises:
            IdentifyHardwareManufacturerException: If the hardware
manufacturer
            cannot be identified.
        """
```
```

```python

        try:
            wmi_session = wmi.WMI()

            manufacturer = wmi_session.Win32_Processor()[0].Manufacturer

            if manufacturer == 'GenuineIntel':
                self.__manufacturer = CpuType.INTEL
            elif manufacturer == 'AuthenticAMD':
                self.__manufacturer = CpuType.AMD
            else:
                raise IdentifyHardwareManufacturerException(HT.CPU)
        except (ModuleNotFoundError, wmi.x_wmi, IndexError,
AttributeError):
            raise IdentifyHardwareManufacturerException(HT.CPU)

    def __update_manufacture_linux(self) -> None:
        """Update hardware manufacturer when running on Linux OS.

        Raises:
            IdentifyHardwareManufacturerException: If the hardware
manufacturer
            cannot be identified.
        """

        try:
            info = cpuinfo.get_cpu_info()

            manufacturer = info['vendor_id_raw']

            if manufacturer == 'GenuineIntel':
                self.__manufacturer = CpuType.INTEL
            elif manufacturer == 'AuthenticAMD':
                self.__manufacturer = CpuType.AMD
            else:
                raise IdentifyHardwareManufacturerException(HT.CPU)
        except (ModuleNotFoundError, KeyError):
            raise IdentifyHardwareManufacturerException(HT.CPU)

    def _update_hardware_name(self) -> None:
        if self.operating_system == OsType.WINDOWS:
            self.__update_hardware_name_windows()
        elif self.operating_system == OsType.LINUX:
            self.__update_hardware_name_linux()
        else:
            raise OSError("Unable to identify operating system")

    def __update_hardware_name_windows(self) -> None:
        """ Set the CPU name.

        Raises:
            HardwareNameIdentifyException: Unable to identify CPU name in
Windows.
        """

        try:
            wmi_session = wmi.WMI()

            self.set_name = wmi_session.Win32_Processor()[0].Name
        except (ModuleNotFoundError, wmi.x_wmi, IndexError,
AttributeError):
```

```python
            raise HardwareNameIdentifyException(HT.CPU)

    def __update_hardware_name_linux(self) -> None:
        """ Set the CPU name.

        Raises:
            HardwareNameIdentifyException: Unable to identify CPU name in Linux.
        """

        try:
            self.set_name = cpuinfo.get_cpu_info()['brand_raw']
        except (ModuleNotFoundError, KeyError):
            raise HardwareNameIdentifyException(HT.CPU)

    def get_power(self) -> float:
        if self.operating_system == OsType.WINDOWS:
            return self.__get_power_on_windows()

        else:
            return self.__get_power_on_linux()

    def __get_power_on_linux(self) -> float:
        """ Returns the value of the CPU power in W in Linux.

        Returns:
            float: CPU power.
        """

        try:
            command = ["sudo", "perf", "stat", "-e", "power/energy-pkg/", "sleep", "0.1"]
            power = subprocess.run(command, capture_output=True, text=True)

            power = power.stderr.split(" ")
            power = [string for string in power if string.strip()]

            for index, string in enumerate(power):
                if string.find('\n\n') != -1:
                    power = power[index + 1]
                    power = power.replace(",", ".")
                    break

            power = float(power)/0.1
        except (PermissionError, subprocess.SubprocessError, AttributeError, IndexError, ValueError) as e:
            print('Error getting power from CPU: ', str(e))

        return power

    def __get_power_on_windows(self) -> float:
        """ Returns the value of the CPU power in W in Windows.

        Returns:
            float: CPU power.
        """
        try:
            cpu = next((hardware for hardware in self.computer.Hardware if hardware.HardwareType == HardwareType.Cpu), None)
            cpu.Update()
```

~~~python
            time.sleep(0.1)

            power = next((sensor for sensor in cpu.Sensors if
sensor.SensorType == SensorType.Power and (sensor.Name == "CPU Package"
or sensor.Name == "Package")))
            power = power.Value
        except AttributeError as e:
            print('Error getting power from CPU: ', str(e))

        return power

    def get_cpu_percent_for_process(self) -> float:
        """ Returns the percentage value of the monitored process on the
CPU.

        Returns:
            float: CPU percent.

        Example:
            ```python

            from monitor import Monitor

            monitor = Monitor({'cpu': True})
            components = monitor.get_monitored_components()
            print(components['cpu']
['component'].get_cpu_percent_for_process()) # 0.37
            ~~~
        """
        script_pid = os.getpid()
        sum_all = 0
        cpu_percent = 0
        for process in psutil.process_iter():
            try:
                with process.oneshot():
                    process_pid = process.pid
                    process_cpu_percent = process.cpu_percent()
                    print(type(process))

                    if process_pid:
                        sum_all += process_cpu_percent

                        if process_pid == script_pid:
                            cpu_percent += process_cpu_percent
            except (psutil.NoSuchProcess, psutil.AccessDenied,
psutil.ZombieProcess):
                pass

        if sum_all != 0:
            return cpu_percent/sum_all
        else:
            return 0.0
```

`get_manufacturer` `property`

The hardware manufacturer.

Returns:
```

| Name | Type | Description |
|---|---|---|
| `CpuType` | `CpuType` | A type of CPU. |

**Example**

```
from monitor import Monitor

monitor = Monitor({'cpu': True})

components = monitor.get_monitored_components()
print(components['cpu']['component'].get_manufacturer) #CpuType.INTEL
```

`__get_power_on_linux()`

Returns the value of the CPU power in W in Linux.

Returns:

Name	Type	Description
`float`	`float`	CPU power.

Source code in `power_pyro\cpu.py`

```python
def __get_power_on_linux(self) -> float:
    """ Returns the value of the CPU power in W in Linux.

    Returns:
        float: CPU power.
    """

    try:
        command = ["sudo", "perf", "stat", "-e", "power/energy-pkg/",
"sleep", "0.1"]
        power = subprocess.run(command, capture_output=True, text=True)

        power = power.stderr.split(" ")
        power = [string for string in power if string.strip()]

        for index, string in enumerate(power):
            if string.find('\n\n') != -1:
                power = power[index + 1]
                power = power.replace(",", ".")
                break

        power = float(power)/0.1
    except (PermissionError, subprocess.SubprocessError, AttributeError,
IndexError, ValueError) as e:
        print('Error getting power from CPU: ', str(e))

    return power
```

`__get_power_on_windows()`

Returns the value of the CPU power in W in Windows.

Returns:

Name	Type	Description
`float`	`float`	CPU power.

Source code in `power_pyro\cpu.py`

```python
def __get_power_on_windows(self) -> float:
    """ Returns the value of the CPU power in W in Windows.

    Returns:
        float: CPU power.
    """
    try:
        cpu = next((hardware for hardware in self.computer.Hardware if
hardware.HardwareType == HardwareType.Cpu), None)
        cpu.Update()
        time.sleep(0.1)

        power = next((sensor for sensor in cpu.Sensors if
sensor.SensorType == SensorType.Power and (sensor.Name == "CPU Package"
or sensor.Name == "Package")))
        power = power.Value
    except AttributeError as e:
        print('Error getting power from CPU: ', str(e))

    return power
```

`__update_hardware_name_linux()`

Set the CPU name.

Raises:

Type	Description
`HardwareNameIdentifyException`	Unable to identify CPU name in Linux.

Source code in `power_pyro\cpu.py`

```python
def __update_hardware_name_linux(self) -> None:
    """ Set the CPU name.

    Raises:
        HardwareNameIdentifyException: Unable to identify CPU name in
Linux.
    """

    try:
        self.set_name = cpuinfo.get_cpu_info()['brand_raw']
    except (ModuleNotFoundError, KeyError):
        raise HardwareNameIdentifyException(HT.CPU)
```

`__update_hardware_name_windows()`

Set the CPU name.

Raises:

Type	Description
`HardwareNameIdentifyException`	Unable to identify CPU name in Windows.

Source code in `power_pyro\cpu.py`

```python
def __update_hardware_name_windows(self) -> None:
    """ Set the CPU name.

    Raises:
        HardwareNameIdentifyException: Unable to identify CPU name in
Windows.
    """

    try:
        wmi_session = wmi.WMI()

        self.set_name = wmi_session.Win32_Processor()[0].Name
    except (ModuleNotFoundError, wmi.x_wmi, IndexError, AttributeError):
        raise HardwareNameIdentifyException(HT.CPU)
```

`__update_manufacture_linux()`

Update hardware manufacturer when running on Linux OS.

Raises:

Type	Description
`IdentifyHardwareManufacturerException`	If the hardware manufacturer

Source code in `power_pyro\cpu.py`

```python
def __update_manufacture_linux(self) -> None:
    """Update hardware manufacturer when running on Linux OS.

    Raises:
        IdentifyHardwareManufacturerException: If the hardware
manufacturer
        cannot be identified.
    """

    try:
        info = cpuinfo.get_cpu_info()

        manufacturer = info['vendor_id_raw']

        if manufacturer == 'GenuineIntel':
            self.__manufacturer = CpuType.INTEL
        elif manufacturer == 'AuthenticAMD':
            self.__manufacturer = CpuType.AMD
        else:
            raise IdentifyHardwareManufacturerException(HT.CPU)
    except (ModuleNotFoundError, KeyError):
        raise IdentifyHardwareManufacturerException(HT.CPU)
```

`__update_manufacture_windows()`

Update hardware manufacturer when running on Windows OS.

Raises:

Type	Description
`IdentifyHardwareManufacturerException`	If the hardware manufacturer

Source code in `power_pyro\cpu.py`

```python
def __update_manufacture_windows(self) -> None:
    """Update hardware manufacturer when running on Windows OS.

    Raises:
        IdentifyHardwareManufacturerException: If the hardware
manufacturer
        cannot be identified.
    """

    try:
        wmi_session = wmi.WMI()

        manufacturer = wmi_session.Win32_Processor()[0].Manufacturer

        if manufacturer == 'GenuineIntel':
            self.__manufacturer = CpuType.INTEL
        elif manufacturer == 'AuthenticAMD':
            self.__manufacturer = CpuType.AMD
        else:
            raise IdentifyHardwareManufacturerException(HT.CPU)
    except (ModuleNotFoundError, wmi.x_wmi, IndexError, AttributeError):
        raise IdentifyHardwareManufacturerException(HT.CPU)
```

`get_cpu_percent_for_process()`

Returns the percentage value of the monitored process on the CPU.

Returns:

Name	Type	Description
`float`	`float`	CPU percent.

Example

```python
from monitor import Monitor

monitor = Monitor({'cpu': True})
components = monitor.get_monitored_components()
print(components['cpu']['component'].get_cpu_percent_for_process()) # 0.37
```

Source code in `power_pyro\cpu.py`

~~~python
def get_cpu_percent_for_process(self) -> float:
    """ Returns the percentage value of the monitored process on the CPU.

    Returns:
        float: CPU percent.

    Example:
        ```python

        from monitor import Monitor

        monitor = Monitor({'cpu': True})
        components = monitor.get_monitored_components()
        print(components['cpu']
['component'].get_cpu_percent_for_process()) # 0.37
        ~~~
    """
    script_pid = os.getpid()
    sum_all = 0
    cpu_percent = 0
    for process in psutil.process_iter():
        try:
            with process.oneshot():
                process_pid = process.pid
                process_cpu_percent = process.cpu_percent()
                print(type(process))

                if process_pid:
                    sum_all += process_cpu_percent

                    if process_pid == script_pid:
                        cpu_percent += process_cpu_percent
        except (psutil.NoSuchProcess, psutil.AccessDenied,
psutil.ZombieProcess):
            pass

    if sum_all != 0:
        return cpu_percent/sum_all
    else:
        return 0.0
```

## Módulo GPU

### `Gpu`

Bases: `ProcessingUnit`

Represents a Graphics Processing Unit (GPU) responsible for accessing and retrieving power consumption valuesfrom hardware sensors.
```

Attributes:

Name	Type	Description
`__manufacturer`	`GpuType`	GPU type.

Source code in `power_pyro\gpu.py`

~~~python
class Gpu(ProcessingUnit):
    """Represents a Graphics Processing Unit (GPU) responsible
       for accessing and retrieving power consumption values
       from hardware sensors.

    Attributes:
        __manufacturer (GpuType): GPU  type.
    """

    def __init__(self, operating_system: OsType):
        super().__init__(operating_system)
        self.__manufacturer: GpuType

        if operating_system == OsType.WINDOWS:
            self.computer.IsGpuEnabled = True

        self._update_manufacture()
        self._update_hardware_name()

    @property
    def get_manufacturer(self) -> GpuType:
        """ The hardware manufacturer.

        Returns:
            GpuType: A type of GPU.

        Example:
            ```python

            from monitor import Monitor

            monitor = Monitor({'gpu': True})

            components = monitor.get_monitored_components()
            print(components['gpu']['component'].get_manufacturer)
#GpuType.NVIDIA

            ~~~
        """
        return self.__manufacturer

    def __is_there_dedicated_gpu_windows(self) -> bool:
        """ Check if it is a dedicated gpu in Windows OS.

        Returns:
            bool:
                - 'True' if a dedicated gpu is found on Windows.
                - 'False' if a dedicated gpu is not found in Windows.
        """
        computer = Computer()
        computer.Open()
        computer.IsGpuEnabled = True

        gpu = next((hardware for hardware in computer.Hardware if
(hardware.HardwareType == HardwareType.GpuIntel or

hardware.HardwareType == HardwareType.GpuAmd or
```
```

```python

hardware.HardwareType == HardwareType.GpuNvidia)), None)

        if gpu != None:
            gpu.Update()
            time.sleep(0.1)

            if next((sensor for sensor in gpu.Sensors if sensor.Name ==
"D3D Dedicated Memory Used"), None) != None:
                computer.Close()
                return True

        computer.Close()
        return False

    def _update_manufacture(self) -> None:
        if self.operating_system == OsType.WINDOWS:
            self.__update_manufacture_windows()

        elif self.operating_system == OsType.LINUX:
            self.__update_manufacture_linux()
        else:
            raise OSError("Unable to identify operating system")

    def __update_manufacture_windows(self) -> None:
        """Update hardware manufacturer when running on Windows OS.

        Raises:
            IdentifyHardwareManufacturerException: If the hardware
manufacturer
            cannot be identified.

            ResourceUnavailableException: If a dedicated GPU is not found
in Windows.
        """
        if not self.__is_there_dedicated_gpu_windows():
            raise ResourceUnavailableException("GPU", "Resource not
found!")

        computer = Computer()
        computer.Open()
        computer.IsGpuEnabled = True

        for hardware in computer.Hardware:
            hardware_type = str(hardware.HardwareType)

            if hardware_type == 'GpuNvidia':
                self.__manufacturer = GpuType.NVIDIA
            elif hardware_type == 'GpuAmd':
                self.__manufacturer = GpuType.AMD
            else:
                raise IdentifyHardwareManufacturerException(HT.GPU)

        computer.Close()

    def __update_manufacture_linux(self) -> None:
        """Update hardware manufacturer when running on Linux OS.

        Raises:
            IdentifyHardwareManufacturerException: If the hardware
manufacturer
```

```python
            cannot be identified.
        """

        if self.__is_there_nvidia_on_linux():
            self.__manufacturer = GpuType.NVIDIA
        elif self.__is_there_amd_on_linux():
            self.__manufacturer = GpuType.AMD
        else:
            raise IdentifyHardwareManufacturerException(HT.GPU)

    def _update_hardware_name(self) -> None:
        if self.operating_system == OsType.WINDOWS:
            self.__update_hardware_name_windows()
        elif self.operating_system == OsType.LINUX:
            self.__update_hardware_name_linux
        else:
            raise OSError("Unable to identify operating system")

    def __update_hardware_name_windows(self) -> None:
        """ Set the GPU name.

        Raises:
            ResourceUnavailableException: If a dedicated GPU is not found
in Windows.
        """
        if not self.__is_there_dedicated_gpu_windows():
            raise ResourceUnavailableException("GPU", "Resource not
found!")

        computer = Computer()
        computer.Open()
        computer.IsGpuEnabled = True

        for hardware in computer.Hardware:
            self.set_name = hardware.Name

        computer.Close()

    def __update_hardware_name_linux(self) -> None:
        """ Set the GPU name.

        Raises:
            HardwareNameIdentifyException: Unable to identify GPU name in
Linux.
        """
        try:
            if self.__is_there_nvidia_on_linux():
                self.set_name = subprocess.check_output("nvidia-smi --
query-gpu=name --format=csv,noheader", shell=True).decode().strip()
            elif self.__is_there_amd_on_linux():
                output = subprocess.check_output("lspci | grep -i vga",
shell=True).decode().strip()
                self.set_name = re.findall(r'\w+ \w+ \w+ \w+ / \w+
\w+\W\w+', output)
            else:
                raise HardwareNameIdentifyException(HT.GPU)
        except (FileNotFoundError, subprocess.CalledProcessError,
UnicodeDecodeError):
            raise HardwareNameIdentifyException(HT.GPU)

    def get_power(self) -> float:
        if self.operating_system == OsType.WINDOWS:
            return self.__get_power_on_windows()

        else:
            if self.__manufacturer == GpuType.NVIDIA:
                return self.__get_nvidia_power_on_linux()
            elif self.__manufacturer == GpuType.AMD:
                return self.__get_amd_power_on_linux()

    def __get_power_on_windows(self) -> float:
        """ Returns the value of the GPU power in W in Windows.

        Returns:
            float: GPU power.
        """
        gpu = next((hardware for hardware in self.computer.Hardware if
(hardware.HardwareType == HardwareType.GpuIntel or

hardware.HardwareType == HardwareType.GpuAmd or

hardware.HardwareType == HardwareType.GpuNvidia)), None)
        gpu.Update()
        time.sleep(0.1)

        power = next((sensor for sensor in gpu.Sensors if
sensor.SensorType == SensorType.Power and (sensor.Name == "GPU Power" or
sensor.Name == "GPU Package")))
        return power.Value

    def __is_there_nvidia_on_linux(self) -> bool:
        """ Check if the GPU present in linux is NVIDIA.

        Returns:
            bool:
                - 'True' if you have NVIDIA GPU on linux.
                - 'False' if you don't have NVIDIA GPU on linux.
        """
        try:
            subprocess.run(['nvidia-smi'], stdout=subprocess.PIPE,
stderr= subprocess.PIPE, check=True)
            return True
        except (FileNotFoundError, subprocess.CalledProcessError):
            return False

    def __get_nvidia_power_on_linux(self) -> float:
        """ Returns the value of the NVIDIA GPU power in W in Linux.

        Returns:
            float: GPU power.
        """
        try:
            result = subprocess.run(["nvidia-smi", "--query-
gpu=power.draw", "--format=csv,noheader,nounits"],
                                    stdout=subprocess.PIPE,
                                    stderr=subprocess.PIPE,
                                    text=True,
                                    check=True)

            return float(result.stdout.strip())
        except Exception as e:
            print('Error getting power from GPU: ', str(e))
            return 0.0

    def __is_there_amd_on_linux(self) -> bool:
        """ Check if the GPU present in Linux is AMD.

        Returns:
            bool:
                - 'True' if you have AMD GPU on Linux.
                - 'False' if you don't have AMD GPU on Linux.

        Raises:
            Exception: Error when checking AMD dedicated video card on
Linux.
        """
        try:
            result = subprocess.check_output(['lspci', '-nnk'],
universal_newlines=True)

            for line in result.splitlines():
                if re.search(r"( VGA | 3D )", line) and re.search(r"AMD",
line.upper()):
                    return True

            return False
        except Exception as e:
            raise Exception(f'Error checking for AMD graphics card:{e}')

    def __get_amd_power_on_linux(self) -> float:
        """ Returns the value of the AMD GPU power in W in Linux.

        Returns:
            float: GPU power.
        """
        try:
            hwmon_path = '/sys/class/hwmon/'

            for hwmon in os.listdir(hwmon_path):
                hwmon_dir = os.path.join(hwmon_path, hwmon)

                name_file = os.path.join(hwmon_dir, 'name')
                if os.path.exists(name_file):
                    with open(name_file, 'r') as file:
                        device = file.read().strip()

                    if device == 'amdgpu':
                        power_file = os.path.join(hwmon_dir,
'power1_input')
                        if os.path.exists(power_file):
                            with open(power_file, 'r') as file:
                                power = float(file.read().strip())

                            power /= 10**6
                            return power
        except (FileNotFoundError, PermissionError, ValueError) as e:
            print('Error getting power from GPU: ', str(e))
```

`get_manufacturer` property

The hardware manufacturer.

Returns:

Name	Type	Description
`GpuType`	`GpuType`	A type of GPU.

Example

```
from monitor import Monitor

monitor = Monitor({'gpu': True})

components = monitor.get_monitored_components()
print(components['gpu']['component'].get_manufacturer) #GpuType.NVIDIA
```

`__get_amd_power_on_linux()`

Returns the value of the AMD GPU power in W in Linux.

Returns:

Name	Type	Description
`float`	`float`	GPU power.

Source code in `power_pyro\gpu.py`

```python
def __get_amd_power_on_linux(self) -> float:
    """ Returns the value of the AMD GPU power in W in Linux.

    Returns:
        float: GPU power.
    """
    try:
        hwmon_path = '/sys/class/hwmon/'

        for hwmon in os.listdir(hwmon_path):
            hwmon_dir = os.path.join(hwmon_path, hwmon)

            name_file = os.path.join(hwmon_dir, 'name')
            if os.path.exists(name_file):
                with open(name_file, 'r') as file:
                    device = file.read().strip()

                if device == 'amdgpu':
                    power_file = os.path.join(hwmon_dir, 'power1_input')
                    if os.path.exists(power_file):
                        with open(power_file, 'r') as file:
                            power = float(file.read().strip())

                        power /= 10**6
                        return power
    except (FileNotFoundError, PermissionError, ValueError) as e:
        print('Error getting power from GPU: ', str(e))
```

`__get_nvidia_power_on_linux()`

Returns the value of the NVIDIA GPU power in W in Linux.

Returns:

Name	Type	Description
`float`	`float`	GPU power.

Source code in `power_pyro\gpu.py`

```python
def __get_nvidia_power_on_linux(self) -> float:
    """ Returns the value of the NVIDIA GPU power in W in Linux.

    Returns:
        float: GPU power.
    """
    try:
        result = subprocess.run(["nvidia-smi", "--query-gpu=power.draw",
"--format=csv,noheader,nounits"],
                                stdout=subprocess.PIPE,
                                stderr=subprocess.PIPE,
                                text=True,
                                check=True)

        return float(result.stdout.strip())
    except Exception as e:
        print('Error getting power from GPU: ', str(e))
        return 0.0
```

`__get_power_on_windows()`

Returns the value of the GPU power in W in Windows.

Returns:

Name	Type	Description
`float`	`float`	GPU power.

Source code in `power_pyro\gpu.py`

```python
def __get_power_on_windows(self) -> float:
    """ Returns the value of the GPU power in W in Windows.

    Returns:
        float: GPU power.
    """
    gpu = next((hardware for hardware in self.computer.Hardware if
(hardware.HardwareType == HardwareType.GpuIntel or

hardware.HardwareType == HardwareType.GpuAmd or

hardware.HardwareType == HardwareType.GpuNvidia)), None)
    gpu.Update()
    time.sleep(0.1)

    power = next((sensor for sensor in gpu.Sensors if sensor.SensorType
== SensorType.Power and (sensor.Name == "GPU Power" or sensor.Name ==
"GPU Package")))
    return power.Value
```

`__is_there_amd_on_linux()`

Check if the GPU present in Linux is AMD.

Returns:

Name	Type	Description
`bool`	`bool`	• 'True' if you have AMD GPU on Linux. • 'False' if you don't have AMD GPU on Linux.

Raises:

Type	Description
`Exception`	Error when checking AMD dedicated video card on Linux.

Source code in `power_pyro\gpu.py`

```python
def __is_there_amd_on_linux(self) -> bool:
    """ Check if the GPU present in Linux is AMD.

    Returns:
        bool:
            - 'True' if you have AMD GPU on Linux.
            - 'False' if you don't have AMD GPU on Linux.

    Raises:
        Exception: Error when checking AMD dedicated video card on Linux.
    """
    try:
        result = subprocess.check_output(['lspci', '-nnk'],
universal_newlines=True)

        for line in result.splitlines():
            if re.search(r"( VGA | 3D )", line) and re.search(r"AMD",
line.upper()):
                return True

        return False
    except Exception as e:
        raise Exception(f'Error checking for AMD graphics card:{e}')
```

`__is_there_dedicated_gpu_windows()`

Check if it is a dedicated gpu in Windows OS.

Returns:

Name	Type	Description
`bool`	`bool`	• 'True' if a dedicated gpu is found on Windows. • 'False' if a dedicated gpu is not found in Windows.

Source code in `power_pyro\gpu.py`

```python
def __is_there_dedicated_gpu_windows(self) -> bool:
    """ Check if it is a dedicated gpu in Windows OS.

    Returns:
        bool:
            - 'True' if a dedicated gpu is found on Windows.
            - 'False' if a dedicated gpu is not found in Windows.
    """
    computer = Computer()
    computer.Open()
    computer.IsGpuEnabled = True

    gpu = next((hardware for hardware in computer.Hardware if
(hardware.HardwareType == HardwareType.GpuIntel or

hardware.HardwareType == HardwareType.GpuAmd or

hardware.HardwareType == HardwareType.GpuNvidia)), None)

    if gpu != None:
        gpu.Update()
        time.sleep(0.1)

        if next((sensor for sensor in gpu.Sensors if sensor.Name == "D3D
Dedicated Memory Used"), None) != None:
            computer.Close()
            return True

    computer.Close()
    return False
```

`__is_there_nvidia_on_linux()`

Check if the GPU present in linux is NVIDIA.

Returns:

Name	Type	Description
`bool`	`bool`	<ul><li>'True' if you have NVIDIA GPU on linux.</li><li>'False' if you don't have NVIDIA GPU on linux.</li></ul>

Source code in `power_pyro\gpu.py`

```python
def __is_there_nvidia_on_linux(self) -> bool:
    """ Check if the GPU present in linux is NVIDIA.

    Returns:
        bool:
            - 'True' if you have NVIDIA GPU on linux.
            - 'False' if you don't have NVIDIA GPU on linux.
    """
    try:
        subprocess.run(['nvidia-smi'], stdout=subprocess.PIPE, stderr=
subprocess.PIPE, check=True)
        return True
    except (FileNotFoundError, subprocess.CalledProcessError):
        return False
```

`__update_hardware_name_linux()`

Set the GPU name.

Raises:

Type	Description
`HardwareNameIdentifyException`	Unable to identify GPU name in Linux.

Source code in `power_pyro\gpu.py`

```python
def __update_hardware_name_linux(self) -> None:
    """ Set the GPU name.

    Raises:
        HardwareNameIdentifyException: Unable to identify GPU name in
Linux.
    """
    try:
        if self.__is_there_nvidia_on_linux():
            self.set_name = subprocess.check_output("nvidia-smi --query-
gpu=name --format=csv,noheader", shell=True).decode().strip()
        elif self.__is_there_amd_on_linux():
            output = subprocess.check_output("lspci | grep -i vga",
shell=True).decode().strip()
            self.set_name = re.findall(r'\w+ \w+ \w+ \w+ / \w+ \w+\W\w+',
output)
        else:
            raise HardwareNameIdentifyException(HT.GPU)
    except (FileNotFoundError, subprocess.CalledProcessError,
UnicodeDecodeError):
        raise HardwareNameIdentifyException(HT.GPU)
```

`__update_hardware_name_windows()`

Set the GPU name.

Raises:

Type	Description
`ResourceUnavailableException`	If a dedicated GPU is not found in Windows.

Source code in `power_pyro\gpu.py`

```python
def __update_hardware_name_windows(self) -> None:
    """ Set the GPU name.

    Raises:
        ResourceUnavailableException: If a dedicated GPU is not found in
Windows.
    """
    if not self.__is_there_dedicated_gpu_windows():
        raise ResourceUnavailableException("GPU", "Resource not found!")

    computer = Computer()
    computer.Open()
    computer.IsGpuEnabled = True

    for hardware in computer.Hardware:
        self.set_name = hardware.Name

    computer.Close()
```

`__update_manufacture_linux()`

Update hardware manufacturer when running on Linux OS.

Raises:

Type	Description
`IdentifyHardwareManufacturerException`	If the hardware manufacturer

Source code in `power_pyro\gpu.py`

```python
def __update_manufacture_linux(self) -> None:
    """Update hardware manufacturer when running on Linux OS.

    Raises:
        IdentifyHardwareManufacturerException: If the hardware
manufacturer
        cannot be identified.
    """

    if self.__is_there_nvidia_on_linux():
        self.__manufacturer = GpuType.NVIDIA
    elif self.__is_there_amd_on_linux():
        self.__manufacturer = GpuType.AMD
    else:
        raise IdentifyHardwareManufacturerException(HT.GPU)
```

`__update_manufacture_windows()`

Update hardware manufacturer when running on Windows OS.

Raises:

Type	Description
`IdentifyHardwareManufacturerException`	If the hardware manufacturer
`ResourceUnavailableException`	If a dedicated GPU is not found in Windows.

Source code in `power_pyro\gpu.py`

```python
def __update_manufacture_windows(self) -> None:
    """Update hardware manufacturer when running on Windows OS.

    Raises:
        IdentifyHardwareManufacturerException: If the hardware
manufacturer
        cannot be identified.

        ResourceUnavailableException: If a dedicated GPU is not found in
Windows.
    """
    if not self.__is_there_dedicated_gpu_windows():
        raise ResourceUnavailableException("GPU", "Resource not found!")

    computer = Computer()
    computer.Open()
    computer.IsGpuEnabled = True

    for hardware in computer.Hardware:
        hardware_type = str(hardware.HardwareType)

        if hardware_type == 'GpuNvidia':
            self.__manufacturer = GpuType.NVIDIA
        elif hardware_type == 'GpuAmd':
            self.__manufacturer = GpuType.AMD
        else:
            raise IdentifyHardwareManufacturerException(HT.GPU)

    computer.Close()
```

Módulo Memória

`Memory`

Bases: `HardwareComponent`

Represents a Memory component responsible for calculating power consumption based on the amount of memory used.

Attributes:

Name	Type	Description
`__WATT_PER_GB`	`float`	Power consumption per GB of memory.

Source code in `power_pyro\memory.py`

```python
class Memory(HardwareComponent):
    """Represents a Memory component responsible for calculating power
consumption
       based on the amount of memory used.

    Attributes:
        __WATT_PER_GB (float): Power consumption per GB of memory.
    """
    def __init__(self, operating_system: OsType):
        super().__init__(operating_system)
        self.__WATT_PER_GB: float = self.__watt_per_gb()

    def __watt_per_gb(self) -> float:
        """Calculates the power consumption per GB of memory.

        Returns:
            float: Power consumption per GB of memory.

        Raises:
            OSError: If the operating system is not recognized.
        """
        if self.operating_system == OsType.WINDOWS:
            return self.__watt_per_gb_on_windows()
        elif self.operating_system == OsType.LINUX:
            return self.__watt_per_gb_on_linux()
        else:
            raise OSError("Unable to identify operating system")

    def __watt_per_gb_on_windows(self) -> float:
        """Calculates the power consumption per GB of memory on Windows
OS.

        Returns:
            float: Power consumption per GB.

        Raises:
            RuntimeError: Unable to get information from memory.
        """
        try:
            BYTES_TO_GIGABYTES = 1024**3
            wmi_session = wmi.WMI()

            num_memory_modules = len(wmi_session.Win32_PhysicalMemory())

            memory_module = wmi_session.Win32_PhysicalMemory()[0]
            gb_per_module =
int(memory_module.Capacity)/BYTES_TO_GIGABYTES
        except (ModuleNotFoundError, IndexError, TypeError, wmi.x_wmi):
            raise RuntimeError("Unable to get watts per GB information
from memory")

        return (5 * num_memory_modules)/gb_per_module

    def __watt_per_gb_on_linux(self) -> float:
        """Calculates the power consumption per GB of memory on Linux OS.

        Returns:
```

~~~python
            float: Power consumption per GB.

        Raises:
            RuntimeError: Unable to get information from memory.
        """
        try:
            output = subprocess.check_output(["sudo", "dmidecode", "-t", "memory"], universal_newlines=True)

            num_memory_modules_found = len(re.findall(r"\tSize:", output))
            number_unused_memory_modules = len(re.findall(r"Size: No Module Installed", output))

            num_memory_modules = num_memory_modules_found - number_unused_memory_modules

            gb_per_module = re.findall(r"\tSize: \d+ \w+", output)
            gb_per_module = gb_per_module[0].split(": ")
            gb_per_module = gb_per_module[1].split(" ")
            gb_per_module = int(gb_per_module[0])
        except (FileNotFoundError, PermissionError, subprocess.CalledProcessError, IndexError, ValueError):
            raise RuntimeError("Unable to get watts per GB information from memory")

        return (5 * num_memory_modules)/gb_per_module

    def get_power(self) -> float:
        """Returns the power consumption of the memory in W.

        Returns:
            float: Memory power consumption.

        Example:
            Monitor and print memory power consumption:

            ```python
            from power_pyro import Monitor

            monitor = Monitor({'memory': True})  # Enable memory monitoring
            power = monitor.memory.get_power()  # Access memory subsystem
            print(f"Current memory power: {power:.2f} W") # 15.2 W
            ~~~

        """
        try:
            pid = os.getpid()
            process = psutil.Process(pid)

            power = process.memory_info().rss
            power /= (1024 ** 3)
            power *= self.__WATT_PER_GB
        except (psutil.NoSuchProcess, psutil.AccessDenied, psutil.ZombieProcess) as e:
            print('Error getting power from memory:', str(e))

        return power
```
```

`__watt_per_gb()`

Calculates the power consumption per GB of memory.

Returns:

Name	Type	Description
`float`	`float`	Power consumption per GB of memory.

Raises:

Type	Description
`OSError`	If the operating system is not recognized.

Source code in `power_pyro\memory.py`

```python
def __watt_per_gb(self) -> float:
    """Calculates the power consumption per GB of memory.

    Returns:
        float: Power consumption per GB of memory.

    Raises:
        OSError: If the operating system is not recognized.
    """
    if self.operating_system == OsType.WINDOWS:
        return self.__watt_per_gb_on_windows()
    elif self.operating_system == OsType.LINUX:
        return self.__watt_per_gb_on_linux()
    else:
        raise OSError("Unable to identify operating system")
```

`__watt_per_gb_on_linux()`

Calculates the power consumption per GB of memory on Linux OS.

Returns:

Name	Type	Description
`float`	`float`	Power consumption per GB.

Raises:

Type	Description
`RuntimeError`	Unable to get information from memory.

Source code in `power_pyro\memory.py`

```python
def __watt_per_gb_on_linux(self) -> float:
    """Calculates the power consumption per GB of memory on Linux OS.

    Returns:
        float: Power consumption per GB.

    Raises:
        RuntimeError: Unable to get information from memory.
    """
    try:
        output = subprocess.check_output(["sudo", "dmidecode", "-t",
"memory"], universal_newlines=True)

        num_memory_modules_found = len(re.findall(r"\tSize:", output))
        number_unused_memory_modules = len(re.findall(r"Size: No Module
Installed", output))

        num_memory_modules = num_memory_modules_found -
number_unused_memory_modules

        gb_per_module = re.findall(r"\tSize: \d+ \w+", output)
        gb_per_module = gb_per_module[0].split(": ")
        gb_per_module = gb_per_module[1].split(" ")
        gb_per_module = int(gb_per_module[0])
    except (FileNotFoundError, PermissionError,
subprocess.CalledProcessError, IndexError, ValueError):
        raise RuntimeError("Unable to get watts per GB information from
memory")

    return (5 * num_memory_modules)/gb_per_module
```

`__watt_per_gb_on_windows()`

Calculates the power consumption per GB of memory on Windows OS.

Returns:

Name	Type	Description
`float`	`float`	Power consumption per GB.

Raises:

Type	Description
`RuntimeError`	Unable to get information from memory.

Source code in `power_pyro\memory.py`

```python
def __watt_per_gb_on_windows(self) -> float:
    """Calculates the power consumption per GB of memory on Windows OS.

    Returns:
        float: Power consumption per GB.

    Raises:
        RuntimeError: Unable to get information from memory.
    """
    try:
        BYTES_TO_GIGABYTES = 1024**3
        wmi_session = wmi.WMI()

        num_memory_modules = len(wmi_session.Win32_PhysicalMemory())

        memory_module = wmi_session.Win32_PhysicalMemory()[0]
        gb_per_module = int(memory_module.Capacity)/BYTES_TO_GIGABYTES
    except (ModuleNotFoundError, IndexError, TypeError, wmi.x_wmi):
        raise RuntimeError("Unable to get watts per GB information from
memory")

    return (5 * num_memory_modules)/gb_per_module
```

`get_power()`

Returns the power consumption of the memory in W.

Returns:

Name	Type	Description
`float`	`float`	Memory power consumption.

Example

Monitor and print memory power consumption:

```
from power_pyro import Monitor

monitor = Monitor({'memory': True})  # Enable memory monitoring
power = monitor.memory.get_power()  # Access memory subsystem
print(f"Current memory power: {power:.2f} W") # 15.2 W
```

Source code in `power_pyro\memory.py`

~~~
def get_power(self) -> float:
    """Returns the power consumption of the memory in W.

    Returns:
        float: Memory power consumption.

    Example:
        Monitor and print memory power consumption:

        ```python
        from power_pyro import Monitor

        monitor = Monitor({'memory': True})  # Enable memory monitoring
        power = monitor.memory.get_power()  # Access memory subsystem
        print(f"Current memory power: {power:.2f} W") # 15.2 W
        ```

    """
    try:
        pid = os.getpid()
        process = psutil.Process(pid)

        power = process.memory_info().rss
        power /= (1024 ** 3)
        power *= self.__WATT_PER_GB
    except (psutil.NoSuchProcess, psutil.AccessDenied,
psutil.ZombieProcess) as e:
        print('Error getting power from memory:', str(e))

    return power
~~~