# Software Testing

Juyoung Yang, Arogya Kharel
didwndud3299@kaist.ac.kr, akharel@kaist.ac.kr

Web Engineering and Service Computing Lab,
Korea Advanced Institute of Science and Technology (KAIST)

2025. 05. 28~29

KAIST

KAIST WEBENG

# Contents

KAIST

KAIST WEBENG

# Introduction to Software Testing

# Software Testing

**An investigation conducted to provide stakeholders with information about the quality of the software under test (SUT)**

- **Software quality attributes**
  - **Correctness (Functionality)** ← **We will focus on this !**

  The software behaves exactly as intended according to its specifications
  - **Performance**

  The software meets some performance-related expectations (execution time, network throughput, memory usage)
  - **Usability**

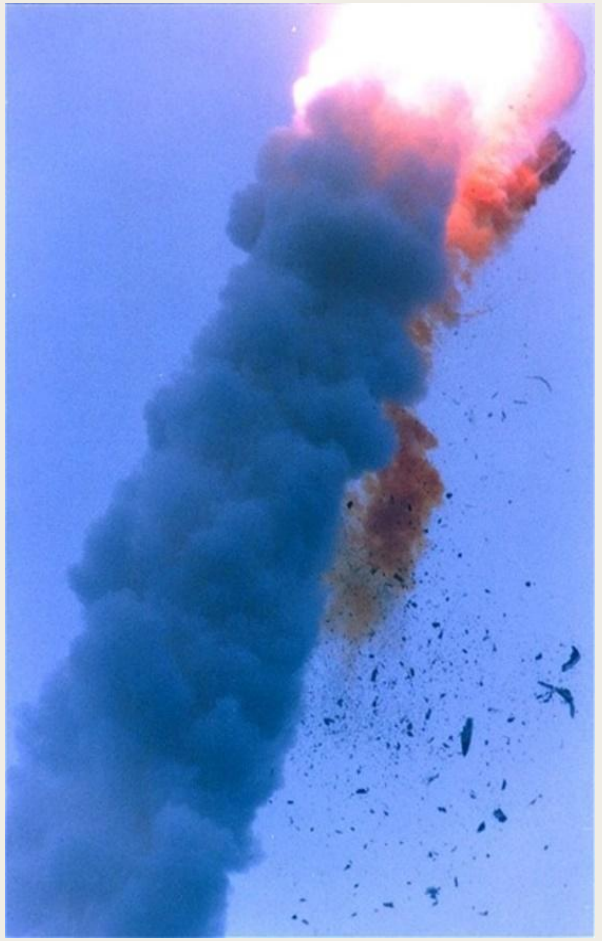
  The software is easy to learn and use (intuitive user interfaces)
  - **Reliability**

  The software performs well under various conditions and for a specific period

# Why is Software Testing Important?

## Even a small & trivial bug can cause big damage

- **Ariane 5**[1]
  - The Ariane 5 rocket, developed by the European Space Agency, exploded just 40 seconds after launch
  - The failure was caused by an **integer overflow**
  - This small error resulted in the loss of the rocket and cargo worth around 600 million euros





[1] https://www.cas.mcmaster.ca/~baber/TechnicalReports/Ariane5/Ariane5.htm

# Why is Software Testing Important?

## Even a small & trivial bug can cause big damage

- **Toyota Recall**[2]
  - 625,000 hybrid cars recalled in 2015 due to a software glitch in the braking system



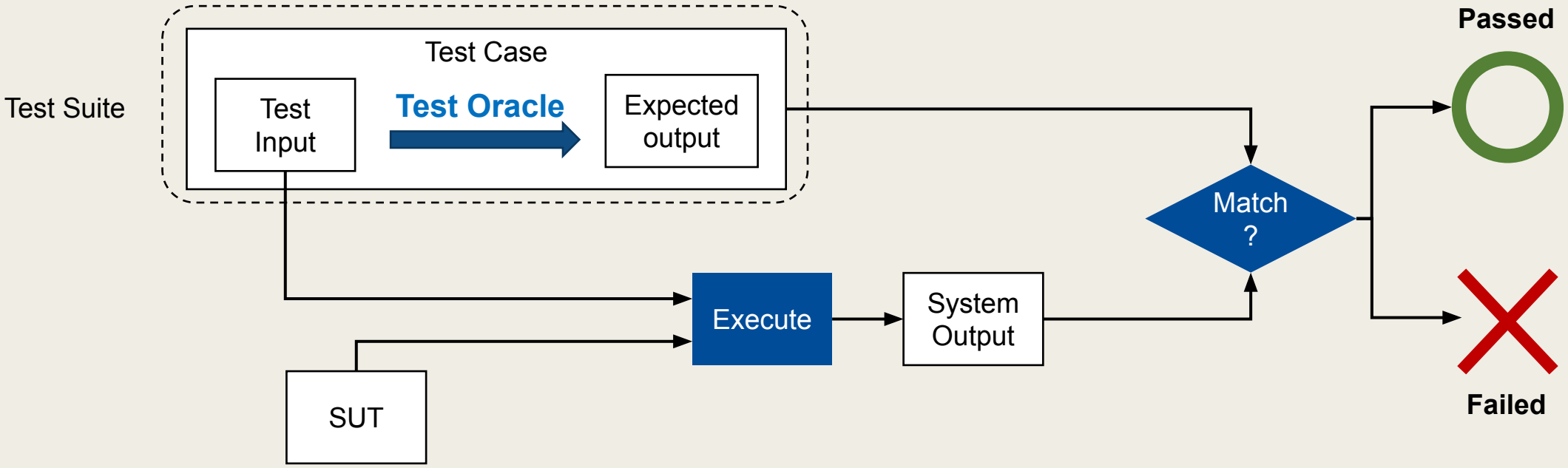

[2] https://www.bbc.com/news/business-33532673

# Software Testing Terminology

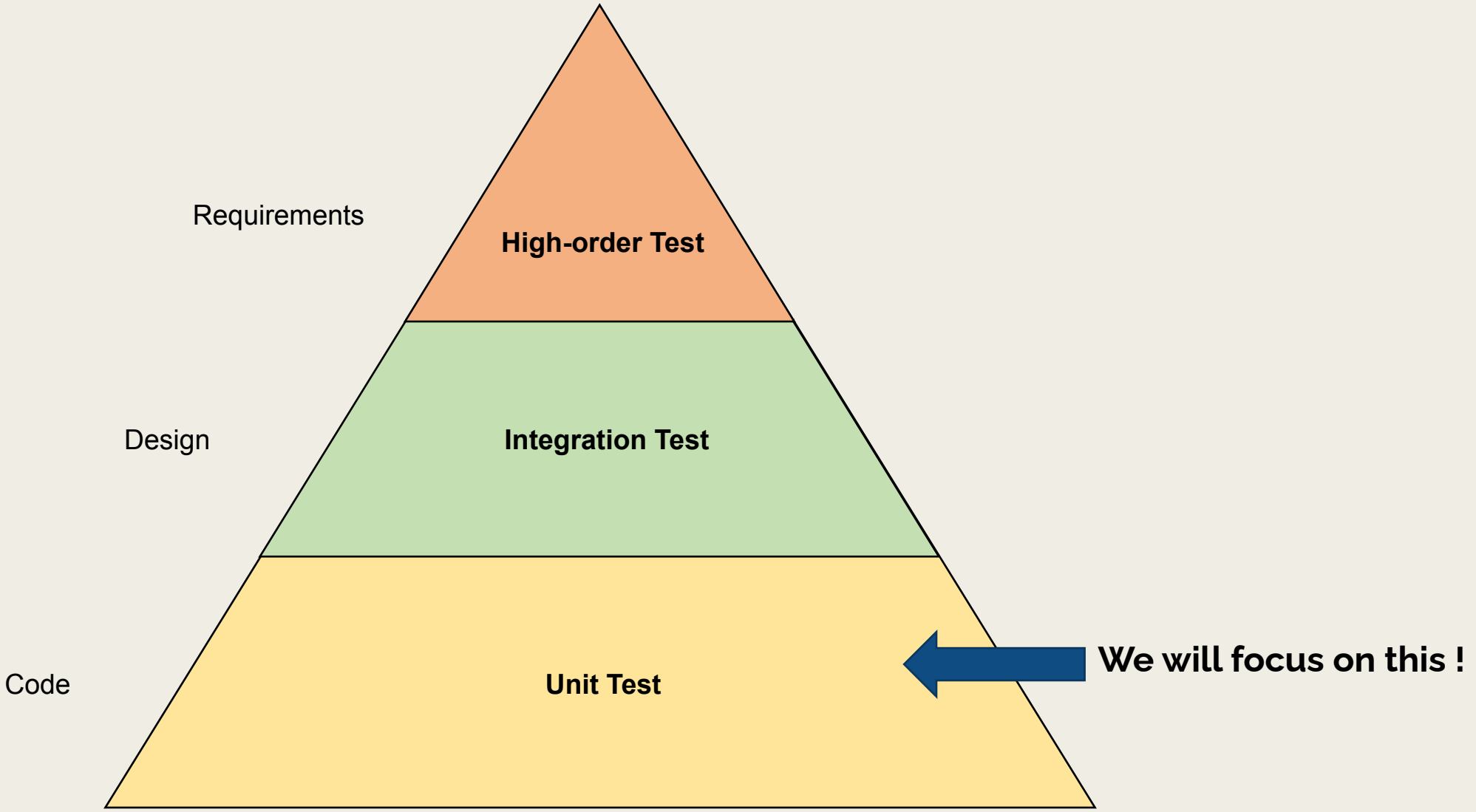
- **Test Input**: a set of input values that are used to execute the given program
- **Test Oracle**: a provider of the expected output based on the test input
- **Test Case:** test input + expected output
- **Test Suite:** a collection of test cases

# Process of Software Testing

**Check whether the system's output and the expected output are the same**



# Software Testing Hierarchy



KAIST

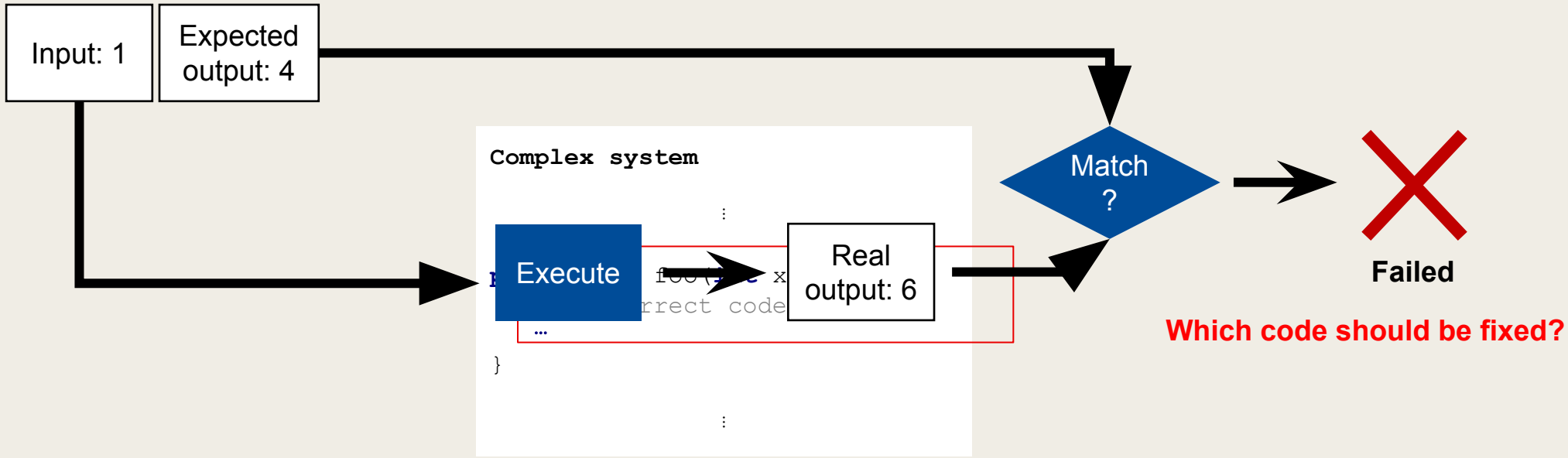
KAIST WEBENG

# Unit Testing (Python)

# Why Unit Test?

**Test each unit (function, class) of the software to check whether they satisfy the requirements**

- **Detect bugs early and ensure each unit works correctly in isolation**



# SUT in This Tutorial

- Basic requirements of a car

*A car tracks its driving time, distance, and speed.*
*A car can change its speed.*
*A car should be able to return its current speed and average speed.*

- Based on the requirements, the following code can be written

 car.py (See Appendix A)

```
class Car:
    def __init__(self, speed=0):
        self.speed = speed
        self.odometer = 0
        self.time = 0

    def change_speed(self, change):
        ...
    def get_current_speed(self):
        ...
    def average_speed(self):
       ...
```

# Writing and Running Test Cases

- Based on the SUT, unit test code can be written  test_car.py (Appendix B)
  - Specify test input to get the real output
  - Specify the expected test output
  - Compare the real output and expected output

```python
import pytest
import random
from car import Car


@pytest.fixture
def car():
    return Car()


def test_initial_speed(car):
    assert car.speed == 0
```

- Run test cases
  - `pytest <test_file>`
  - Ex) `pytest test_car.py`

```
$ pytest test_car.py
============================================== test session starts ==============================================
platform win32 -- Python 3.12.6, pytest-8.3.3, pluggy-1.5.0
rootdir: C:\Users\didwn\OneDrive\바탕 화면\testing_tutorial\cs350-testing-exercise
collected 7 items

test_car.py .......                                                                                    [100%]

============================================== 7 passed in 0.03s ==============================================
```

# Checking Test Coverage

- **Test coverage**
  - How much of the source code is executed by the test suite
    - Statement coverage
    - Branch coverage
  - High coverage generally leads to more reliable and robust software
  - However, **100% coverage does not guarantee 100% correctness**

  □ Coincidental correctness: quality of test cases also matters

- **Check test coverage**
  - `coverage run –branch –m pytest test_car.py`
  - `coverage report`

```
$ coverage report
Name          Stmts   Miss Branch BrPart  Cover
-----------------------------------------------
car.py           21      6      6      1    59%
test_car.py      28      0      4      0   100%
-----------------------------------------------
TOTAL            49      6     10      1    81%
```

# Tutorial Setting

- **Install Python 3.10.11**
  https://www.python.org/downloads/release/python-31011/

| | | | | | | |
|---|---|---|---|---|---|---|
| Windows installer (64-bit) | Windows | Recommended | a55e9c1e6421c84a4bd8b4be41492f51 | 27.7 MB | SIG | .sigstore |

- **Confirm git installation**
  - Open Powershell
  - Try `git version`

- **If error, install git**

  https://github.com/git-for-windows/git/releases/tag/v2.49.0.windows.1

| | |
|---|---|
| Git-2.49.0-64-bit.exe | 726056328967f242fe6e9afbfe7823903a928aff577dcf6f517f2fb6da6ce83c |

# Tutorial Setting

- Fork the tutorial repository from the URL below

https://github.com/kaist-webeng/cs350-testing-exercise.git

- Clone your forked repository
  - git clone https://github.com/<your ID>/cs350-testing-exercise.git

- Install the required python packages (required python version: 3.10)
  - pip install –r requirements.txt

# Challenges of Generating Test Cases

- **Exhaustive Testing**
  - Can we test all possible test inputs?

    Ex) Password Checker: checks 8-character passwords using only uppercase letters (A–Z)  There are $26^8$ = 208,827,064,576 possible combinations
  - All test inputs need to be executed & the result should be compared with expected result  **prohibitive cost**

- **Small & effective test suite**
  - Testing allows only a **sampling** of an enormously large program input space
  - The difficulty lies in how to come up with **effective sampling**

# Challenges of Generating Test Cases

- **Quality of test suite**
  - A high-quality test suite ensures **high coverage** and includes **edge cases** that can reveal the faults
  - However, **manually writing** such test cases is **time-consuming and error-prone**

- **Automated test generation tools**
  - Create **high-coverage, effective test cases** without human effort

KAIST

KAIST WEBENG

# Automated Test Case Generation

# What is Pynguin?

- **PYthoN General UnIt Test generator**[1, 2]
  - A tool that allows developers to generate unit tests automatically.

- **Mature tools exist for statically typed languages like Java**
  - No comprehensive tool existed for **dynamically typed** languages like Python

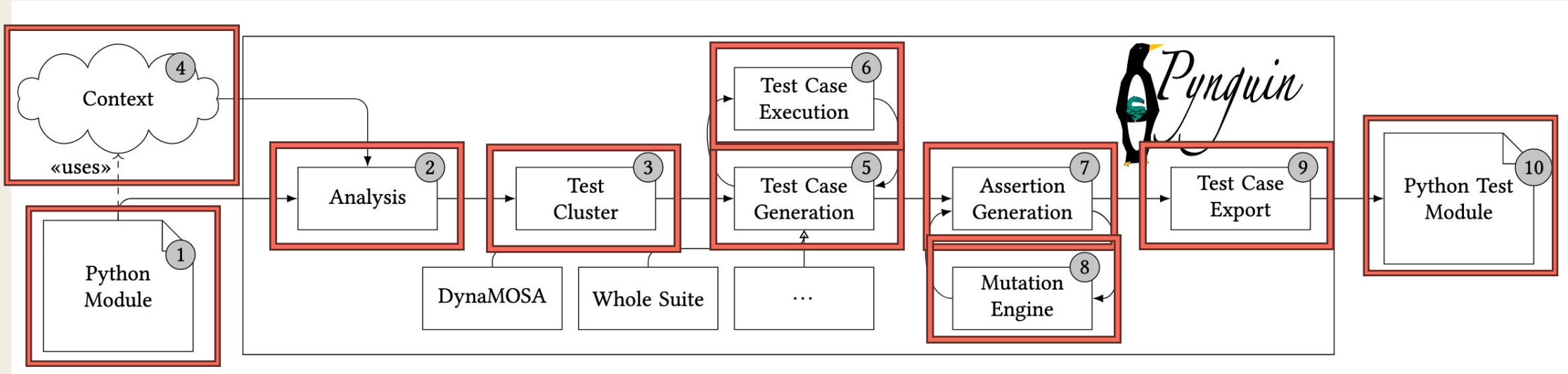
- **Pynguin steps in as the first tool to enable fully-automated generation**




[1] https://github.com/se2p/pynguin

[2] Lukasczyk, S. and Fraser, G. 2022. Pynguin: automated unit test generation for Python. *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings* (New York, NY, USA, Oct. 2022), 168–172.

# Execution Steps of Pynguin



# Going Back Inside Our Car

- **Pynguin requires setting env var[3]:**

```
export PYNGUIN_DANGER_AWARE=x
```

- **Provide pynguin with source and output paths:**

```
$ pynguin --project-path .
--output-path pynguin-results
--module-name car
```

- **Check your generated test file!**

```python
class Car:
    def __init__(self, speed=0):
        self.speed = speed
        self.odometer = 0
        self.time = 0

    def change_speed(self, change):
        ...
    def get_current_speed(self):
        ...
    def average_speed(self):
        ...
```

```
$ coverage report
Name          Stmts   Miss Branch BrPart  Cover
-----------------------------------------------
car.py           21      6      6      1    59%
test_car.py      28      0      4      0   100%
-----------------------------------------------
TOTAL            49      6     10      1    81%
```



[3] https://pynguin.readthedocs.io/en/latest/user/quickstart.html

Test Generation

# Pynguin Results

```python
# Test cases automatically generated by Pynguin (https://www.pynguin.eu).
# Please check them before you use them.
import pytest
import car as module_0


@pytest.mark.xfail(strict=True)
def test_case_0():
    bool_0 = False
    car_0 = module_0.Car()
    assert car_0.speed == 0
    assert car_0.odometer == 0
    assert car_0.time == 0
    var_0 = car_0.change_speed(bool_0)
    assert car_0.time == 1
    var_0.step()


def test_case_1():
    none_type_0 = None
    car_0 = module_0.Car(none_type_0)
    assert car_0.odometer == 0
    assert car_0.time == 0
    with pytest.raises(Exception):
        car_0.average_speed()


def test_case_2():
    car_0 = module_0.Car()
    assert car_0.speed == 0
    assert car_0.odometer == 0
    assert car_0.time == 0
```

The test is expected to fail!

```
$ coverage report
Name          Stmts   Miss Branch BrPart  Cover
-----------------------------------------------
car.py           21      6      6      1    59%
test_car.py      28      0      4      0   100%
-----------------------------------------------
TOTAL            49      6     10      1    81%
```

VS

```
Name          Stmts   Miss Branch BrPart  Cover
-----------------------------------------------
car.py           21      0      6      0   100%
test_car.py      75      0      0      0   100%
-----------------------------------------------
TOTAL            96      0      6      0   100%
```

KAIST

KAIST WEBENG

# Automated Continuous Integration

# Automated Continuous Integration

- **Continuous Integration (CI)**: a development practice where developers frequently integrate code into a shared repository

- **Automated CI** enables that code changes are automatically built, tested, and validated each time they are committed

 It is essential for fast and error-free integration of the code

- **CI tools**
  - GitHub Actions

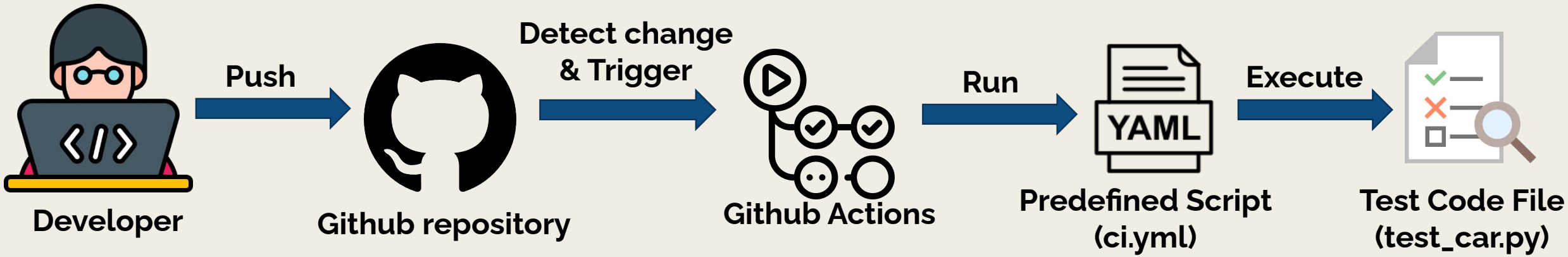
Actions

  - Jenkins

# Github Action

- When a developer pushes the changed code, GitHub Actions automatically run a predefined script that executes test code

Our predefined script: ci.yml (See Appendix D)



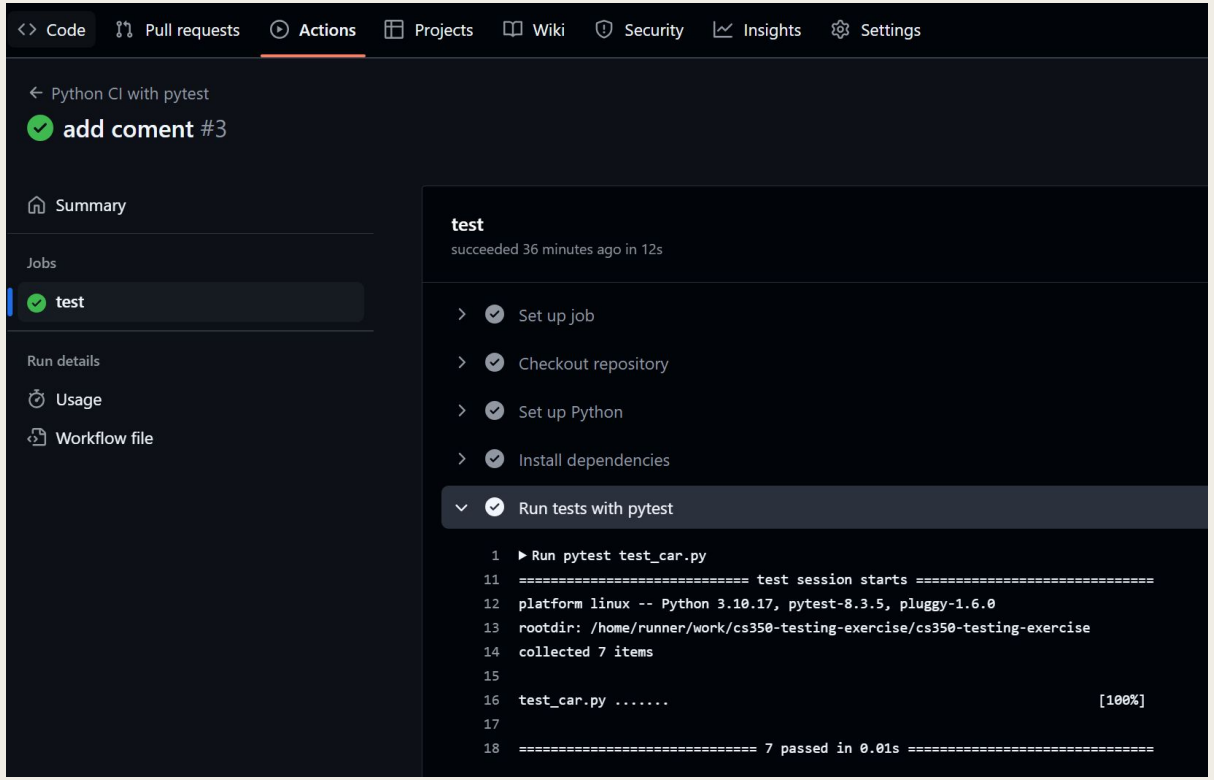
CI

# Github Action

- Add your CI script (ci.yml) to below path

**<root directory>/.github/workflows**

- Change your source code (car.py)
- Push your new commit
- Check your predefined script for CI is automatically executed




KAIST WEBENG

# Thank you

Juyoung Yang, Arogya Kharel
didwndud3299@kaist.ac.kr, akharel@kaist.ac.kr

Web Engineering and Service Computing Lab,
Korea Advanced Institute of Science and Technology (KAIST)

2025. 05. 28~29

KAIST

KAIST WEBENG

# Appendix

A. Source Code – car.py

B. Sample Test Code – test_car.py

C. Source Code - lift.py

D. Sample Test Code - test_lift.py

C. Generated Test Code – pynguin-results/test_car.py

Appendix A

# Source Code – car.py

```
class Car:
    def __init__(self, speed=0):
        self.speed = speed
        self.odometer = 0
        self.time = 0

    def change_speed(self, change):
        if change < 0:
            if self.speed + change < 0:
                self.speed = 0
            else:
                self.speed += change
        else:
            self.speed += change

        self.step()

    def get_current_speed(self):
        return self.speed

    def step(self):
        self.odometer += self.speed
        self.time += 1

    def average_speed(self):
        if self.time == 0:
            raise Exception("Divide by 0! Car did not move!")
        else:
            return self.odometer / self.time
```

Appendix B

# Sample Test Code – test_car.py

```python
import pytest
import random
from car import Car


@pytest.fixture
def car():
    return Car()


def test_initial_speed(car):
    assert car.speed == 0


def test_initial_odometer(car):
    assert car.odometer == 0


def test_initial_time(car):
    assert car.time == 0
```

```python
# Tests for changing speed
def test_change_speed(car):
    car.change_speed(7)
    assert car.speed == 7


def test_multiple_speed_change(car):
    total_change = 0
    random.seed(42)
    for _ in range(3):
        change = random.randint(0, 10)
        total_change += change
        car.change_speed(change)
    assert car.speed == total_change


# Tests for getting current speed
def test_get_current_speed_from_start(car):
    assert car.get_current_speed() == 0


def test_get_current_speed_after_change(car):
    car.change_speed(27)
    assert car.get_current_speed() == 27
```

Appendix C

# Sample Test Code – lift.py

```
class Lift:
    def __init__(self, highest_floor, max_riders =10):
        self.top_floor = highest_floor
        self.current_floor = 0
        self.capacity = max_riders
        self.num_riders = 0

    def get_top_floor(self):
        return self.top_floor

    def get_current_floor(self):
        return self.current_floor

    def get_capacity(self):
        return self.capacity

    def get_num_riders(self):
        return self.num_riders

    def is_full(self):
        return self.num_riders == self.capacity
```

```
    def add_riders(self, num_entering):
        if self.num_riders + num_entering <= self.capacity:
            self.num_riders += num_entering
        else:
            self.num_riders = self.capacity

    def go_up(self):
        if self.current_floor < self.top_floor:
            self.current_floor += 1

    def go_down(self):
        if self.current_floor > 0:
            self.current_floor -= 1

    def call(self, floor):
        if 0 <= floor <= self.top_floor:
            while self.current_floor != floor:
                if floor > self.current_floor:
                    self.go_up()
                else:
                    self.go_down()
```

Appendix D

# Sample Test Code – test_lift.py

```
import pytest
from lift import Lift

def test_initial_values ():
    lift = Lift(5)
    assert lift.get_top_floor() == 5
    assert lift.get_current_floor() == 0
    assert lift.get_num_riders() == 0
    assert not lift.is_full()

def test_add_riders_not_full ():
    lift = Lift(5)
    lift.add_riders( 5)
    assert lift.get_num_riders() == 5
    assert not lift.is_full()


def test_go_up ():
    lift = Lift(3)
    lift.go_up()
    lift.go_up()
    lift.go_up()   # should be at top
    lift.go_up()   # should stay at top
    assert lift.get_current_floor() == 3
```

# Appendix E Generated Test Code – pynguin-results/test_car.py

```python
# Test cases automatically generated by Pynguin
(https://www.pynguin.eu).
# Please check them before you use them.
import pytest
import car as module_0


def test_case_0():
    float_0 = -13.59933065467643
    car_0 = module_0.Car()
    assert car_0.speed == 0
    assert car_0.odometer == 0
    assert car_0.time == 0
    var_0 = car_0.change_speed(float_0)
    assert car_0.time == 1


@pytest.mark.xfail(strict=True)
def test_case_1():
    bool_0 = False
    car_0 = module_0.Car()
    assert car_0.speed == 0
    assert car_0.odometer == 0
    assert car_0.time == 0
    var_0 = car_0.change_speed(bool_0)
    assert car_0.time == 1
    var_0.step()


def test_case_2():
    float_0 = -35.84348955890579
    car_0 = module_0.Car()
    assert car_0.speed == 0
    assert car_0.odometer == 0
    assert car_0.time == 0
    var_0 = car_0.change_speed(float_0)
    assert car_0.time == 1
    car_1 = module_0.Car()
    assert car_1.speed == 0
    assert car_1.odometer == 0
    assert car_1.time == 0
    car_2 = module_0.Car()
    assert car_2.speed == 0
    assert car_2.odometer == 0
    assert car_2.time == 0
    with pytest.raises(Exception):
        car_2.average_speed()
```

```python
@pytest.mark.xfail(strict=True)
def test_case_3():
    float_0 = -12.76562758095508
    car_0 = module_0.Car()
    assert car_0.speed == 0
    assert car_0.odometer == 0
    assert car_0.time == 0
    var_0 = car_0.change_speed(float_0)
    assert car_0.time == 1
    car_1 = module_0.Car()
    assert car_1.speed == 0
    assert car_1.odometer == 0
    assert car_1.time == 0
    var_1 = car_0.average_speed()
    assert var_1 == pytest.approx(0.0, abs=0.01, rel=0.01)
    var_0.change_speed(car_1)


def test_case_4():
    car_0 = module_0.Car()
    assert car_0.speed == 0
    assert car_0.odometer == 0
    assert car_0.time == 0


@pytest.mark.xfail(strict=True)
def test_case_5():
    car_0 = module_0.Car()
    assert car_0.speed == 0
    assert car_0.odometer == 0
    assert car_0.time == 0
    var_0 = car_0.step()
    assert car_0.time == 1
    var_1 = car_0.get_current_speed()
    assert var_1 == 0
    car_0.change_speed(car_0)
```

```python
@pytest.mark.xfail(strict=True)
def test_case_6():
    car_0 = module_0.Car()
    assert car_0.speed == 0
    assert car_0.odometer == 0
    assert car_0.time == 0
    var_0 = car_0.step()
    assert car_0.time == 1
    var_0.step()


@pytest.mark.xfail(strict=True)
def test_case_7():
    float_0 = -0.23206040601398698
    car_0 = module_0.Car()
    assert car_0.speed == 0
    assert car_0.odometer == 0
    assert car_0.time == 0
    var_0 = car_0.change_speed(float_0)
    assert car_0.time == 1
    bool_0 = True
    var_1 = car_0.change_speed(bool_0)
    assert car_0.speed == 1
    assert car_0.odometer == 1
    assert car_0.time == 2
    var_2 = car_0.get_current_speed()
    assert var_2 == 1
    var_3 = car_0.change_speed(float_0)
    assert car_0.speed == pytest.approx(0.767939593986013, abs=0.01, rel=0.01)
    assert car_0.odometer == pytest.approx(1.767939593986013, abs=0.01, rel=0.01)
    assert car_0.time == 3
    car_0.change_speed(var_1)
```

KAIST WEBENG

KAIST

KAIST WEBENG

# EOF