

Analysis of Gauge Tool

Vidhyashree Nagabhushana(v_n63)

1. INTRODUCTION

1.1 Overview of the Tool

Gauge is a free and open source framework for writing and running acceptance tests. It was officially released on June 18th, 2018 [1]. It has a consistent cross platform support for writing test code. Gauge has a rich syntax based on Markdown. Gauge supports writing step implementations in Java, C#, Python, Typescript and Golang. Gauge helps us create and maintain test suites. It has a modular architecture with extensible plugin support. Also, gauge helps us to focus on testing the flow of an application by making steps as reusable as possible. We will not need to build custom frameworks using a programming language. It also supports Data-Driven Testing, parallel execution, cross-browser testing, and behavior driven testing. Gauge has great support for VS Code. Gauge is simple to learn and flexible. The tool also presents test results in the output console of VS code. Alternatively, a very descriptive HTML report is generated with detailed information about the specifications that were executed, about the number of scenarios that were executed and counts of those that passed and failed. Gauge is supported by Windows, Linux and macOS.

1.2 Overview of the Input Problems

A class called “area” is created to compute the area of different shapes. The four functions “area_of_triangle”, “area_of_rectangle”, “area_of_trapezoid”, “area_of_hexagon” are defined which compute the respective areas of the shapes. The function “area_of_triangle” takes three parameters a,b and c and computes the area of a triangle with three sides known. The function “area_of_rectangle” takes as input the length and breadth values and returns the area. The function “area_of_trapezoid” takes as input two base values a, b and height h and returns the area of a trapezoid. The function “area_of_hexagon” takes as input the value of one side a and returns the area of the hexagon.

2. OBSERVATIONS

2.1 Installing the tool:

Gauge tool can be installed from the website link:

<https://gauge.org/>

After navigating to the website, click on Documentation. Select Installing Gauge from the left panel of options. Once that option is selected, it redirects to the Installing Gauge page.

As a part of the project, I installed Gauge for Windows OS and programming language as Python. The IDE/Editor is VS Code. The pre-requisites to install Gauge are:

- Windows OS
- Python Runner
- VS Code Editor

Steps for installation of the tool Gauge:

Select your setup to see the instructions for installing gauge.

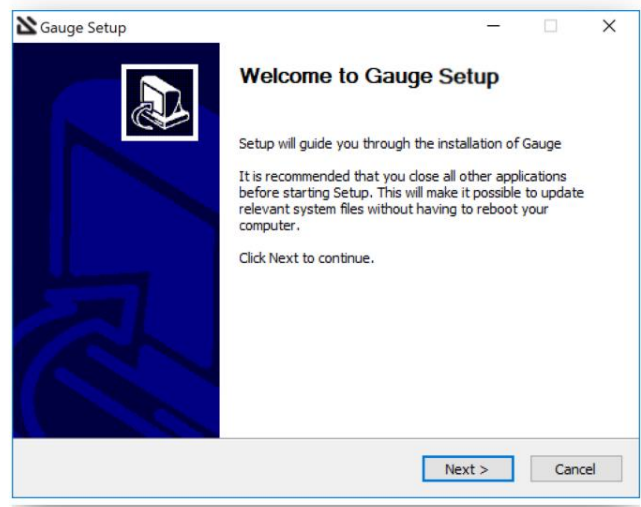
1. OS	2. Language	3. IDE/Editor
<input type="radio"/> Linux	<input type="radio"/> C#	<input checked="" type="radio"/> VS Code
<input type="radio"/> macOS	<input type="radio"/> Java	
<input checked="" type="radio"/> Windows	<input type="radio"/> JavaScript	
	<input checked="" type="radio"/> Python	
	<input type="radio"/> Ruby	

Select the OS as Windows, Language as Python and IDE/Editor as VS Code.

Download the installation bundle for Windows to get the latest stable release of Gauge.

Download the following installation bundle to get the latest stable release of Gauge.

Windows Installer



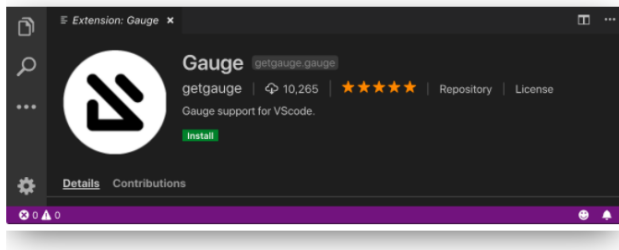
Once installing gauge is complete, install the gauge extension for VS Code Plugin.

1. Install the following Gauge extension for VS Code.

Gauge extension

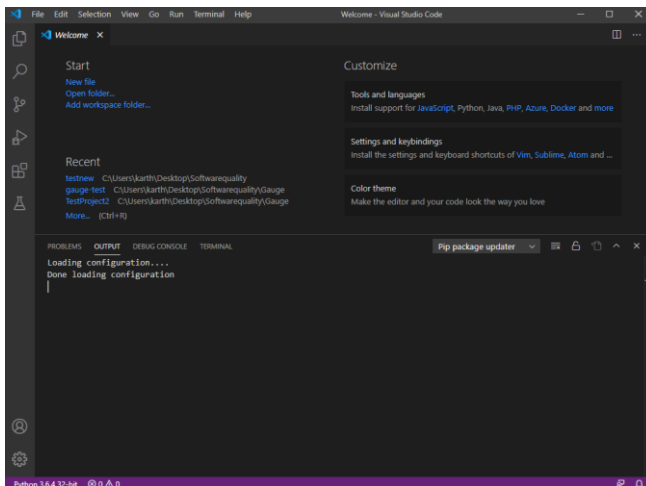
Click on the install button in the extension page.

2. On the extension page that opens in the IDEs, click the install button

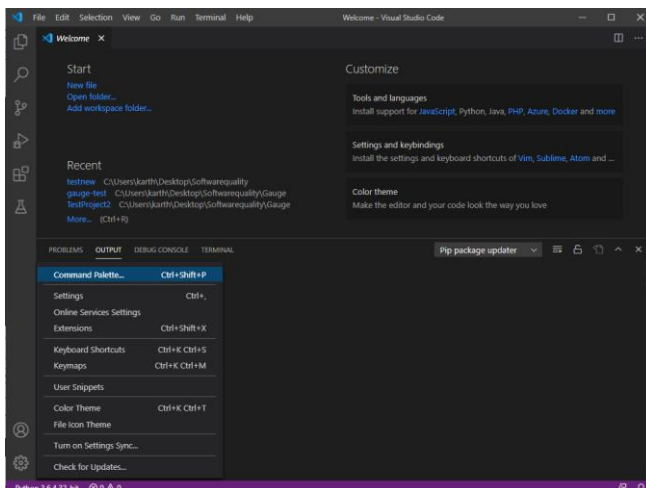



2.2 Learning the tool:

Install Gauge as mentioned above to learn the tool. Once installed, a welcome page like below is seen. This is the interface of Gauge with VS Code.

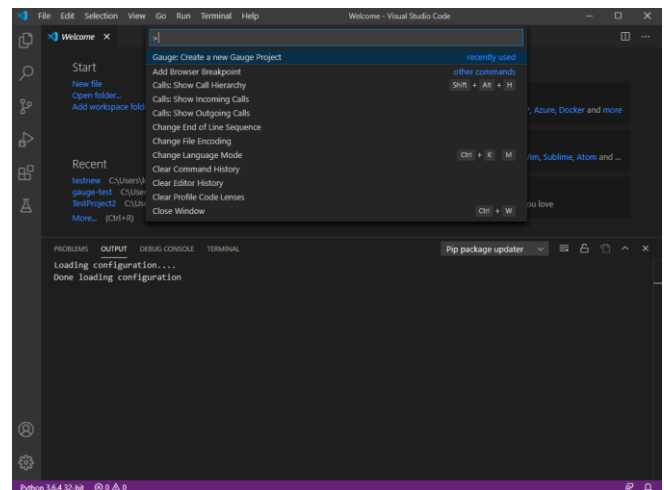


The first step in learning the tool is to explore different options available and how a new project can be created. Gauge projects can be created and executed in VS Code using the Gauge extension for VS Code. We will extensively use the editor's command palette to create a project.

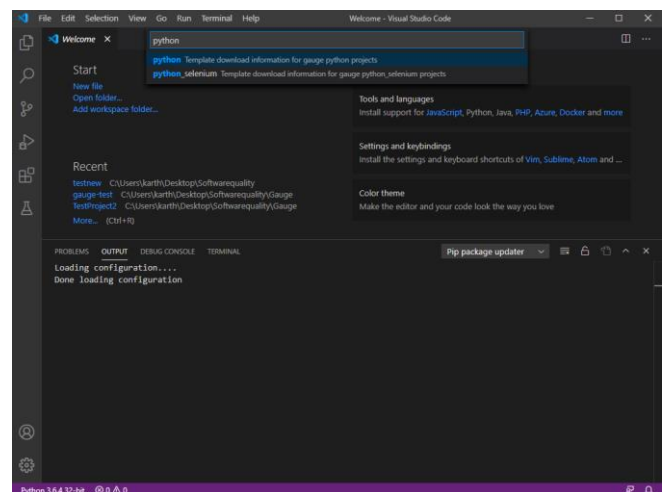


The command to access the editor's command palette is ctrl + shift + p. Alternatively, we can also click on the Manage icon  and select the Command Palette option. Once we click on command palette, execute the following command to create a new testing project in Gauge:

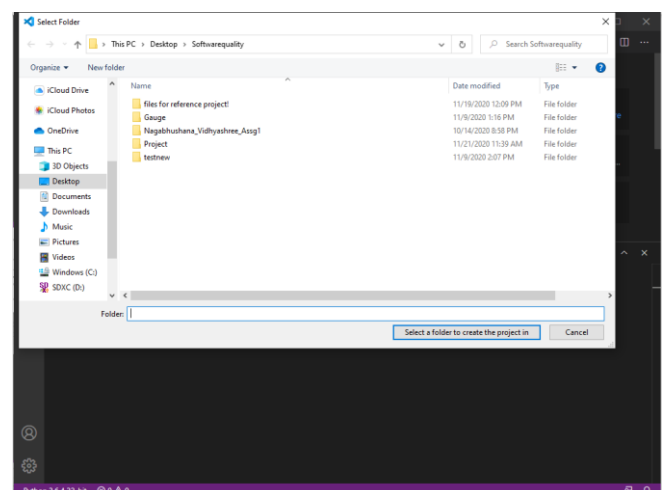
Gauge: Create new Gauge Project



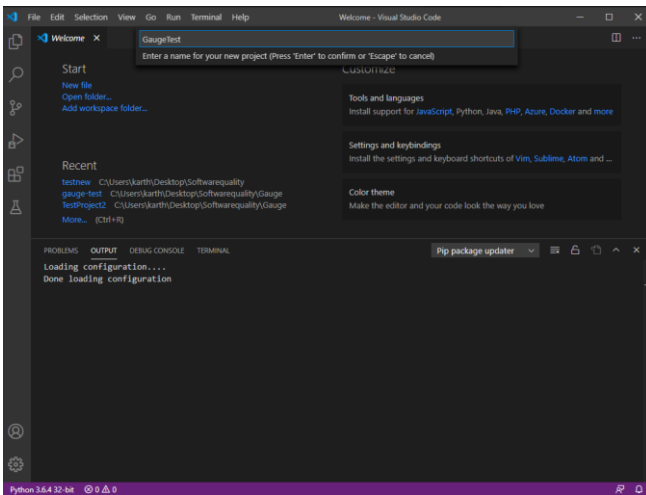
Select the python template to create the sample testing project.



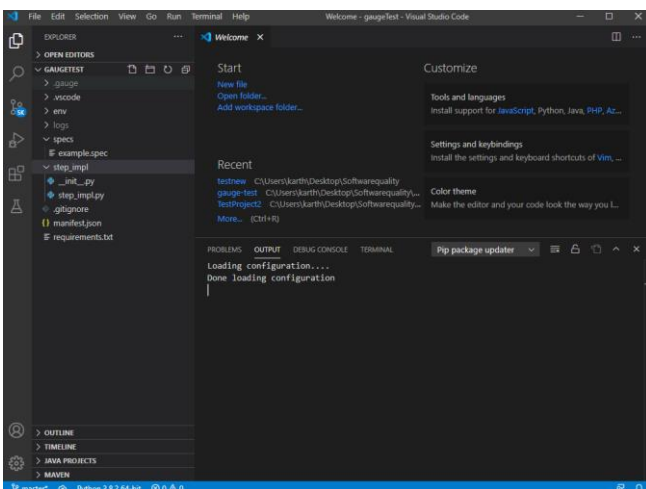
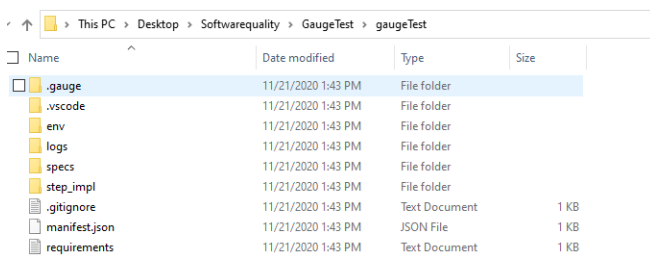
Choose a location to create a new folder to create a new project



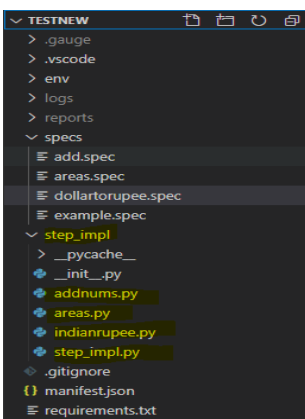
Give a name to the project.



After we've successfully created a gauge project, we'll be able to see a sample project with an example specification.

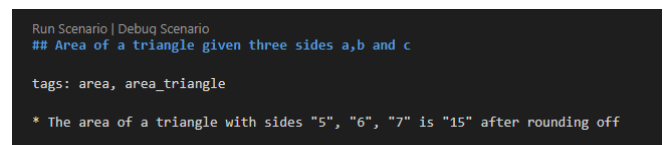


Once a project is created, we will now add all the python files that need testing in the step_impl folder as shown below:

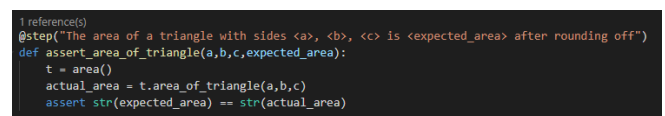


Once we add the .py python files, we will create specification files. A test specification (spec) is a detailed statement of what will be tested. In Gauge, these are written in a .spec file. A specification is a business test case which describes a particular feature of the application that needs testing. Gauge specifications support a .spec or .md file format and these specifications are written in a syntax similar to Markdown. Every spec can contain one or more Scenario. Every spec can be marked with labels using Tags.

When a Gauge project is created and initialized, a specs directory is automatically created with a sample file, example.spec. This sample file helps us understand how to write a specification. We create a spec file for our .py files keeping the example.spec as a reference. The specs written to test "area_of_triangle" is shown below:



The corresponding specification snippet shows the scenario to test the function which computes area of a triangle given three sides a,b and c. The components of a specification are as follows. The **"## Area of a triangle given three sides a,b and c"** in the specification is the scenario heading which starts with **"##"**. **"Tags: area, area_triangle"** are the tags associated with the scenario. Tags are used to associate labels with specifications or scenarios. With tags, we can group specs or scenarios of the same type to run them all at once using the command **"gauge run specs --tags area"**. This command when executed in the terminal runs all the scenarios or specifications that have the tag "area" which is replaceable by the "tag_name". Steps are another very important aspect of the Gauge specifications. Steps are executable components that are written using the markdown unordered list syntax. **"*The area of a triangle with sides "5", "6", "7" is "15" after rounding off"** is the step for the scenario to test the area of a triangle. We use **"*"** to write the step implementation in the scenario. Steps are defined to take values as parameters so that they can be reused with different parameter values. In the above scenario, **"5", "6", "7"** and **"15"** are parameter values enclosed within **"**. We must have step implementations for each of the scenarios or specifications defined in the .spec file. These step implementations are written in the .py files. The step implementation for the above scenario is as below:



Every step implementation has an equivalent code as per the language plugin used while installing Gauge. The code is run when the steps inside a spec are executed. The code must have the same number of parameters as mentioned in the step.

To define the step implementation, we use **"@step"** before the implementation. The step implementation should have the same statement as the step as discussed above. The number of parameters of the step and the corresponding step implementation should be the same and in the same order. In

the above example, the step is “*The area of a triangle with sides "5", "6", "7" is "15" after rounding off”, the step implementation is as shown below: “@step("The area of a triangle with sides <a>, , <c> is <expected_area> after rounding off "). We can notice that the sentences are the same and a maps to 5, b maps to 6, c maps to 7 and expected_area maps to 15. Once the step and step implementations are in place, we run the Scenario or the whole specification by clicking on the Run Scenario or Run specification button.

2.3 Analyzing the input problems using the tool:

As mentioned earlier, the input I am considering for the analysis of the tool is a class “area” which has four functions as shown below:

```
from getgauge.python import step, before_scenario, Messages
class area:
    def area_of_triangle(self,a,b,c):
        a = int(a)
        b = int(b)
        c = int(c)
        s = (a+b+c)/2
        area = (s*(s-a)*(s-b)*(s-c))**0.5
        return round(area)

    def area_of_rectangle(self,a,b):
        a = int(a)
        b = int(b)

        return a * b

    def area_of_trapezoid(self, a,b,h):
        a = int(a)
        b = int(b)
        h = int(h)

        area = ((a+b)/2)*h

        return round(area)

    def area_of_hexagon(self, a):
        a = int(a)
        area = ((3 * (3)**0.5)* (a * a))/2
        return round(area)
```

To test the above functions, we will have to write step implementations for each of the functions as described below.

The step implementation for the function “area_of_triangle” is as shown below:

```
1 reference(s)
@step("The area of a triangle with sides <a>, <b>, <c> is <expected_area> after rounding off")
def assert_area_of_triangle(a,b,c,expected_area):
    t = area()
    actual_area = t.area_of_triangle(a,b,c)
    assert str(expected_area) == str(actual_area)
```

As discussed earlier, the step implementation starts with the @step. The step implementation takes parameters a, b, c and expected_area. The definition of the assertion function calls the function area_of_triangle with parameters a, b, c and stores the result in actual_area. We then compare actual_area to the expected_area received as a parameter from the specification file. If both are equal, the tool passes the test, otherwise gives and error and fails the test.

The step implementation to test the function “area_of_rectangle” is as shown below:

```
1 reference(s)
@step("The area of a rectangle with length <l> and breadth <b> is <expected_area> after rounding off.")
def assert_area_of_rectangle(l,b,expected_area):
    r = area()
    actual_area = r.area_of_rectangle(l,b)
    assert str(expected_area) == str(actual_area)
```

The step implementation takes parameters l, b and expected_area. The definition of the assertion function calls the function area_of_rectangle with parameters l, b and stores the result in actual_area. We then compare actual_area to the expected_area received as a parameter from the specification file. If both are equal, the tool passes the test, otherwise gives and error and fails the test.

The step implementation to test the function “area_of_rectangle” with a table of values is as shown below:

```
1 reference(s)
@step("Below is the table with the values of length and breadth <table>")
def assert_area_of_rect(table):
    r = area()
    a = table.get_column_values_with_name("length")
    b = table.get_column_values_with_name("breadth")
    actual = []
    for a1,b1 in zip(a,b):
        actual.append(str(r.area_of_rectangle(a1,b1)))
    #print(actual)
    expected = table.get_column_values_with_name("Area")
    #print(expected)
    assert expected == actual
```

The step implementation takes a single parameter of a table here. This table has various length and breadths defined as columns and for each of the length and breadth, the corresponding area is also defined in a column in the specification file. We extract the column values into a and b. **a will be a list of lengths, b will be a list of breadths.** We loop over each of the values of length and breadth and call the function to compute the area for the respective values of lengths and breadths. We store the areas in a list “actual”. We now extract all the values in the column with name “area” into our “expected” list. We then compare actual to the expected received as a parameter from the specification file. If both are equal, the tool passes the test, otherwise gives and error and fails the test.

Similarly, the step implementations to test for the functions area_of_trapezoid and area_of_hexagon are as below:

```
1 reference(s)
@step("The area of a trapezoid with sides a = <a>, b = <b> and h = <h> after rounding off is <expected_area>")
def assert_area_of_trapezoid(a,b,h,expected_area):
    t = area()
    actual_area = t.area_of_trapezoid(a,b,h)
    assert str(expected_area) == str(actual_area)
```

```
1 reference(s)
@step("The area of a hexagon with side a = <a> is <expected_area>")
def assert_area_of_hexagon(a,expected_area):
    h = area()
    actual_area = h.area_of_hexagon(a)
    assert str(expected_area) == str(actual_area)
```

As we have now written all the step implementations, we will be using these step implementations in the specifications file. The specification will be created under the specs folder. The specification file to test the functions of the class “area” is created as “areas.spec”. The specification file for testing the areas class is as below:

```

E areas.spec > ...
Run Spec | Debug Spec
#Spec to find the area of different shapes

Run Scenario | Debug Scenario
## Area of a triangle given three sides a,b and c

tags: area, area_triangle

* The area of a triangle with sides "5", "6", "7" is "15" after rounding off

Run Scenario | Debug Scenario
## Area of a rectangle given two sides l and b

tags: area, area_rectangle

This scenario is to test for the area of a rectangle

* The area of a rectangle with length "10" and breadth "5" is "50" after rounding off.

Run Scenario | Debug Scenario
## Area of rectangle where values are passed using a table

tags: area, area_rectangle

This scenario is to test for the area of a rectangle where the input values are given as a table

* Below is the table with the values of length and breadth


| length | breadth | Area |
|--------|---------|------|
| 10     | 20      | 200  |
| 3      | 8       | 24   |
| 4      | 12      | 48   |
| 9      | 8       | 72   |



Run Scenario | Debug Scenario
## Area of a trapezoid where the values of a,b and h are given

tags: area, area_trapezoid

This scenario is to test for the area of a trapezoid

* The area of a trapezoid with sides a = "8", b = "5" and h = "17" after rounding off is "110"

Run Scenario | Debug Scenario
## Area of a hexagon where side a is given

tags: area, area_hexagon

This scenario is to test for the area of a hexagon

* The area of a hexagon with side a = "15" is "585"

```

In gauge, each test is called a scenario, so the above test written is a scenario. Considering each of the scenarios in the specifications separately, the scenario for testing the area_of_triangle is as below:

```

Run Spec | Debug Spec
#Spec to find the area of different shapes

Run Scenario | Debug Scenario
## Area of a triangle given three sides a,b and c

tags: area, area_triangle

* The area of a triangle with sides "5", "6", "7" is "15" after rounding off

```

In the above scenario, the specification header is defined using “#” and “#Spec to find the area of different shapes” is the specification heading. We then have “##Area of a triangle given three sides a, b and c” which is the heading of the scenario. After the scenario heading, we have defined tags. Tags are used to associate labels with specifications or scenarios. With tags, we can group specs or scenarios of the same type to run them all at once. We have written the step with “*”. In this step, instead of mentioning the parameters like in step implementation, we are mentioning the values “5”, “6”, “7” and “15”. Once we write the scenario heading, tags and step, we will see the Run Scenario button. We can click on the “Run Scenario” on to of each of the tests to run that scenario in the specification file. To run all the scenarios at once, we can click on the “Run Spec” option that is present at the top of the file.

One of the scenarios to test for area_of_rectangle is as below:

```

Run Scenario | Debug Scenario
## Area of rectangle where values are passed using a table

tags: area, area_rectangle

This scenario is to test for the area of a rectangle where the input values are given as a table

* Below is the table with the values of length and breadth


| length | breadth | Area |
|--------|---------|------|
| 10     | 20      | 200  |
| 3      | 8       | 24   |
| 4      | 12      | 48   |
| 9      | 8       | 72   |


```

In the above scenario, it can be noticed that there is a table with three columns length, breadth, and Area. As discussed above, the table acts as a value to the parameter <table> in the step implementation. We can extract the columns separately and process each of them to verify that the area corresponding to the length and breadth is correct.

The scenarios to test for the remaining functions are shown below:

```

Run Scenario | Debug Scenario
## Area of a rectangle given two sides l and b

tags: area, area_rectangle

This scenario is to test for the area of a rectangle

* The area of a rectangle with length "10" and breadth "5" is "50" after rounding off.

Run Scenario | Debug Scenario
## Area of a trapezoid where the values of a,b and h are given

tags: area, area_trapezoid

This scenario is to test for the area of a trapezoid

* The area of a trapezoid with sides a = "8", b = "5" and h = "17" after rounding off is "110"

Run Scenario | Debug Scenario
## Area of a hexagon where side a is given

tags: area, area_hexagon

This scenario is to test for the area of a hexagon

* The area of a hexagon with side a = "15" is "585"

```

When we click on “Run scenario” for any of the above scenarios, the output will be displayed stating if the test passed or failed:

```

PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL

Python: 3.6.4
# Spec to find the area of different shapes
## Area of a trapezoid where the values of a,b and h are given

Successfully generated html-report to -> c:\Users\karth\Desktop\Softwarequality\testnew\reports\html-report\index.html

Specifications: 1 executed 1 passed 0 failed 0 skipped
Scenarios: 1 executed 1 passed 0 failed 0 skipped

Total time taken: 199ms
Success: Tests passed.

```

When we click on “Run Spec”, all the scenarios in that spec will get executed and the output will be displayed stating if the test passed or failed:

```

Python: 3.6.4
# Spec to find the area of different shapes
## Area of a triangle given three sides a,b and c
## Area of a rectangle given two sides l and b
## Area of rectangle where values are passed using a table
## Area of a trapezoid where the values of a,b and h are given
## Area of a hexagon where side a is given

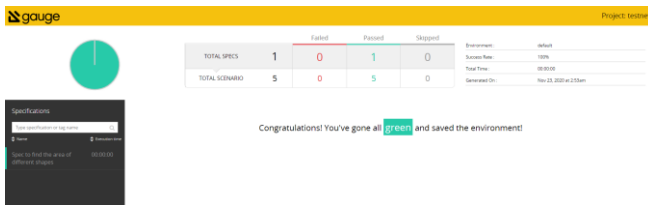
Successfully generated html-report to -> c:\Users\karth\Desktop\Softwarequality\testnew\reports\html-report\index.html

Specifications: 1 executed 1 passed 0 failed 0 skipped
Scenarios: 5 executed 5 passed 0 failed 0 skipped

Total time taken: 182ms
Updates are available. Run 'gauge update -c' for more info.
Success: Tests passed.

```

We can notice the spec heading and the scenario headings visible in the report. Alternatively, we can also view the HTML version of the report:



2.4 Strengths:

- **Simple, flexible and user friendly:**

Gauge is very simple to learn. The syntax is very easy to use and understandable to even a layman. Any person with not much prior experience of testing can read and understand the test scenarios with ease [4].

Below is a sample scenario:

```
Run Spec | Debug Spec
# Specification to add numbers

This spec is to add two numbers

Run Scenario | Debug Scenario
## Add two numbers

tags: sum

* The sum of "10" and "20" is "30".
```

The above specification has a comment “This spec is to add two numbers” which is straightforward and helps those who are new to this tool understand what exactly the specification is about. Since we can add comments to specifications and scenarios separately, it makes it very easy to understand what the specification is about or what each scenario is about. Also, the very simple syntax makes it easy to learn and understand.

- **Usage of Tags:**

One of the key features of Gauge is the usage of tags. Tags are used to associate labels with specifications or scenarios. Tags help in searching or filtering specs or scenarios. Tags are written as comma separated values in the specification with a prefix **tags:**. Both scenarios and specifications can be separately tagged, however, only one set of tags can be added to a single specification or scenario. A tag applied to a spec automatically applies to a scenario as well. When the number of tags used is more, tags can be defined in multiple lines to enhance readability. If scenarios in different spec files have the same tag, they are grouped together and when we try executing the scenarios, they get executed. The command used in the terminal to execute scenarios with the same tag is as below:

gauge run specs -tags tag_name

An example snippet to show all scenarios with tag as area run together:

```
C:\Users\karth\Desktop\Softwarequality\testnew>gauge run specs --tags area

Python: 3.6.4
# Spec to find the area of different shapes
## Area of a triangle given three sides a,b and c P
## Area of a rectangle given two sides l and b P
## Area of rectangle where values are passed using a table P
## Area of a trapezoid where the values of a,b and h are given P
## Area of a hexagon where side a is given P

Successfully generated html-report to -> C:\Users\karth\Desktop\Softwarequality\testnew\reports\html-report\index.html

Specifications: 1 executed 1 passed 0 failed 0 skipped
Scenarios: 5 executed 5 passed 0 failed 0 skipped

Total time taken: 310ms
Updates are available. Run 'gauge update -c' for more info.
```

- **Using steps to write scenarios:**

The most important strength of Gauge is the usage of Steps to write scenarios. Writing test scenarios is very simple and usually, we can write tests in a single line with all the necessary parameter values, this will make it easy for anyone to learn or to implement tests using steps. Below is an example of a simple scenario using one step:

```
Run Scenario | Debug Scenario
## Dollar to rupee

tags: conversion

* The conversion of "10" dollars to rupee is "740"
```

- **Parallel Execution:**

Specs can be executed in parallel to run the tests faster. Running tests in parallel creates a number of execution streams depending on the number of CPU cores available on our system and distributes the load among worker processes. This is a very helpful feature when we want to reduce the execution time for a very large test set.

To run the specifications on different cores, we use the command below:

gauge run -parallel specs

When we run the command in the terminal, the output is as below:

```
C:\Users\karth\Desktop\Softwarequality\testnew>gauge run --parallel specs

Python: 3.6.4
[runner: 4] # Specification to add numbers
[runner: 1] # Spec to find the area of different shapes
[runner: 2] # Dollar to rupee spec
[runner: 3] # Specification Heading
[runner: 4] ## Add two numbers
[runner: 2] ## Dollar to rupee
[runner: 1] ## Area of a triangle given three sides a,b and c
[runner: 3] ## Vowel counts in single word
[runner: 2] ## Dollar to rupee symbols
[runner: 4]
[runner: 1] ## Area of a rectangle given two sides l and b
[runner: 3] ## Vowel counts in multiple word
[runner: 2]
[runner: 1] ## Area of rectangle where values are passed using a table
[runner: 1] ## Area of a trapezoid where the values of a,b and h are given
[runner: 3]
[runner: 1] ## Area of a hexagon where side a is given
[runner: 1]
Successfully generated html-report to -> C:\Users\karth\Desktop\Softwarequality\testnew\reports\html-report\index.html

Specifications: 4 executed 4 passed 0 failed 0 skipped
Scenarios: 10 executed 10 passed 0 failed 0 skipped

Total time taken: 4.075s
Updates are available. Run 'gauge update -c' for more info.
```

We can notice that the load is distributed among the different CPU cores. There are 4 cores in my system and hence, we can see that the load is distributed amongst runners 1, 2, 3 and 4, each indicating one of the CPU cores. We can also use the command below to run the scenarios in parallel:

gauge run -p specs

The above command gives the same output as earlier. The number of parallel execution streams can be specified by using the `-n` flag.

gauge run -parallel -n=3 specs

The above command will distribute the load between three cores of the CPU as we specified `n` as 3. The output is as below:

```
C:\Users\karth\Desktop\Softwarequality\testnew\gauge run --parallel -n=3 specs
Python: 3.6.4
[runner: 2] # Specification to add numbers
[runner: 2] ## Add two numbers
[runner: 1] # Spec to find the area of different shapes
[runner: 3] # Dollar to rupee spec
[runner: 1] ## Area of a triangle given three sides a,b and c
[runner: 3] ## Dollar to rupee
[runner: 2]
[runner: 2] # Specification Heading
[runner: 3] ## Dollar to rupee symbols
[runner: 1] ## Area of a rectangle given two sides l and b
[runner: 2] ## Vowel counts in single word
[runner: 3]
[runner: 1] ## Area of rectangle where values are passed using a table
[runner: 2] ## Vowel counts in multiple word
[runner: 1] ## Area of a trapezoid where the values of a,b and h are given
[runner: 2]
[runner: 1] ## Area of a hexagon where side a is given
[runner: 1]
Successfully generated html-report to -> C:\Users\karth\Desktop\Softwarequality\testnew\reports\html-report\index.html

Specifications: 4 executed    4 passed    0 failed    0 skipped
Scenarios:      10 executed   10 passed    0 failed    0 skipped

Total time taken: 3.721s
Updates are available. Run 'gauge update -c' for more info.
```

We should notice that in the above output snippet, only runners 1, 2 and 3 are considered which implies that the load was distributed between 3 CPU cores.

- **Very descriptive Output:**

Once the specifications are run, we will get immediate feedback in the output console of VS Code as shown below:

```
Python: 3.6.4
# Spec to find the area of different shapes
## Area of a triangle given three sides a,b and c
## Area of a rectangle given two sides l and b
## Area of rectangle where values are passed using a table
## Area of a trapezoid where the values of a,b and h are given
## Area of a hexagon where side a is given

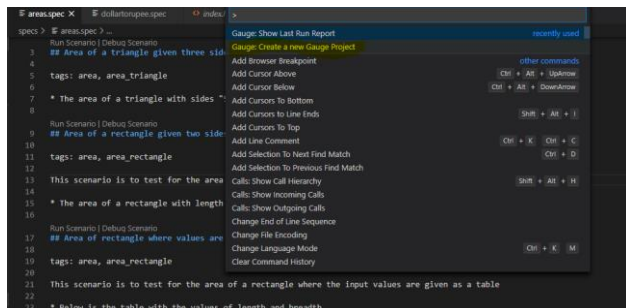
Successfully generated HTML-report to -> c:\Users\karth\Desktop\Softwarequality\testnew\reports\html-report\index.html

Specifications: 1 executed 1 passed 0 failed 0 skipped
Scenarios: 5 executed 5 passed 0 failed 0 skipped

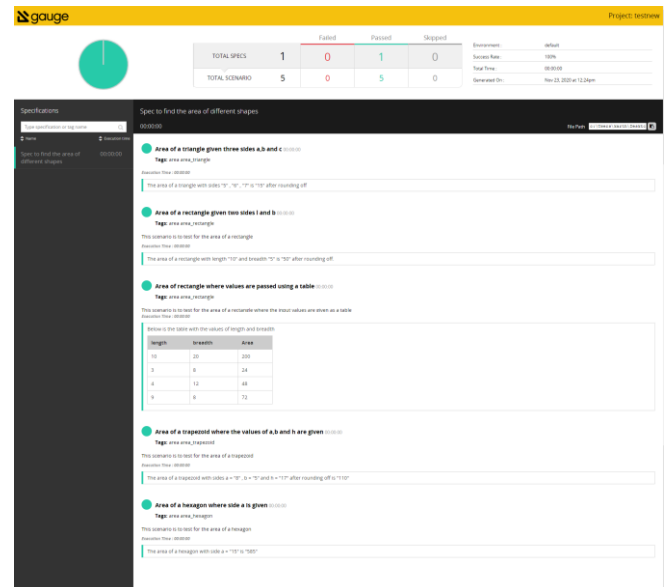
Total time taken: 89ms
Updates are available. Run 'gauge update -c' for more info.
Success: Tests passed.
```

We can also view the HTML report of tests run by following either of the steps below: Open the html report by clicking on the view summary link in VS Code.

Alternatively, In the editor's command palette, we can type **Gauge: Show Last Run Report** to view the report in the browser:



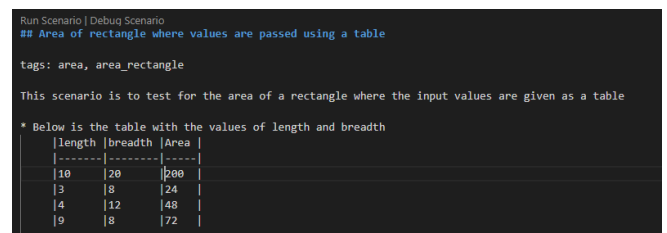
Here is what a Gauge report looks like:



We can notice that it is very descriptive. Results are tabulated with the total number of scenarios that were executed, total number of specs involved, number of passed specs and scenarios and the count of those that failed. We also have the skipped tests count. It shows the success rate, date of generation of the report and also the total time taken to run all the tests.

- **Data driven testing:**

Gauge also supports data driven testing using Markdown tables.



We are feeding the table with 3 columns as the input values to our step implementation shown below:

```
1 reference(s)
@step("Below is the table with the values of length and breadth <table>")
def assert_area_of_rect(table):
    r = area()
    a = table.get_column_values_with_name("length")
    b = table.get_column_values_with_name("breadth")
    actual = []
    for a1,b1 in zip(a,b):
        actual.append(str(r.area_of_rectangle(a1,b1)))
    #print(actual)
    expected = table.get_column_values_with_name("Area")
    #print(expected)
    assert expected == actual
```

This will execute the scenario for all rows in the table. In the example above, we refactored a specification to be concise and flexible without changing the implementation.

- **Modular Architecture:**

We have seen earlier about how specifications are structured. The specification file is divided into modules where we can consider each of the test/scenario to be a module. We can run one module at a time by clicking on

the corresponding “Run Scenario” button or we can run the whole spec using the “Run Spec”. The scenarios are further divided into scenario heading, tags, comments, steps, and parameters which makes it more structured and modular. Below is a sample scenario:

```
Run Scenario | Debug Scenario
## Dollar to rupee

tags: conversion

This spec tests the conversion value from dollars to rupees

* The conversion of "10" dollars to rupee is "740"
```

2.5 Weakness:

- **False Results when a test is skipped:**

If a test is skipped due to a syntax error in the code, instead of updating it as a failed scenario, it will skip execution of the test. Unfortunately, the final test shows “test passed” when that is not the case. Below is a snippet of a time when there was a syntax error, but the test was skipped, and the status was “test passed”:

```
16 def assert_rupee_symbol(dollar, rupee):
17     degree_sign = u'\u2013'
18     print(degree_sign)
19     assert '$' == degree_sign
20
21
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
File "c:\Users\karth\Desktop\Softwarequality\testnew\step_impl\indianrupee.py", line 20
    degree_sign = u'\u2013'
                        ^
SyntaxError: invalid syntax
Successfully generated html-report to => c:\Users\karth\Desktop\Softwarequality\testnew\reports\html-report\index.html
Specifications: 0 executed 0 passed 0 failed 1 skipped
Scenarios: 0 executed 0 passed 0 failed 1 skipped
Total time taken: 140ms
Success: Tests passed.
```

This result can be very misleading when large sets of tests are executed and there are a few syntax errors, a few tests skipped, but the test says success and the person keeping a track might be misled and ignore the failed cases and update the records stating all tests have passed. For some critical application functions, this might lead to more serious problems.

- **Limited support and training:**

There is very limited support and training available for the tool. Since the tool is relatively new and there are very few articles and blogs online, the only source of information to learn more about the tool is the Documentation in their website <https://gauge.org/>. There are no live online connects or Webinars that can be referred to for further understanding [3]. If we encounter any issues during the installation or analysis of the tool, not many similar issues or their fixes can be found on their website. We must create a bug in GitHub. Contacting individuals get challenging.

- **Deployment limited to Mac, Windows, and Linux:**

The tool can be deployed in devices that run on Mac, Windows, or Linux only. We cannot use the tool in mobile devices with Android OS or on iPhone/iPads [3].

- **Not Unicode Compliant:**

Gauge is not Unicode compliant [3]. Consider the code snippet below:

```
1 reference(s)
@step("The dollar symbols should be the same for the <symbol> and the unicode value")
def assert_dollar_symbol(symbol):
    #print(symbol)
    unicode_val = u"\u0024" #dollar. symbol = u"\u00b0" for degree but gives questionmark in a box
    sentence = ("This is the dollar symbol : " + unicode_val)
    #print(sentence)
    assert sentence == "This is the dollar symbol : " + symbol
```

In the above step implementation, Unicode value is defined in the runner, i.e., the .py file and we are trying to print it, so there is no problem.

```
Run Scenario | Debug Scenario
## Dollar to rupee symbols

tags: unicode_compliance_runner

* The dollar symbols should be the same for the "$" and the unicode value
```

This is the specification for the step implementation. Here, we are just passing the symbol “\$” as a parameter and comparing it with the Unicode value defined inside the runner. Hence, there is no issue. Runner is python but specification is a part of gauge. Since we are not passing the Unicode value from gauge to runner, it has no effect. Scenario executes and passes the case:

```
Python: 3.6.4
# Dollar to rupee spec
## Dollar to rupee symbols

Successfully generated html-report to => c:\Users\karth\Desktop\Softwarequality\testnew\reports\html-report\index.html
Specifications: 1 executed 1 passed 0 failed 0 skipped
Scenarios: 1 executed 1 passed 0 failed 0 skipped
Total time taken: 81ms
Updates are available. Run 'gauge update -c' for more info.
Success: Tests passed.
```

Let us consider another example where there is a problem:

```
1 reference(s)
@step("This tests that the unicode value <val> is passed correctly from gauge to the runner and it prints <symbol>")
def assert_same_symbol(val, symbol):
    print("This is the symbol passed : " + symbol + " and this is the Unicode value passed : " + val)
    assert val == symbol
```

In the above step implementation, we are having two parameters “val” and “symbol”. Val is supposed to get the Unicode value from Gauge (spec file) and symbol is supposed to be “\$” sent through Gauge to the runner too. We expect that the symbol “\$” and the Unicode value for the symbol both to be equal. Consider the scenario for the above snippet:

```
Run Scenario | Debug Scenario
## Passing Unicode value as a parameter

tags: unicode_compliance

* This tests that the unicode value "\u0024" is passed correctly from gauge to the runner and it prints "$"
```

Notice the highlight where a Unicode value is passed from the spec file of Gauge to the step implementation in the runner python file. We expect that to print “\$” too when we execute the scenario, but the runner gets the Unicode value instead of the corresponding symbol hence failing the test:


```

$ dollar to reuse spec
## Passing unicode value as a parameter
This is the symbol passed : $ and this is the unicode value passed : u0024

Failed Step: This tests that the unicode value "u0024" is passed correctly from gauge to the runner and it prints "1"
Specification: c:\Users\karth\Desktop\Softwarequality\testnew\specs\dollartunespec.spec:25
Error Message: Exception occurred
Stacktrace:
Traceback (most recent call last):
  File "C:\Users\karth\AppData\Local\Programs\Python\Python32\lib\site-packages\getgauge\executor.py", line 31, in execute_method
    step.impl(*params)
  File "C:\Users\karth\Desktop\Softwarequality\testnew\step_impl\indianrunes.py", line 31, in assert_same_symbol
    assert val == symbol
AssertionError

Successfully generated html-report to => c:\Users\karth\Desktop\Softwarequality\testnew\reports\html-report\index.html

Specifications: 1 executed 0 passed 1 failed 0 skipped
Scenarios: 1 executed 0 passed 1 failed 0 skipped

Total time taken: 38ms
Updates are available. Run 'gauge update -c' for more info.
Error: tests failed.

```

Hence, Gauge is not Unicode compliant.

- **Passing an array:**

There is no facility to pass an array of values to the step implementation as a parameter. I could not figure out a way to pass an array to the step implementation. After a lot of research on their website and after contacting the developer of the tool on LinkedIn, the developer mentioned there was no way we could pass an array as a parameter. We have tables instead, but tables will take different values and corresponding expected outputs.

- **Time format for small tests:**

The HTML Report shows time in HH:MM:SS. When the time taken to execute scenarios is less than a second, i.e., time is in milli seconds or less, the results does not show that. Consider the below example scenario:

```

Python: 3.6.4
# Specification to add numbers
## Add two numbers

Successfully generated html-report to => c:\Users\karth\Desktop\Softwarequality\testnew\reports\html-report\index.html

Specifications: 1 executed 1 passed 0 failed 0 skipped
Scenarios: 1 executed 1 passed 0 failed 0 skipped

Total time taken: 75ms
Updates are available. Run 'gauge update -c' for more info.
Success: Tests passed.

```

The total time taken to execute the simple scenario is 75 ms. Consider the report now:

		Failed	Passed	Skipped	Environment:
TOTAL SPECS	1	0	1	0	default
TOTAL SCENARIO	1	0	1	0	Success Rate: 100%
					Total Time: 75ms
					Generated On: Nov-23, 2020 at 1:38pm

We see that the time shown is 00:00:00 as there is no way to accommodate milli seconds.

- **Parallel execution by using threads:**

Parallel execution by using threads is not supported for any language except for Java [1]. Below is the screenshot of the Note that confirms the same from the gauge official website:

<https://docs.gauge.org/execution.html?os=windows&language=python&ide=vscode>

Note:

Currently, only the Java language runner supports parallel execution of specs by using threads.

- **All parameters are considered strings:**

Since python is a loosely typed language and Gauge passes all the values as strings, we must explicitly convert the values to their specific types. For java and other languages, the values are collected as the data types mentioned in the function definition even when gauge passes it as a string. There should be a provision to

indicate the type of inputs in the specs so that loosely typed languages can work with the values without having to explicitly convert them each time.

```

def add_nums(a,b):
    sum = int(a) + int(b)
    return (sum)

```

To define a simple function too, had to convert the values into integers.

2.6 Suggested Improvements:

- Can add Unicode compliance to the tool.
- They could conduct more in person or live sessions for anyone who wants more help understanding the tool. They can have webinar and make the videos available on their website.
- Time format can be edited to accommodate milli seconds when small tests are run.
- They should fix the issue where incorrect test results are obtained when the tests are skipped.
- Make it available on Mobile devices.
- Could add features where arrays can be passed as parameters to the step implementations.
- There could be a provision to indicate the type of inputs in the specs so that loosely typed languages can work with the values without having to explicitly convert them each time.
- Currently, Gauge is available for C#, Java, JavaScript, Python, Ruby. There can be provision for many more languages like C++.
- Can add the feature to allow parallel execution using threads for other languages like C#, Ruby, Python. Currently, it is only available for Java.

3. CONCLUSION

Testing is one of the most critical steps in a software development process. Throughout the development process, there are several types of testing that are performed like Unit Testing, Integration Testing, Regression Testing, Acceptance Testing. Of the listed techniques above, Acceptance testing, as the name suggests is the testing performed by the customer. Once the System Testing process is completed by the testing team and is signed-off, the entire Product/application is handed over to the customer/few users of customers/both, to test for its acceptability i.e., Product/application should be flawless in meeting both the critical and major Business requirements. Also, end-to-end business flows are verified similar as in real-time scenario [2]. In acceptance testing, only the functionality is verified to ensure that the product meets the specified acceptance criteria. There is no need for design/implementation knowledge. Since the customers are the ones that perform acceptance testing, in majority of the cases, customers are not well aware about software development or are not very technically sound to understand

the underlying design and implementation details and perform the testing. This calls for a testing tool that is very simple, easy to learn and user friendly to use. The customer should be able to understand the instructions to perform the testing. Hence, Gauge is a perfect tool that can be chosen for Acceptance testing as it is very simple, flexible and user friendly. It also gives very descriptive results for the customer to keep a track of the tests passed, skipped and the ones that failed. Gauge is a free and open source framework for writing and running acceptance tests. Gauge has a rich syntax based on Markdown. Writing specifications is very easy and it supports many programming languages. Having analyzed the tool for some time, I can conclude that testing gets easier with Gauge. It has many interesting features like parallel execution that makes the execution of many specs and scenarios faster.

4. REFERENCES

- [1] <https://docs.gauge.org/?os=windows&language=python&ide=vscode>
- [2] <https://www.softwaretestinghelp.com/what-is-acceptance-testing/>
- [3] <https://www.capterra.com/p/147883/Gauge/>
- [4] <https://blog.getgauge.io/why-we-built-gauge-6e31bb4848cd>

5. APPENDIX

All test cases and codes used for testing are embedded in the report as and when necessary in the form of screenshots.