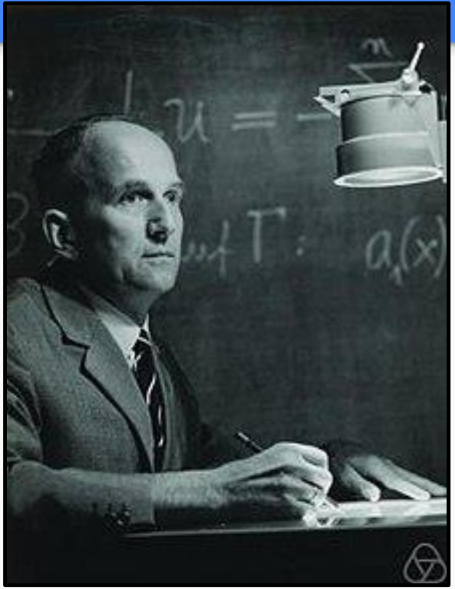
CSCI 141
Computer Science I

01-Testing

Problem: The Collatz Conjecture

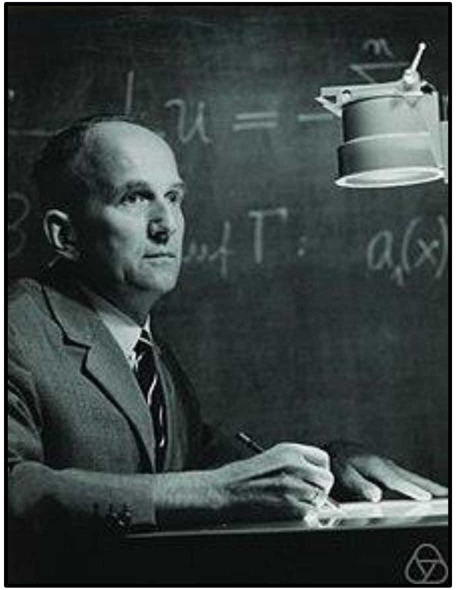
- The Collatz Conjecture states that, for any positive integer N, the sequence described by the following mathematical function will always reach N=1.
  - *F(N) = F(N/2)* if N is even.
  - *F(N) = F(3N + 1)* if N is odd.
  - *F(N)* where N < 1 is undefined.
- Our first task will be to write a *recursive* function that counts the number of steps it takes to reach 1 from some arbitrary N.



Lothar Collatz was a German mathematician. He originally proposed the Collatz Conjecture in 1937

Activity: Collatz by Hand

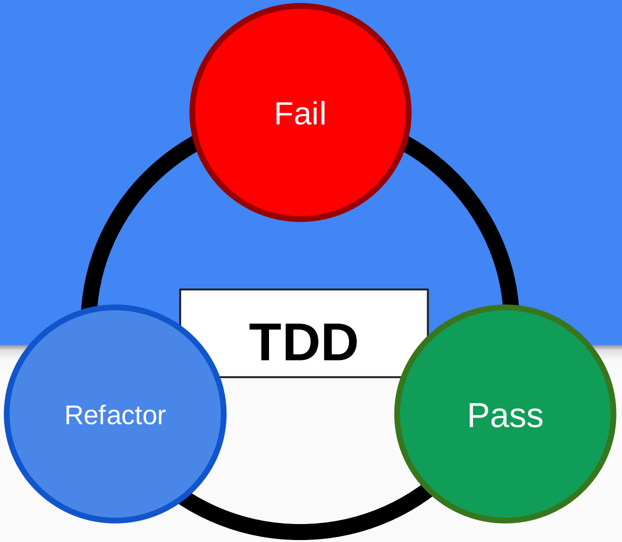
$$collatz(N) = \begin{cases} 1: & \text{if N is 1} \\ collatz(N/2): & \text{if N is even} \\ collatz(3N+1): & \text{if N is odd} \end{cases}$$



(group): Determine the sequence that ends with 1 using the following starting values for N:

- 1 1 (1 STEP)
- 2 2 1 (2 STEPS)
- 10 10 5 16 8 4 2 1 (7 STEPS)
- 21 21 64 32 16 8 4 2 1 (8 STEPS)
- 13 13 40 20 10 5 16 8 4 2 1 (10 STEPS)

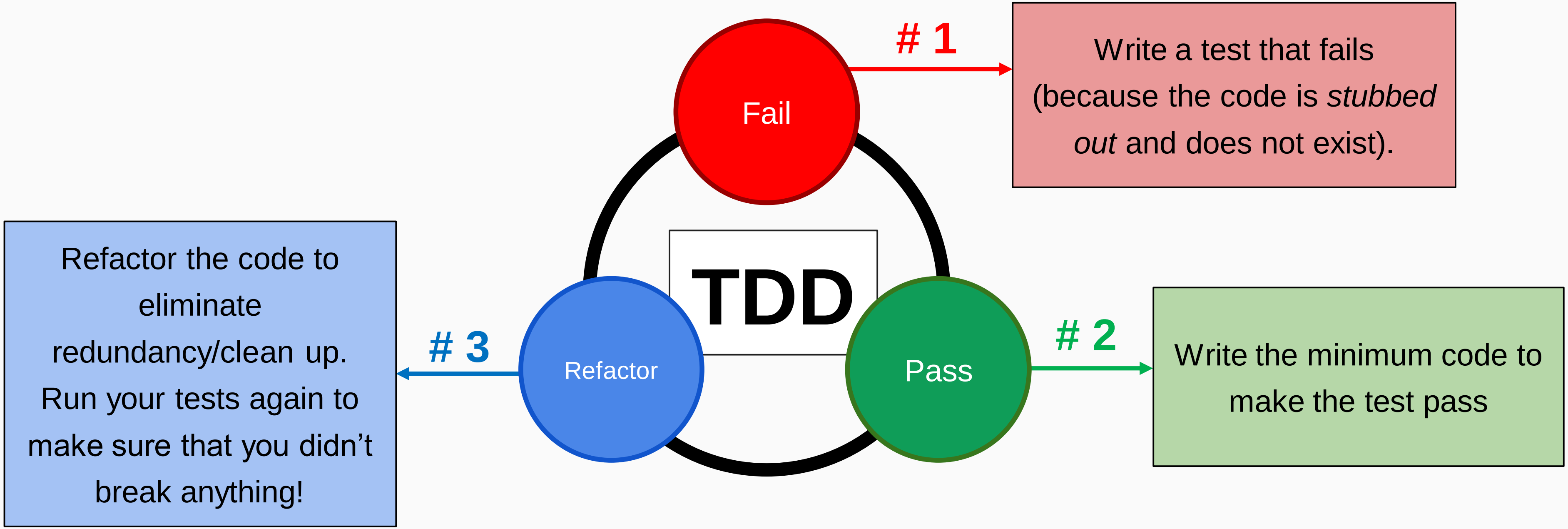
Testing



- So far this semester, we have encouraged you to implement your solution, and then test it.
  - This kind of testing is called **Test Last Development (TLD)**; you first implement your program’s functionality and then write the tests for it after

- A common practice in industry is Test Driven Development (TDD)
  - You write the tests first, and then implement your functionality after
  - Incrementally, you develop your solution to pass all the tests you previously wrote

Test Driven Development



1

Write a test that fails (because the code is *stubbed out* and does not exist).

2

Write the minimum code to make the test pass

3

Refactor the code to eliminate redundancy/clean up. Run your tests again to make sure that you didn’t break anything!

TDD Framework: Production Code

- The program to be delivered to the customer goes in **`collatz.py`**
    - This code is referred to as production code

collatz.py

```
def collatz(n):
  pass # un-implemented

def main():
  pass # un-implemented

if __name__ == "__main__”:
    main()
```

The `collatz()` function implementation goes here

The `main()` function implementation the customer will run goes here

Only invoke the `main()` function if this is the program being run directly by the interpreter

TDD Framework: Test Suite

- The test code, not intended for the customer, goes in a separate module, **`test_collatz.py`**

`test_collatz.py`

```
import collatz

def run_tests():
  pass # un-implemented

if __name__ == "__main__":
    run_tests()
```

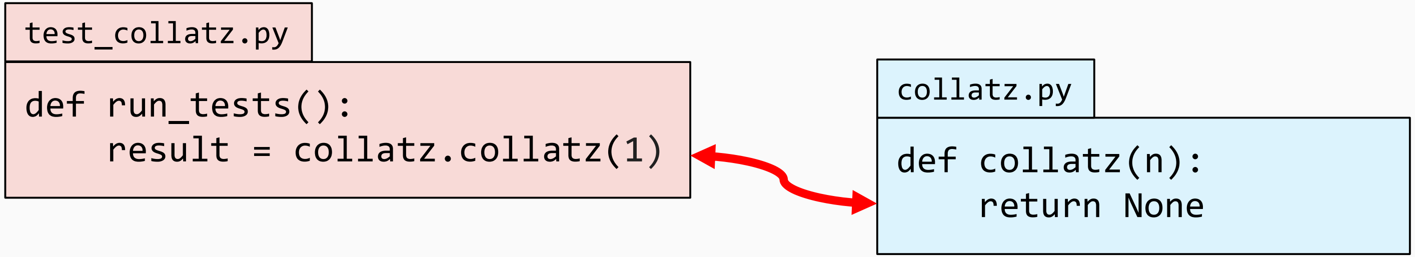
Import the main module to get access to `collatz()`

All unit tests of `collatz()` go here

Only invoke the `run_tests()` function if this is the program being run directly by the interpreter

Stubbing Functions

- In order to write a test that fails, we first need to stub out the collatz() function. A stubbed function has the following properties:
  - A proper function signature that can be called by our tester
  - A function with an initially minimum body, e.g. a placeholder `return` or `pass`



test_collatz.py

```
def run_tests():
    result = collatz.collatz(1)
```

collatz.py

```
def collatz(n):
    return None
```

Anatomy of a Good Test

- Good tests have the following characteristics
  - Are small in nature and test one thing
  - Are fast and execute quickly
- Unit testing is the practice of the developer writing automated tests to ensure a section of an application, e.g. a unit, behaves as intended
- Tests should be re-run every time the code is changed

Step 1A: Fail

- By definition, we know `collatz(1)` should return 1

test_collatz.py

```
def run_tests():
  result = collatz.collatz(1) # n = 1, result = None
  if result == 1:             # None != 1
        print("passed")
  else:
        print("failed for 1",
            "; expected 1 ",
            "but got", result)
```

collatz.py

```
def collatz(n):
  return None
```

```
collatz(1) = None
```

output

```
Failed for 1
; expected 1
but got None
```

Step 1B: Pass

- Implement `collatz()` so it works for n=1

test_collatz.py

```
def run_tests():
  result = collatz.collatz(1) # n = 1, result = 1
  if result == 1:             # 1 == 1
        print("passed")
  else:
        print("failed for 1",
           "; expected 1 ",
           "but got", result)
```

collatz.py

```
def collatz(n):
  if n == 1:
    return 1
```

```
collatz(1) = 1
```

output

```
passed
```

Step 1C: Refactor

- With only a single test case, there is no need to refactor the code yet

test_collatz.py

```
def run_tests():
  result = collatz.collatz(1)
  if result == 1:
        print("passed")
  else:
        print("failed for 1",
           "; expected 1 ",
           "but got", result)
```

collatz.py

```
def collatz(n):
  if n == 1:
    return 1
```

Step 2A: Fail

- By definition, we assume if N is even, `collatz(N)` should return 1

test_collatz.py

```
def run_tests():
  # previous test for n=1 not shown here
  result = collatz.collatz(2) # n = 2, result = None
  if result == 1:             # None != 1
        print("passed")
  else:
        print("failed for 2",
           "; expected 1 ",
           "but got", result)
```

collatz.py

```
def collatz(n):
  if n == 1:
    return 1
```

```
collatz(2) = None
```

output

```
Failed for 1
; expected 1
but got None
```

Step 2B: Pass

- By definition, we assume if N is even, `collatz(N)` should return 1

test_collatz.py

```
def run_tests():
  …
  result = collatz.collatz(2) # n = 2, result = 1
  if result == 1:             # 1 == 1
        print("passed")
  else:
        print("failed for 1",
           "; expected 1 ",
           "but got", result)
```

collatz.py

```
def collatz(n):
  if n == 1:
    return 1
  elif n % 2 == 0:
    return collatz(n // 2)
```

```
collatz(2) = collatz(2)
           = collatz(1)
           = 1
```

output

```
passed
passed
```

Step 2C: Refactor

- Each test has a similar structure that will keep repeating for each new test case we add

- Let’s refactor the code to capture that duplication into a single new function, collatz_test(), that we can keep re-using

```
def run_tests():
  result = collatz.collatz(1)
  if result == 1:
        print("passed")
  else:
        print("failed for 1",
           "; expected 1 ",
           "but got", result)

  result = collatz.collatz(2)
  if result == 2:
        print("passed")
  else:
        print("failed for 2",
           "; expected 1 ",
           "but got", result)
```

Step 2C: Refactor – collatz_test()

```
def run_tests():
  result = collatz.collatz(1)
  if result == 1:
        print("passed")
  else:
        print("failed for 1",
           "; expected 1 ",
           "but got", result)

  result = collatz.collatz(2)
  if result == 1:
        print("passed")
  else:
        print("failed for 1",
           "; expected 1 ",
           "but got", result)
```

```
def test_collatz(n, expected):
    result = collatz.collatz(n)
    if result == expected:
        print("passed")
    else:
        print("failed for", n,
              "; expected", expected,
              "but got", result)

def run_tests():
    test_collatz(1, 1)
    test_collatz(2, 1)
```

output

```
passed
passed
```

Step 3A: Fail

- By definition, we assume if N is odd, `collatz(N)` should return 1

test_collatz.py

```
…
def run_tests():
  test_collatz(1, 1)
  test_collatz(2, 1)
  test_collatz(3, 1)
```

collatz.py

```
def collatz(n):
  if n == 1:
    return 1
  elif n % 2 == 0:
    return collatz(n // 2)
```

```
collatz(3) = None
```

output

```
passed
Passed
Failed for 3; expected 1 but got None
```

Step 3B: Pass

- By definition, we assume if N is odd, `collatz(N)` should return 1

test_collatz.py

```
…
def run_tests():
  test_collatz(1, 1)
  test_collatz(2, 1)
  test_collatz(3, 1)
```

collatz.py

```
def collatz(n):
  if n == 1:
    return 1
  elif n % 2 == 0:
    return collatz(n // 2)
  else:
    return collatz(3 * n + 1)
```

```
collatz(3) = collatz(10)
           = collatz(5)
           = collatz(16)
           = collatz(8)
           = collatz(4)
           = collatz(2)
           = collatz(1)
           = 1
```

output

```
passed
passed
passed
```

Step 3C: Refactor

- Can easily modify the test suite in run_tests() to test for more values of N

test_collatz.py

```
…
def run_tests():
    # test N from 1 to 10
    for n in range(1,11):
        test_collatz(n, 1)
```

collatz.py

```
def collatz(n):
  if n == 1:
    return 1
  elif n % 2 == 0:
    return collatz(n // 2)
  else:
    return collatz(3 * n + 1)
```

output

```
passed
passed
passed
passed
passed
passed
passed
passed
passed
passed
```

Finish Production Code

- The requirement of our production code is:

collatz.py

```
def main():
    n = int(input('Enter N: '))
    if n <= 0:
        print('N > 0')
    else:
        print('collatz(‘, n, ')=‘,
            collatz(n))
```

Prompt user for N

Handle an invalid N

Compute and display the result for N

console

```
$ python3 collatz.py
Enter N: 0
N > 0
$ python3 collatz.py
Enter N: 21
collatz( 21 ) = 1
```

Collatz Steps

$$collatz(N) = \begin{cases} 1: & \text{if N is 1} \\ collatz(N/2): & \text{if N is even} \\ collatz(3N+1): & \text{if N is odd} \end{cases}$$

- Recall the original problem was two-fold for collatz(N). We will tackle this problem in the following order:
  1. Count the total number of steps
  2. Determine the sequence

Determine the sequence that ends with 1 using the following starting values for N:

- 1 1 (1 STEP)
- 2 2 1 (2 STEPS)
- 10 10 5 16 8 4 2 1 (7 STEPS)
- 21 21 64 32 16 8 4 2 1 (8 STEPS)
- 13 13 40 20 10 5 16 8 4 2 1 (10 STEPS)

Collatz Steps: Production Code Stub

- We start with the same framework as the previous exercise:

collatz_steps.py

```
def collatz_steps(n):
  return None

def main():
  pass

if __name__ == "__main__”:
    main()
```

The sequence generator and step counter is implemented here

The production `main()` will be implemented after the test suite is complete and runs correctly

Collatz Steps: Test Suite Stub

test_collatz_steps.py

```
from collatz_steps import collatz_steps as cs



def run_tests():
    pass

if __name__ == "__main__”:
    run_tests()
```

This import allows us to call the `collatz_steps()` function in the `collatz_steps` module, simply as `cs()`

All unit tests of `collatz_steps()` go here

Step 1A Fail:

- `collatz_steps(1)` takes 1 step to converge at 1

test_collatz_steps.py

```
…
def run_tests():
    result = cs(1)
    if result == 1:
       print(“passed”)
     else:
        print("failed for", 1,
              "; expected", expected,
              "but got", result)
```

collatz_steps.py

```
def collatz_steps(n):
  return None
```

```
cs(1) = None
```

output

```
Failed for 1
; expected 1
but got None
```

Step 1B Pass:

- `collatz_steps(1)` takes **1** step to converge at **1**

`test_collatz_steps.py`

```
…
def run_tests():
    result = cs(1)
    if result == 1:
       print(“passed”)
     else:
        print("failed for", 1,
              "; expected", expected,
              "but got", result)
```

`collatz_steps.py`

```
def collatz_steps(n):
  if n == 1:
    return 1
```

```
cs(1) = 1
```

output

```
passed
```

Step 1C: Refactor

test_collatz_steps.py

```
def test_collatz_steps(name, n, expected):
    result = cs(n)
    if result == expected:
        print(name, "passed")
    else:
        print(name, "failed for", n,
              "; expected", expected,
              "but got", result)

def run_tests():
    test_collatz_steps(
        "collatz_steps(1)", 1, 1)
```

Pass N to `collatz_steps()`

Display results of a pass or fail case

Pass test name, N, and expected to `test_collatz_steps()`

output

```
collatz(1) passed
```

Step 2A: Fail

- `collatz_steps(2)` takes 2 steps to converge at 1

test_collatz_steps.py

```
…
def run_tests():
    …
    test_collatz_steps(
        "collatz_steps(2)", 2, 2)
```

collatz_steps.py

```
def collatz_steps(n):
  if n == 1:
    return 1
```

```
cs(2) = None
```

output

```
collatz(1) passed
Failed for 2
; expected 2
but got None
```

Step 2B: Pass

- `collatz_steps(2)` takes 2 steps to converge at 1

test_collatz_steps.py

```
…
def run_tests():
    …
    test_collatz_steps(
        "collatz_steps(2)", 2, 2)
```

collatz_steps.py

```
def collatz_steps(n):
  if n == 1:
    return 1
  elif n % 2 == 0:
    return 1 + collatz_steps(n // 2)
```

```
cs(2) = 1 + cs(1)
      = 1 + 1
      = 2
```

output

```
collatz(1) passed
collatz(2) passed
```

Step 2C: Refactor

- There is no need to refactor the code further, it is extensible to any number of test cases we devise.

Step 3B: Fail

- `collatz_steps(10)` takes 7 steps to converge at 1

test_collatz_steps.py

```
…
def run_tests():
    …
    test_collatz_steps(
        "collatz_steps(10)",
        10,
        7)
```

collatz_steps.py

```
def collatz_steps(n):
  if n == 1:
    return 1
  elif n % 2 == 0:
    return 1 + collatz_steps(n // 2)
```

```
cs(10) = 1 + cs(5)
       = 1 + None
```

output

```
collatz(1) passed
collatz(2) passed
error
```

Step 3B: Pass

output

```
collatz(1) passed
collatz(2) passed
collatz(10) passed
```

- `collatz_steps(10)` takes 7 steps to converge at 1

collatz_steps.py

```python
def collatz_steps(n):
  if n == 1:
    return 1
  elif n % 2 == 0:
    return 1 + collatz_steps(n // 2)
  else:
    return 1 + collatz_steps(3 * n + 1)
```

```
cs(10) = 1 + cs(5)
       = 1 + 1 + cs(16)
       = 1 + 1 + 1 + cs(8)
       = 1 + 1 + 1 + 1 + cs(4)
       = 1 + 1 + 1 + 1 + 1 + cs(2)
       = 1 + 1 + 1 + 1 + 1 + 1 + cs(1)
       = 1 + 1 + 1 + 1 + 1 + 1 + 1
       = 7
```

Collatz Sequence

- Finally, we will modify collatz_steps() so it can also print out the sequence of numbers as it is recursively computing the steps.

- To test this part, it must be done visually by the tester

- We will aid the tester and print out the expected sequence for each test in run_tests()

Collatz Sequence: run_tests()

test_collatz.py

```
…
def run_tests():
  print("1 (expected)")
  test_collatz_steps("\ncollatz_steps(1)", 1, 1)
  print("2 1 (expected)")
  test_collatz_steps("\ncollatz_steps(2)", 2, 2)
  print("10 5 16 8 4 2 1 (expected)")
  test_collatz_steps("\ncollatz_steps(10)", 10, 7)
  print("21 64 32 16 8 4 2 1 (expected)")
  test_collatz_steps("\ncollatz_steps(21)", 21, 8)
  print("13 40 20 10 5 16 8 4 2 1 (expected)")
  test_collatz_steps("\ncollatz_steps(13) ", 13, 10)
```

Collatz Sequence: collatz_steps()

collatz_steps.py

```
def collatz_steps(n):

  print(n, end=" ")

  if n == 1:
    return 1
  elif n % 2 == 0:
    return 1 + collatz_steps(n // 2)
  else:
    return 1 + collatz_steps(3 * n + 1)
```

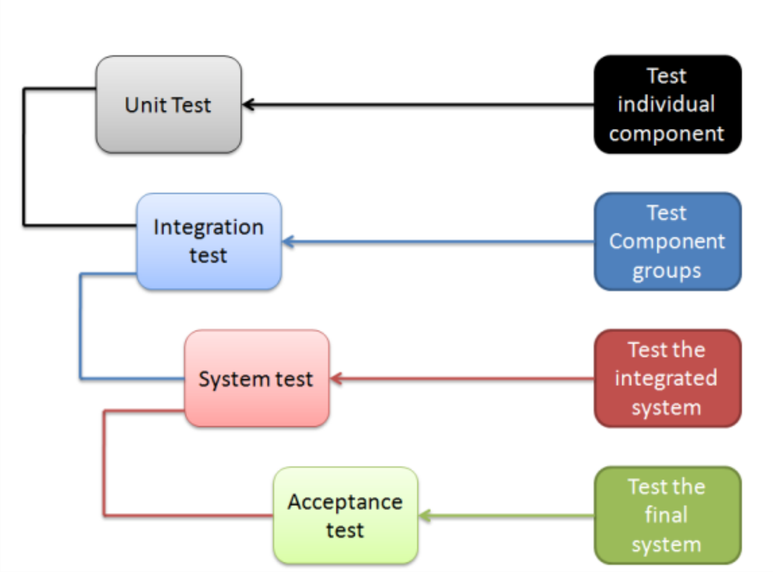
print current n

output

```
1 (expected)
1
collatz_steps(1) passed
2 1 (expected)
2 1
collatz_steps(2) passed
10 5 16 8 4 2 1 (expected)
10 5 16 8 4 2 1
collatz_steps(10) passed
21 64 32 16 8 4 2 1 (expected)
21 64 32 16 8 4 2 1
collatz_steps(21) passed
13 40 20 10 5 16 8 4 2 1 (expected)
13 40 20 10 5 16 8 4 2 1
collatz_steps(13) passed
```

Software Testing Levels

- There are many levels of testing that a software product goes through



Done by developers

Done by testers

Done by testers

Done by end users