cpu_health_checks

Release 1.0.0

Felipe Santana Rojas

Jul 04, 2023

CONTENTS:

CHAPTER

ONE

INTRODUCTION

ADDING TEST TEXT

1.1 Overview

The CPU Health Checks package is designed to perform various CPU health checks on any platform (Linux, MacOS, Windows). It provides a set of functionalities to monitor pending reboots, disk space availability, idle CPU usage, network connection status, download speed, latency, and battery charge. The package consists of three modules: *cpu_health.py*, *utilities.py*, and *test_checks.py*.

1.2 Goals

The main goals of the CPU Health Checks package are:

- Perform comprehensive CPU health checks across different platforms.
- Provide an easy-to-use interface to monitor critical aspects of CPU health.
- Allow flexibility by enabling users to override default configuration parameters.
- Facilitate automation by providing a main function that performs all the health checks.
- Storing results on log files to check for trends in time and possible performance outliers.

1.3 Results

Each check function returns True if the check passed (for example if there is enough disk space), and False otherwise. Calling the main() wrapper function in cpu_health.py returns a dictionary Where the keys are the name of the checks performed and the values are the result of the corresponding test.

All the general information and timestamps are written into a general log file, and another log file dedicated only to monitor the internet download speed.

Finally the functions print colored messages on screen to clearly highlight when the checks passed, when they didn't (providing extra information on why the didn't), and when the usage, file content, or input parameters raised an exception that prevented the check to be performed.

CHAPTER

TWO

INSTALLING

To install the CPU Health Checks package you need to Install from Source using Git Clone following these steps:

1. Clone the repository using the following command:

   ```
   git clone https://github.com/fsantanar/cpu_health_checks.git
   ```

   This will create a local copy of the repository on your machine.

   Please note that before this, you will need to have git installed on your computer. If you don't have it, please follow the git installation instructions. If you are running a Mac computer, you can install the Xcode Command Line Tools, which include git, by typing "xcode-select –install" in your terminal.

2. Navigate to the repository directory:

   ```
   cd cpu_health_checks
   ```

3. Install the package:

   ```
   pip install .
   ```

   Note: If you want to install in developer mode and modify the modules locally without needing to pip install each time add the "-e" option after "pip install".

4. Go to the folder with the python modules and test the package running the main() function in the cpu_health module using the "auto" option.

   ```
   cd src/cpu_health_checks
   python cpu_health.py auto
   ```

   Which should perform the CPU health checks on your computer.

CHAPTER

THREE

CONFIGURATION FILE

The CPU Health Checks package uses a YAML configuration file to specify various input parameters. By default, the configuration file is searched at *config/configuration.yml* and the *default* main Key in it is used.

You can change the default config file and mode when calling the main() wrapper function or the CPUCheck constructor in the cpu_health.py module by specifying the configuration file relative path with the *config_file* parameter, and the main key to use with the *config_mode* parameter.

For example if you want to run all the tests based on the content of the "my_laptop" main key, from the configuration file "myconfig.yml" at folder "configs/mac/" you can type: main(config_file='configs/mac/myconfig.yml', config_mode='my_laptop').

Then, by default the parameters from the configuration file/mode set will be used as input, but the user can override any parameter defined in the configuration by specifying it when instantiating the *CPUCheck* object or when using the main() wrapper.

3.1 Example Configuration File

The following is an example of a configuration file (*configuration.yml*) used by the CPU Health Checks package, and is the one provided as starting point by the package:

```
'default':
  # General
  logs_folder: 'logs/'

  # check_enough_disk_space
  min_gb: 2
  min_percent_disk: 10
  folders_to_print: 3

  # check_enough_idle_usage
  max_cpu_usage: 75

  # check_network_available
  website_to_check: 'www.google.com'

  # check_good_download_speed
  max_connection_attempts: 5
  file_sizes_to_download: ['1MB', '10MB', '100MB', '1GB', '10GB']
  block_size: 8192
  sleep_time: 1
```

(continues on next page)

(continued from previous page)

```
  speed_log_filename: 'download_speed_register.txt'
  minimum_previous_tests: 3
  std_deviations_limit: 2
  speed_min_mbps: 1
  minimum_download_time: 3

  # check_fast_latency
  latency_url: 'www.google.com'
  latency_limit_ms: 100

  # check_enough_battery_charge
  min_percent_battery: 10
  min_remaining_time_mins: 15
```

You can modify any of these parameters according to your requirements. The commented line on top of each group of parameters indicates the function in which the parameters are used.

You can also define multiple main keys in the configuration file (other than *default*) and quickly switch between which one is used by setting the *config_mode* parameter when calling the CPUCheck() constructor or the main() wrapper function.

CHAPTER

FOUR

DESCRIPTION

The CPU Health Checks package provides a comprehensive set of CPU health check functionalities. It consists of three modules:

1. *cpu_health.py*: This module contains the core functionality of the package. It includes the *CPUCheck* class, which is responsible for performing the CPU health checks. It also provides a *main* function that serves as a wrapper to create a *CPUCheck* object and execute all the health checks. The *cpu_health* module relies on the *utilities* module for supporting functions.
2. *utilities.py*: This module contains supporting functions that are used by the *cpu_health* module. These functions handle tasks such as reading the configuration file, calculating disk space, checking network connectivity, measuring download speed, and more.
3. *test_checks.py*: This module contains unit tests for the *cpu_health* module. It includes various test cases to ensure the correctness of the CPU health checks.

4.1 Preparation

First make sure that the configuration file config/configuration.yml (with respect to package root folder) main key (default value is "default") has the input parameter values you want to use. Then check that the "config_file" default parameter value corresponds to the location of the configuration file with respect to the folder where you are running the modules. If not You can change this value when calling CPUCheck() or main().

Finally make sure that the logs folder exists and its locations with respect to the folder where you are running the modules is properly defined in the "logs_folder" parameter, either using the value in the configuration file or defining it explicitly when calling CPUCheck() or main().

If you are running the modules from folder src/cpu_health_checks (their default location), the configuration file and the logs folder will correspond to the default values in the "config_file", and "logs_folder" parameters, so you don't need to worry about those unless you want to change them.

After installing the package you would be able to import the modules using statements like *import cpu_health_checks.cpu_health as cpu_health*, or *import cpu_health_checks.utilities as utilities*. You can also run the Python modules directly by doing "python path/to/cpu_health_module_folder/cpu_health.py" replacing "path/to/cpu_health_module_folder/" with the actual folder containing the cpu_health.py and utilities.py modules.

4.2 General Flow/Usage

The main ways to run the CPU health checks are the following:

1. Running individual checks: You can create an instance of the CPUCheck class and call specific methods to perform individual CPU health checks. For example:

```
import cpu_health_checks.cpu_health as cpu_health

# Create a CPUCheck object
checkobj = cpu_health.CPUCheck()

# Perform specific health checks
res1 = checkobj.check_enough_disk_space()
res2 = checkobj.check_enough_idle_usage()
res3 = checkobj.check_network_connection()
# ... add more checks as needed
# Retrieve the results and take actions based on the returned values
```

   Each check function returns True if the check passed and False otherwise.

   Remember that you can also do this by running the cpu_health.py module from the terminal using python interactive mode and then in the python interpreter create the object doing for example checkobj = CPUCheck().

2. Running all checks at once: you can run all the CPU health checks at once using the main() wrapper function. There are two ways to do this:

   a. Execute the main() function of cpu_health.py using the default parameters defined in the .yml configuration file, and the default values for "config_file", and "config_mode" defined in the CPUCheck() constructor. For example:

      In Python, first import the cpu_health module:

```
import cpu_health_checks.cpu_health as cpu_health
```

      And then run the main() function with the default input parameters without explicitly defining any parameter at call time:

```
main()
```

      You can also do this in a single step without needing to use python interactively by just going to the folder containing the Python modules and running the cpu_health.py module using the "auto" option when running the module:

```
python cpu_health.py auto
```

   b. Execute the main() function of cpu_health.py using input parameters to override the configuration file. For example:

      In Python, first import the cpu_health module:

```
import cpu_health_checks.cpu_health as cpu_health
```

      And then call the main function overriding the desired parameters. Remember to make sure that default values for "config_file" and "logs_folder" are appropriate given the folder you are running the module, or otherwise define them explicitly as input parameters of the main() function at call time. For example:

```
result = cpu_health.main(logs_folder='logs/linux/', latency_url='www.example.com
↪')
```

When doing this, change "logs_folder" and "latency_url" for the actual input parameters you want to override form the configuration file. Here "result" will then be a dictionary with the keys having the names of the checks performed and the values will be the results of each test.

You can do the same running cpu_health.py in python interactive mode and then typing "main(logs_folder='logs/linux/', latency_url='www.example.com')" in the python interpreter.

Both methods generate logs in the specified logs folder, providing detailed information about the health check results.

4.3 Modules Role

Each component of the CPU Health Checks package has a specific role:

- *CPUCheck*: This class encapsulates the CPU health checks and provides an interface to configure and execute the checks. It utilizes the supporting functions in the *utilities* module.
- *utilities*: This module contains functions that handle various tasks related to CPU health checks. These functions are called by the *CPUCheck* class to perform specific checks or retrieve information.
- *test_checks*: This module tests the behavior of the product. Some checks will return False, indicating that the check didn't pass, which doesn't necessarily mean there is an error in the code; it is just the expected output of the check. However, *test_checks* checks if the behavior of the product is correct by running multiple unittests where the expected result is known a priori given the input parameters. For example, we test if *CPUCheckObj.check_enough_disk_space()* returns False when the *min_percent_disk* attribute of the *CPUCheckObj* object is 100, because we assume that at least some part of the disk is used. Some of these tests should return True, some should return False, and some should raise exceptions. If everything goes as expected, the general test prints "OK" at the end; otherwise, it indicates which tests failed.

CHAPTER

FIVE

USAGE EXAMPLES

This section provides examples on how to use the CPU Health Checks package effectively.

5.1 Example 1: Run the *main()* Wrapper Function Automatically from Command line

To run the CPU health checks using the *main()* function in the *cpu_health.py* module, you can execute the following command:

```
python cpu_health.py auto
```

This will use the configuration parameters defined in the configuration file (*config/configuration.yml*) to create a *CPUCheck* object and perform all the health checks.

5.2 Example 2: Using the *main()* Wrapper Function using non-default parameters

You can run the CPU health checks using the *main()* function using a configuration file and mode different than the default defined in the CPUCheck constructor (*default* key in *config/configuration.yml* file). And for any parameter you can also use values different than the ones defined in your custom configuration file.

For example if you have your custom configuration file called *custom.yml* in folder *inputs/* with respect to the folder where you are running the code you need to use config_file=`inputs/custom.yml`. Now if you want to use the values within the main key *laptop_check* in that file you need to use the config_mode=`laptop_check` value. Finally if there are some values there that you want to override when running a specific check like min_gb=20, and min_percent_battery=50 you can define those at at call time, as it is shown in the example.

```
python -i cpu_health.py
```

```
main(config_file='inputs/custom.yml', config_mode='laptop_check', min_gb=20, min_percent_
↪battery=50)
```

This will use the configuration parameters defined in the configuration file (*config/configuration.yml*) to create a *CPUCheck* object and perform all the health checks.

5.3 Example 3: Using the *CPUCheck()* Constructor

To run individual CPU health checks with specific input parameters, you can create an instance of the *CPUCheck* class and call the relevant methods. Here's an example:

```
import cpu_health_checks.cpu_health as cpu_health

config_file_path = "path/to/configuration.yml"
logs_folder_path = "path/to/logs"

# Create a CPUCheck object called checkobj
checkobj = cpu_health.CPUCheck(config_file=config_file_path, logs_folder=logs_folder_
↪path)

# Perform specific health checks
checkobj.check_enough_disk_space()
checkobj.check_enough_idle_usage()
checkobj .check_network_connection()
# ... add more checks as needed

# Retrieve the results and take actions based on the returned values
```

There you need to replace "path/to/configuration.yml" for the path and filename of the configuration file, and "path/to/logs" with the folder to use to store the general and download speed logs. Both folders have to be specified with respect to the folder where cpu_health is being run.

5.4 Example 4: Using the *test_checks.py* Module

The *test_checks.py* module provides unit tests for the *cpu_health.py* module. You can use it to verify that the code behaves as expected. Here's an example of using the *test_checks.py* module:

```
python -m unittest test_checks.py
```

This command will execute the unit tests defined in the *test_checks.py* module and report the test results.

CHAPTER

SIX

PACKAGE CONTENT

6.1 cpu_health Main Module

class cpu_health_checks.cpu_health.**CPUCheck**(*config_file='../../config/configuration.yml'*, *config_mode='default'*, *logs_folder=None*, *min_gb=None*, *min_percent_disk=None*, *folders_to_print=None*, *max_cpu_usage=None*, *website_to_check=None*, *max_connection_attempts=None*, *file_sizes_to_download=None*, *block_size=None*, *sleep_time=None*, *speed_log_filename=None*, *minimum_previous_tests=None*, *std_deviations_limit=None*, *speed_min_mbps=None*, *minimum_download_time=None*, *latency_url=None*, *latency_limit_ms=None*, *min_percent_battery=None*, *min_remaining_time_mins=None*)

Bases: `object`

A class for performing CPU-related checks and tests.

Methods:

check_no_pending_reboot(): Returns boolean indicating if the PC has no pending reboots.

check_enough_disk_space(): Returns boolean indicating if there is enough disk space.

check_enough_idle_usage(): Returns boolean indicating if the CPU has enough idle usage.

check_network_available(): Returns boolean indicating if network is available.

check_good_download_speed(): Returns boolean indicating if the download speed is above a threshold and is not a low outlier.

check_fast_latency(): Returns boolean indicating if latency is fast.

check_enough_battery_charge(): Returns boolean indicating if there is enough battery charge left.

CPUCheck object __init__ constructor:

This method creates the CPUCheck object using multiple input parameters that can be defined explicitly when calling the constructor or if not taken from the 'config_mode' key of the 'config_file' configuration file, but every parameter from the method signature has to be defined in at least one of the two.

The method checks that all the input parameters have the proper type and if it corresponds it also check if the value is within the corresponding min and max limits. It also starts a logger to store the results of the check methods.

Args:

config_file (str): Path to the configuration file. Default: 'config/configuration.yml'.

config_mode (str): Configuration mode to use. Default: 'default'.

logs_folder (str): Path to the folder where logs are stored. The general log filename is based on major system properties to facilitate comparison of results across platforms/computers.

min_gb (float): Minimum required free disk space in GB.

min_percent_disk (float): Minimum required free disk space as a percentage.

folders_to_print (int): Number of largest subfolders to print.

max_cpu_usage (float): Maximum allowed CPU usage percentage.

website_to_check (str): Website URL to check network connectivity.

max_connection_attempts (int): Number of times to attempt connection before giving up.

file_sizes_to_download (list): List of file sizes to download for testing. The full list of sizes from which you can choose is: 1MB, 10MB, 100MB, 1GB, 10GB, 50GB, 100GB, and 1000GB. Although very large files are not recommended due to large download times.

block_size (int): Block size for downloading files.

sleep_time (float): Sleep time between download requests used to avoid overloading the server.

speed_log_filename (str): Name of the download speed log file.

minimum_previous_tests (int): Minimum number of previous download tests to perform comparison between current and previous values obtained.

std_deviations_limit (float): Standard deviations limit for comparing download speeds.

check_good_download_speed will not pass if the current download speed is less than the average speed minus 'std_deviations_limit' times the standard deviation of the speed.

speed_min_mbps (float): Minimum required download speed in Mbps.

minimum_download_time (float): Minimum required download time in seconds. In check_good_download_speed, files are downloaded from smaller to larger to ensure that a file is large enough for accurate speed measurement but not too large to take too long. If the download time of a file is more than 'minimum_download_time', the final file used to measure the download speed will be the next file in terms of size, which is usually 10 times bigger.

latency_url (str): URL to be used for latency check.

latency_limit_ms (float): High limit in milliseconds for the latency check to pass.

min_percent_battery (float): Minimum battery charge as a percentage.

min_remaining_time_mins (float): Minimum remaining battery charge in minutes.

Example configuration file ('config/configuration.yml'):

```
'default':
  # General
  logs_folder: 'logs/'

  # check_enough_disk_space
  min_gb: 2
  min_percent_disk: 10
  folders_to_print: 3

  # check_enough_idle_usage
  max_cpu_usage: 75
```

(continues on next page)

(continued from previous page)

```
# check_network_available
website_to_check: 'www.google.com'

# check_good_download_speed
max_connection_attempts: 5
file_sizes_to_download: ['1MB', '10MB', '100MB', '1GB', '10GB']
block_size: 8192
sleep_time: 1
speed_log_filename: 'download_speed_register.txt'
minimum_previous_tests: 3
std_deviations_limit: 2
speed_min_mbps: 1
minimum_download_time: 3

# check_fast_latency
latency_url: 'www.google.com'
latency_limit_ms: 100

# check_enough_battery_charge
min_percent_battery: 10
min_remaining_time_mins: 15
```

check_enough_battery_charge()

Checks battery level, plugging, and time remaining.

Returns True if the battery has enough charge and the remaining time is not too low. If not, returns False and checks if the battery is plugged, if it is it recommends to check battery health, if it is not, it recommends to plug it.

check_enough_disk_space()

Checks if there is enough disk space available.

Checks the disk space available, and if the disk space is above the minimum limit and the fraction of free space is above the minimum required then it returns True

If there is no enough space it gives you a hint on how to free space indicating the largest home subfolders.

check_enough_idle_usage()

Returns True if the CPU has enough idle usage.

check_fast_latency()

Checks if the latency is fast measuring the average value to the given host.

It also prints a quality flag associated to the latency value according to genelrally acceted benchmarks

check_good_download_speed()

Perform download speed tests and return the result.

The download speed test is performed by downloading files of different sizes from a predefined URL. The download speed is calculated and compared against specified thresholds to determine if the test passes or fails. If the file is sucessfuly downloaded, the download speed is above the minimum limit, and the speed is not a low outlier compared to previous results, the method returns True, if not it results False.

Returns:

bool: True if the download speed test suceeds, False otherwise.

Raises:

AssertionError: If the logs folder is not a directory.

check_network_available()

Return True if it suceeds to resolve the given URL, and False otherwise.

check_no_pending_reboot()

Returns True if the computer has no pending reboots and False if it has

cpu_health_checks.cpu_health.**main**(***kwargs*)

The main function to execute the cpu checks based on the provided configuration.

It starts by instantiating a CPUCheck object which by default (no kwargs provided at call time) is done taking all the input parameters from the 'default' main key of the configuration file config/configuration.yml. Then, if any kwarg is provided when calling the function, that parameter overrides the value in the configuration file.

Then it runs a series of cpu health checks, which return True if the test pass and False otherwise. Finally it prints and logs the results indicating how many checks passed/failed.

The list of checks to run is: [check_no_pending_reboot, check_enough_disk_space, check_enough_idle_usage, check_network_available, check_good_download_speed, and check_enough_battery_charge]. But if check_network_available fails (returns False), check_good_download_speed and check_fast_latency are not run and it is set automatically to failed. Also the test check_enough_battery_charge is automatically skipped if there is no battery information available (which probably means the code is beign run on a desktop).

Args:

kwargs: Series of optional kwargs to be used for instantiating the CPUCheck object.
These parameters have to be part of the CPUCheck.__init__ signature (see help CPUCheck.__init__ for more information), and they override the value present in the configuration file.

Returns:

dict: Dictionary whose keys are the name of the checks performed and the values
correspond to the result of each test

Raises:

TypeError: If any kwargs are not part of the parameters used
to instantiate the CPUCheck object.

Examples:

Example 1: Running main without any kwargs

```
>>> results = main()
>>> result
{'check_no_pending_reboot': True, 'check_enough_disk_space': True, ...}
```

Example 2: Running main with specific (and extreme) kwargs

```
>>> results = main(min_percent_disk=100, max_cpu_usage=100)
>>> result
{..., 'check_enough_disk_space': False, 'check_enough_idle_usage': True, ...}
```

6.2 utilities Complementary Module

cpu_health_checks.utilities.**check_arguments_validity**(*arguments*)

Check the validity of input arguments based on their types and specified boundaries.

Args:
arguments (dict): A dictionary containing the input arguments and their values.

Raises:
TypeError: If an argument's value does not match the allowed types. ValueError: If an argument's value is outside the specified minimum or maximum bounds.

Returns:
None

cpu_health_checks.utilities.**determines_log_filename**()

Determines the log filename based on major system properties.

This way we can facilitate the comparison between the performance of the different test across different machines.

cpu_health_checks.utilities.**downloads_file**(*url*, *block_size*, *max_attempts*, *logger*, *track_progress*)

Performs a null download of the file in the url.

File is downloaded by splitting it into blocks, trying (max_attempts) of times to establish connection. If it doesn't work it returns False and logs an error. If it succeeds returns True If track_progress is True then it displays a progress bar while downloading.

cpu_health_checks.utilities.**get_configured_logger**(*log_object_name*, *log_file_name*)

Configures a logger with RotatingFileHandler and retrieves it.

cpu_health_checks.utilities.**get_folder_size**(*folder*)

Get the size of a folder in bytes.

Args:
folder (str): Path to the folder.

Returns:
int: Size of the folder in bytes.

cpu_health_checks.utilities.**get_home_subfolder_info**()

Get information about the home subfolders.

Returns:
dict: A dict containing the home subfolders names and sizes.

cpu_health_checks.utilities.**get_input_params**(*function_params*, *function_values*, *config_file*, *config_mode*)

Gets the input parameters to be used for a given function.

It used the parameters explicitly defined when calling the function and if not it uses the ones present in a configuration file, but all the parameters of the function signature have to be present in one of those two.

cpu_health_checks.utilities.**get_largest_subfolders**(*folders_to_print*, *subfolders_sizes*)

Get the largest subfolders based on their sizes.

Args:
folders_to_print (int): The number of folders to print. subfolders_sizes (dict): A dictionary containing subfolder sizes.

Returns:
list: A list of tuples containing the largest subfolders and their sizes.

cpu_health_checks.utilities.**get_megas**(*size*)

Convert a string with bytes info into the number of corresponding mega bytes

cpu_health_checks.utilities.**handle_final_download_test**(*logs_folder*, *speed_log_filename*, *size*, *download_time*, *download_speed_mbps*, *minimum_previous_tests*, *std_deviations_limit*, *speed_min_mbps*)

Handles the result of the download used to measure speed.

This function stores the value in the speed logs, and checks if the speed is below the absolute minimum threshold or if it is too slow compared to usual values obtained if there are enough previous tests to make a significant comparison.

Args:
logs_folder (str): The folder to store the log files.

speed_log_filename (str): The name of the speed log file.

size (str): The size of the file downloaded to measure speed.

download_time (float): The download time in seconds.

download_speed_mbps (float): The download speed in Mbps.

minimum_previous_tests (int): The minimum number of previous tests required to compare current results with results usually obtained.

std_deviations_limit (int): The number of standard deviations used for comparison. If the current download speed is less than the average speed minus 'std_deviations_limit' times the standard deviation of the speed, this function returns False which implied that check_download_speed will not pass.

speed_min_mbps (float): The minimum download speed threshold in Mbps.

Returns:
bool: True if there is an error, False otherwise.

cpu_health_checks.utilities.**load_configuration**(*config_file*, *config_mode*)

Load configuration from a YAML file based on the specified mode.

Args:
config_file (str): The path to the configuration file. config_mode (str): The mode to select from the configuration file.

Returns:
dict or None: The configuration dictionary for the given mode, or None if an error occurs.

cpu_health_checks.utilities.**print_and_log_result**(*result*, *message_passed*, *message_failed*, *logger*)

Performs the common practice that based on the result of a check we print a message if result is True, or an error if result is False, and then log the corresponding message as info.

cpu_health_checks.utilities.**print_error**(*message*, *new_line=True*)

Print an error message in red color.

cpu_health_checks.utilities.**print_message**(*message*, *new_line=True*)

Print a message in green color.

cpu_health_checks.utilities.**print_warning**(*message*)

Print a warning message in yellow color.

cpu_health_checks.utilities.**run_command**(*command*)

Run a command in the shell and capture the output.

Args:
command (str): The command to run.

Returns:
CompletedProcess: The result of running the command.

PYTHON MODULE INDEX

C

INDEX