• Some software bug examples
• Some software problems
• Testing Introduction
• Types of testing
  • Manual and Automated Testing
• Continuous Testing
• Understanding different testing types
  • Functional and non-functional testing
  • {Black, White, Gray} Box Testing
  • Alpha, Pilot, Beta Testing
  • Unit Testing
• Testing in GitLab CI
• code coverage
• Code quality
• Summary
Some software bug examples

• **Gangnam Style music video ‘broke’ YouTube**
  • No. of views exceeds 2,147,483,647 (maximum for 32-bit integrer)

• **Thousands of duplicate vaccination appointments in New Jersey hospital [src]**
  • 10,000 - 11,000 duplicate appointments due to **Software bug**
  • In some cases, a single person had more than 20 vaccine appointments

• **The year 2038 problem**
  • On 03:14:07 UTC on January 19: 32-bit computers won’t be able to tell the difference between the year 2038 and 1970

• The Millennium bug or Y2K
Some software bug examples

Tesla’s Cybertruck broken-window


Img src: https://www.washingtonpost.com/business/2019/11/25/elon-musk-teslas-cybertruck-windows-would-have-been-unbreakable-if-not-sledgehammer/
Some software problems:

• Incorrect calculations –
  • E.g. recall duration calculation in last practice session (Lab08)

• Incorrect data edits –
  • Can a date be Feb 30?

• Ineffective data edits
  • Failed to prevent incorrect input data.
  • An example of this is an alphanumeric address field that allows spaces to be entered before any numbers or letters in the address. Therefore, when searches or sorts are performed on the address field, the search or sort may not find the intended address.
Some software problems:

- **Incorrect implementation of business rules**
  - Can be due to incorrect, missing, or vague system requirements specifications

- **Inadequate software performance**
  - Refers to slow system response times and transaction throughput rates.

- **Confusing or misleading data**
  - Hard to understand the shown data for users

- **Software that is difficult to use**

- **Obsolete software**
  - Due to new hardware or support software changes
Some software problems:

- Inconsistent processing
  - Work correctly in specific platform/OS
- Difficult to maintain and understand
- Unreliable results or performance
- Inadequate support of business needs or objectives
- No longer supported by the vendor
- Incorrect or inadequate interfaces with other systems
Some software problems:

- Incorrect matching and merging of data
- Data searches that yield incorrect results
- Incorrect processing of data
- Incorrect file and data handling
- Inadequate security controls
- Inability to handle production data capacities

But why all these problems occur....
Testing Introduction

• What is testing:
  • Method to check whether the software is working as per the requirement
  • And to ensure it is defect free

• Why?
  • To find error
  • To find security flaw
  • To find if it is usable
  • To find if it works in all circumstances
  • And many more…
Testing Introduction

Benefits:

- Better product quality
- Higher QoS
- Better security
- Cost-effective
  - E.g. In April of 1999, a software bug caused the failure of a $1.2 billion military satellite launch, the costliest accident in history [src: guru99]
  - Fixing bug after release cost you more
Types of testing

- Alpha Testing
- Acceptance Testing
- Ad-hoc Testing
- Accessibility Testing
- Beta Testing
- Back-end Testing
- Browser Compatibility Testing
- Backward Compatibility Testing
- Black Box Testing
- Boundary Value Testing
- Branch Testing
- Comparison Testing
- Compatibility Testing
- Component Testing
- End-to-End Testing
- Equivalence Partitioning
- Example Testing
- Exploratory Testing
- Functional Testing
- Graphical User Interface (GUI) Testing
- Gorilla Testing
- Happy Path Testing
- Incremental Integration Testing
- Install/Uninstall Testing
- Integration Testing
- Load Testing
- Monkey Testing
- Mutation Testing
- Negative Testing
- Non-Functional Testing
- Performance Testing
- Recovery Testing
- Regression Testing
- Risk-Based Testing (RBT)
- Sanity Testing
- Security Testing
- Smoke Testing
- Static Testing
- Stress Testing
- System Testing
- Unit Testing
- Usability Testing
- Vulnerability Testing
- Volume Testing
- White Box Testing
Types of testing

Hold on….
Before going to understand these types…

- Alpha Testing
- Acceptance Testing
- Ad-hoc Testing
- Accessibility Testing
- Beta Testing
- Back-end Testing
- Browser Compatibility Testing
- Backward Compatibility Testing
- Black Box Testing
- Boundary Value Testing
- Branch Testing
- Comparison Testing
- Compatibility Testing
- Component Testing
- End-to-End Testing
- Equivalence Partitioning
- Example Testing
- Exploratory Testing
- Functional Testing
- Graphical User Interface (GUI) Testing
- Gorilla Testing
- Incremental Integration Testing
- Install/Uninstall Testing
- Integration Testing
- Load Testing
- Monkey Testing
- Mutation Testing
- Negative Testing
- Non-Functional Testing
- Performance Testing
- Recovery Testing
- Regression Testing
- Risk-Based Testing (RBT)
- Sanity Testing
- Security Testing
- Static Testing
- Stress Testing
- System Testing
- Unit Testing
- Usability Testing
- Vulnerability Testing
- Volume Testing
- White Box Testing
Types of testing

Hold on....
Before going to understand these types...

Need to know how to perform these testing
Ways to perform testing

- Alpha Testing
- Acceptance Testing
- Ad-hoc Testing
- Accessibility Testing
- Benchmark Testing

- Manual testing
- Browser Compatibility Testing
- Backward Compatibility Testing
- Black Box Testing
- Boundary Value Testing
- Branch Testing
- Comparison Testing
- Compatibility Testing
- Component Testing
- End-to-End Testing
- Equivalence Partitioning

- Example Testing
- Exploratory Testing
- Functional Testing
- Graphical User Interface (GUI) Testing
- Gorilla Testing
- Happy Path Testing
- Incremental Integration Testing
- Install/Uninstall Testing
- Integration Testing
- Load Testing
- Monkey Testing
- Mutation Testing
- Negative Testing
- Non-Functional Testing

- Beta Testing
- Back-end Testing
- Browser Compatibility Testing
- Backward Compatibility Testing
- Black Box Testing
- Boundary Value Testing
- Branch Testing
- Comparison Testing
- Compatibility Testing
- Component Testing
- End-to-End Testing
- Equivalence Partitioning

- Performance Testing
- Recovery Testing
- Regression Testing
- Risk-Based Testing (RBT)
- Sanity Testing
- Security Testing
- Smoke Testing
- Static Testing
- Stress Testing
- System Testing
- Unit Testing
- Usability Testing
- Vulnerability Testing
- Volume Testing
- White Box Testing
Manual vs Automated Testing

**Manual Testing**
- Testing team executes test one-by-one in an individual manner
- Human tester perform testing
- Difficult to ensure sufficient test coverage
- Takes time
- Very inefficient
- Costly

**Automated Testing**
- Can be in parallel manner
- A software script does all testing
- Greater test coverage with large test cases
- Save time
- Efficient
- Cost effective
Continuous Testing

Manual Testing

- Testing team executes test one-by-one in an individual basis
- Human tester perform testing
- Difficult to ensure sufficient test coverage
- Takes time
- Very inefficient
- Costly

Are Continuous Testing and Automated Testing same?

Automated Testing

- Can be in parallel manner
- A software script does all testing
- Greater test coverage with large test cases
- Save time
- Efficient
- Cost effective
Continuous Testing and Test Automation

<table>
<thead>
<tr>
<th>Continuous Testing</th>
<th>Test Automation</th>
</tr>
</thead>
<tbody>
<tr>
<td>on achieving continuous quality &amp; improvement.</td>
<td>automating a test tasks.</td>
</tr>
<tr>
<td>The continuous testing process helps to find the risk, address them and improve the quality of the product.</td>
<td>A set of similar or repetitive tasks, a machine can execute, faster, with a fewer mistake.</td>
</tr>
<tr>
<td>Continuous testing can not be implemented without test automation.</td>
<td>Automation in testing possible without integrating continuous testing.</td>
</tr>
<tr>
<td>Software release may be released weekly to hourly.</td>
<td>Software release can take a month to years.</td>
</tr>
<tr>
<td>Feedback at each stage needs to be instant.</td>
<td>Regular feedback after testing each release.</td>
</tr>
</tbody>
</table>
Continuous testing

• Introduced to reduce the time taken to get feedback to developers.
• Test the unified codebase in its entire lifecycle
• Involves many automated tests
• To make automated tests → create Test Environment
• Create synthetic production data to create test data bed
• Make the testing parallel
How Continuous testing works

- Testing at every stage of the development life cycle.
- A software change continually moves
  - from development -->> testing -->> deployment.
- Evaluate the quality of the software as part of a continuous delivery process
  - Early and often testing

Implement the new feature ➔ Push the code (e.g. to GitLab) ➔ Code quality analysis ➔ Build the code ➔ Deploy to Dev/testing Environment ➔ Run testing automation script against latest code ➔ Report testing result possible to QA team
Continuous testing Benefits

• Faster release
• Better code quality
• Improve test coverage
• Avoid potential errors/bugs
  • Reduces human generated error
• Test even a minor change
• Save time and money
  • Fast testing
• Reusability
• Consistency
  • Same testing procedure for all similar commits
• Reliable
• Accuracy
• Enable Seamless integration in DevOps process
Continuous Testing Adoption (1/2)

Some of the things to keep in mind while adopting continuous testing:

• Environment virtualization
  • This gives Isolation
  • Ability to test from the start always
  • Enable concurrency

• Management of test data
  • Anonymization of data, Synthetic data
  • Data for both pass and fail scenarios

• Test automation
  • Script to make the testing automatic
Continuous Testing Adoption (2/2)

Some of the things to keep in mind while adopting continuous testing:

• API Testing
  • You already know the strength of APIs

• Performance Testing
  • Application scaling up/down
  • Load balancer behavior
  • Need to use external tools that generate loads

• Security Testing
  • Make sure that, while testing you are not exposing sensitive information
  • E.g. recall $gitlabpassword variable from previous practice session (Lab08)

• Automated Test Generation
  • Generation of data, user profiles, scenarios, environments, system configurations...
Types of testing

- Alpha Testing
- Acceptance Testing
- Ad-hoc Testing
- Adequate Testing
- Beta Testing
- Back-end Testing
- Browser Compatibility Testing
- Backward Compatibility Testing
- Black Box Testing
- Branch Testing
- Comparison Testing
- Compatibility Testing
- Component Testing
- End-to-End Testing
- Equivalence Partitioning
- Example Testing
- Exploratory Testing
- Functional Testing
- Graphical User Interface (GUI) Testing
- Happy Path Testing
- Incremental Integration Testing
- Install/Uninstall Testing
- Load Testing
- Monkey Testing
- Mutation Testing
- Negative Testing
- Non-Functional Testing
- Performance Testing
- Recovery Testing
- Regression Testing
- Risk-Based Testing (RBT)
- Sanity Testing
- Safety-Related Testing
- Smoke Testing
- Static Testing
- Stress Testing
- System Testing
- Unit Testing
- Usability Testing
- Vulnerability Testing
- Volume Testing
- White Box Testing

Hold on…

Before going to understand these types…

Lets understand some of these testing types…
Functional testing

• Test every aspect of the software piece
  • Against the functional requirement/specification
  • E.g. User interfaces, API, Databases, security, Client-server communication

• This involves:
  • Unit testing:
    • test each unit or function or module of the application.
  • Usability test:
    • e.g. can you really use modern application on old hardware?
  • Error conditions:
    • Right message when there is an error from user side.
  • Accessibility test:
    • E.g. You should not have the access to others users classified information.
  • Integration Testing
    • Definitely you need to go for this, when you have microservice-based application
Functional testing

• This involves:
  • Unit testing:
  • Usability test:
  • Error conditions:
  • Accessibility test:
  • Integration Testing
    • Definitely you need to go for this, when you have microservice-based application
  • Regression Testing:
    • Test again the old features (maybe not all) that are affected by the new feature
  • User Acceptance Testing:
    • the last step before software goes live
    • Test if the software can really handle large no. of users
    • Application may go for a beta testing period
Non-Functional testing

• Testing beyond the functional requirements
• Minimal requirement
  • the application went through functional testing
• E.g. No of transactions your application can handle
• What you may test under non-functional testing
  • Speed
  • Scalability
  • Load
  • Compatibility
  • Installation testing
  • Security testing
  • maintainability, portability, observability
  • Reliability, Disaster Recovery Testing
Non-Functional testing

• What you may test under non-functional testing
  • Speed
  • Scalability
  • Load
  • Compatibility
  • Installation testing
  • Security testing
  • maintainability, portability, observability
• Reliability, Disaster Recovery Testing
• Comliance
• Localization Testing
Black, White, Gray} Box Testing

How would you adopt functional testing?

• **White-box testing**
  • tester is *aware* of internal functionality
  • internal structure of an item or function to be tested is *unknown*.

• **Black-box testing**
  • tester is *unaware* of internal functionality
  • internal structure of an item or function to be tested is *unknown*.

• **Gray-Box testing**
  • combination of a Black and White box testing
  • Partial knowledge on internal functionality and internal structure
Alpha, Pilot, Beta Testing

**Alpha Testing**
- a type of validation testing
- done before the product is released to customers
- typically done by QA people

**Pilot Testing**
- Done by selected number of users (not to be confused with the beta user)
- Done before beta testing
- In a real environment

**Beta Testing**
- test is conducted at one or more customer sites
- Offered for a limited number of end-users
- Give a real-time feedback
Acceptance testing

• Performed by the client
• Also called User Acceptance Testing (UAT)
• verifies end to end flow of the system as per the business requirements
• one of the last phases of the testing
• A type of block-box testing
  • User just verify the features and functionality
  • No design/implementation knowledge required
Some other testing types

• Integration testing
  • Check if the integrated units (tested) are working without errors
  • relevant to the client/server and distributed systems
  • E.g. payment module, third-party payment, shopping cart module, database module, recommendation module, etc. should be tested before testing their integration

• System Testing
  • \textit{testing the system as a whole}
  • \textit{After integration testing}
Unit Testing

• a software testing method by which individual units of source code, such as functions, methods, and classes are tested

• Test each unit or function.

• We all do this using a print statement (but not in enterprise softwares)

• Usually done by the programmer
  • Less done by Tester

• Need detailed knowledge on that specific unit/module
Unit Testing tools??

Some tools for unit testing

• Unittest
• pytest
• Jtest
• JUnit
• NUnit
• JMockit
• EMMA
• PHP Unit
Unit Testing examples

Example of Python’s *unittest*

```python
mainfile.py

def add(a, b):
    return a + b

def sub(a, b):
    return a - b

def multiply(a, b):
    return a * b

import unittest
from mainfile import *

class test_main(unittest.TestCase):
    def test_add(self):
        self.assertEqual(add(3, 2), 5)
    def test_sub(self):
        self.assertEqual(sub(3, 2), 1)
    def test_multiply(self):
        self.assertEqual(multiply(3, 2), 6)

if __name__ == '__main__':
    unittest.main()
```
Unit Testing examples

Example of Python’s `unittest`

```
# mainfile.py

def add(a, b):
    return a + b

def sub(a, b):
    return a - b

def multiply(a, b):
    return a * b

if __name__ == '__main__':
    unittest.main()
```

```
# test_main.py

import unittest
from mainfile import *

class test_main(unittest.TestCase):
    def test_add(self):
        self.assertEqual(add(3, 2), 5)
    def test_sub(self):
        self.assertEqual(sub(3, 2), 1)
    def test_multiply(self):
        self.assertEqual(multiply(3, 2), 6)

if __name__ == '__main__':
    unittest.main()
```

```
ubuntu@gitlab-lab09-chinmaya:~/unittest$ python3 test_main.py
Ran 3 tests in 0.000s
OK
```
Unit Testing examples

Example of Python’s *unittest*

mainfile.py

```python
def add(a, b):
    return a + b

def sub(a, b):
    return a - b

def multiply(a, b):
    return a * b

def power(a, b):
    return a * b
```

Test function

```python
import unittest

from mainfile import *

class test_main(unittest.TestCase):
    def test_add(self):
        self.assertEqual(add(3, 2), 5)

    def test_sub(self):
        self.assertEqual(sub(3, 2), 1)

    def test_multiply(self):
        self.assertEqual(multiply(3, 2), 6)

    def test_power(self):
        self.assertEqual(power(3, 2), 9)

if __name__ == '__main__':
    unittest.main()
```

A new function with error
Unit Testing examples

Example of Python’s *unittest*

```python
class test_main(unittest.TestCase):
    def test_add(self):
        self.assertEqual(add(3, 2), 5)
    def test_sub(self):
        self.assertEqual(sub(3, 2), 1)
    def test_multiply(self):
        self.assertEqual(multiply(3, 2), 6)
    def test_power(self):
        self.assertEqual(power(3, 2), 9)
```

**Expected Value**

- `add(3, 2)` expected: 5
- `sub(3, 2)` expected: 1
- `multiply(3, 2)` expected: 6
- `power(3, 2)` expected: 9

**Actual Value**

- `add(3, 2)` actual: 5
- `sub(3, 2)` actual: 1
- `multiply(3, 2)` actual: 6
- `power(3, 2)` actual: 9

**A new function with error**

```python
def power(a, b):
    return a**b
```

**One failed**

```
FAIL: test_power (__main__.test_main)
```

**Expected Value**

```
AssertionError: 6 != 9
```

**Actual Value**

```
Ran 4 tests in 0.001s
FAILED (failures=1)
```
Unit Testing examples

Example of Python’s *unittest*

**mainfile.py**
```python
def add(a, b):
    return a + b

def sub(a, b):
    return a - b

def multiply(a, b):
    return a * b

def power(a, b):
    return a ** b
```

**Test function**
```
import unittest
from mainfile import *

class test_main(unittest.TestCase):
    def test_add(self):
        self.assertEqual(add(3, 2), 5)
    def test_sub(self):
        self.assertEqual(sub(3, 2), 1)
    def test_multiply(self):
        self.assertEqual(multiply(3, 2), 6)
    def test_power(self):
        self.assertEqual(power(3, 2), 9)
```

**Expected Value**
```
5
1
6
```

**Actual Value**
```
FAIL: test_power (__main__.test_main)
Traceback (most recent call last):
  File "test_main.py", line 11, in test_power
    self.assertEqual(power(3, 2), 6)
AssertionError: 9 != 6
```

**NO ERROR**
```
FAIL: test_power (__main__.test_main)
Traceback (most recent call last):
  File "test_main.py", line 11, in test_power
    self.assertEqual(power(3, 2), 6)
AssertionError: 9 != 6
```

**One failed**
```
FAIL: test_power (__main__.test_main)
Traceback (most recent call last):
  File "test_main.py", line 11, in test_power
    self.assertEqual(power(3, 2), 6)
AssertionError: 9 != 6
```

**Ran 4 tests in 0.000s**

**FAILED (failures=1)**
```
test_main (mainfile) ... failed
```

**LTAT.06.015 :Lec 09 : Continuous Testing**
# Unit Testing examples

*Unittest* assert methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Checks that</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>assertEqual(a, b)</code></td>
<td><code>a == b</code></td>
</tr>
<tr>
<td><code>assertNotEqual(a, b)</code></td>
<td><code>a != b</code></td>
</tr>
<tr>
<td><code>assertTrue(x)</code></td>
<td><code>bool(x)</code> is True</td>
</tr>
<tr>
<td><code>assertFalse(x)</code></td>
<td><code>bool(x)</code> is False</td>
</tr>
<tr>
<td><code>assertIs(a, b)</code></td>
<td><code>a is b</code></td>
</tr>
<tr>
<td><code>assertIsNot(a, b)</code></td>
<td><code>a is not b</code></td>
</tr>
<tr>
<td><code>assertIsNone(x)</code></td>
<td><code>x</code> is None</td>
</tr>
<tr>
<td><code>assertIsNotNone(x)</code></td>
<td><code>x</code> is not None</td>
</tr>
<tr>
<td><code>assertIn(a, b)</code></td>
<td><code>a in b</code></td>
</tr>
<tr>
<td><code>assertNotIn(a, b)</code></td>
<td><code>a not in b</code></td>
</tr>
</tbody>
</table>

Src: [https://docs.python.org/3/library/unittest.html](https://docs.python.org/3/library/unittest.html)
Pytest framework

Need to install pytest:
```
pip install pytest
```

Testing using pytest

```python
mainfile.py

```test_main.py

```python
import unittest

class test_main(unittest.TestCase):
    def test_add(self):
        self.assertEqual(add(3, 2), 5)
    def test_sub(self):
        self.assertEqual(sub(3, 2), 1)
    def test_multiply(self):
        self.assertEqual(multiply(3, 2), 6)
    def test_power(self):
        self.assertEqual(power(3, 2), 9)

if __name__ == '__main__':
    unittest.main()
```
Pytest framework

Testing using **pytest**

```python
import unittest

class test_main(unittest.TestCase):
    def test_add(self):
        self.assertEqual(add(3,2),5)
    def test_sub(self):
        self.assertEqual(sub(3,2),1)
    def test_multiply(self):
        self.assertEqual(multiply(3,2),6)
    def test_power(self):
        self.assertEqual(power(3,2),9)

if __name__ == '__main__':
    unittest.main()
```

```
ubuntu@gitlab-lab09-chinmaya:/~$ pytest
============================= test session starts =============================
platform linux -- Python 3.8.10, pytest-6.2.5, py-1.11.0, pluggy-1.0.0
rootdir: /home/ubuntu/unittest
plugins: cov-3.0.0
collected 4 items

test_main.py ....
============================= 4 passed in 0.03s =============================
```
Pytest framework

```python
mainfile.py

```def` add(a, b):
    return a + b

`def` test_add():
    `assert` add(3, 2) == 5

`def` sub(a, b):
    return a - b

`def` test_sub():
    `assert` sub(3, 2) == 1

`def` multiply(a, b):
    return a * b

`def` test_multiply():
    `assert` multiply(3, 2) == 6

`def` power(a, b):
    return a ** b

`def` test_power():
    `assert` power(3, 2) == 27

test_main.py

```python

```import` unittest
`from` mainfile` import` *

class` test_main(unittest.TestCase):
    `def` test_add(self):
        self.`assertEqual` add(3, 2), 5

    `def` test_sub(self):
        self.`assertEqual` sub(3, 2), 1

    `def` test_multiply(self):
        self.`assertEqual` multiply(3, 2), 6

    `def` test_power(self):
        self.`assertEqual` power(3, 2), 9

if` __name__` ==` '__main__':
    unittest.main()
Pytest framework

• pytest framework makes it easy to write small tests,
  • yet scales to support complex functional testing
• Only plain assert statements are used
• Auto-discovery of test modules and functions
• Can run unittest
Pytest in GitLab CI

DockerfilePyTest

```
FROM python:3
WORKDIR /usr/src/app

RUN pip install --no-cache-dir pytest pandas

RUN mkdir -p unittest
COPY ./unittest unittest

RUN mkdir report

CMD ["pytest", "--junitxml=./report/test_report.xml", "unittest"]
```

**Assume**: the source and testing codes are inside *unittest* directory in the gitlab repo.

*unittest* directory is in the project’s root directory
Pytest in GitLab CI

.gitlab-ci.yml

```yaml
test_report_using_pytest:
  stage: test
  script:
    - docker login -u dehury -p pw gitlab.cs.ut.ee:5050
    - docker build -t $PYTEST_IMAGE_NAME -f ./dehury/docker/DockerfilePyTest .
    - docker run -v $PWD:/usr/src/app/report/ $PYTEST_IMAGE_NAME
  artifacts:
    when: always
    expose_as: 'Test Report'
    paths: [test_report.xml]
    reports:
      junit: test_report.xml
```
Pytest and GitLab CI workflow

1. Commit and push the code
2. Clone the repo
3. Build docker image
4. Create container for testing
5. Generate the report
6. Push the report as artifact
7. Developer gets the report

 ETAIS VM

Your new Docker image

New container for testing the code

Testing report in xml or html format

Gitlab Runner

Clone: Your Gitlab project

YOUR Gitlab project

Testing report as artifacts

Gitlab CI server

Docker Engine

Your new Docker image

gitLab.cs.ut.ee

Developer
Code coverage

• a metric that can help developer understand how much of the source code is tested.
• Measure through testing tool
• assess the quality of your test suite,
• The coverage report may include:
  • Function coverage: number of functions covered by test code
  • Statement coverage: Number of the statements in the program executed.
  • Branches coverage: how many of the branches of the control structures (if statements for instance) have been executed.
  • Condition coverage: how many of the boolean sub-expressions have been tested for a true and a false value.
  • Line coverage: Total number of lines of source code tested.
• Should you always aim for 100% code coverage
  • Build phase may fail a lot if code coverage is too high
Code coverage

• a metric that can help developer understand how much of the source code is tested.

• Some Existing tools to calculate coverage:
  • Coverage.py, Pytest-cov for python language
  • SimpleCov, coverrail, coverband, rcov for Ruby
  • PHPUnit, PCOV, Xdebug for PHP
  • Blanket.js, Istanbul for Javascript
  • JaCoCo, Jcov, Atlassian Clover, Cobertura for Java
Code coverage

Some useful commands

coverage run -m pytest
coverage report
Code coverage

Some useful commands

- coverage run -m pytest
- coverage html -d cov_html_report_dir

Export the report in HTML format.
Code coverage

Some useful commands

- coverage run -m pytest
- coverage html -d cov_html_report_dir

These commands are used to analyze and report code coverage.

Example code:

```python
src/analyze_csv_with_test.py: 49% coverage

def add_indValue(filepath, newPathFile):
    df_tmp = pd.read_csv(filepath)
    indValue = []
    for i in range(len(df_tmp['value'])):
        if i == 0:
            indValue.append(0)
            continue
        indValue.append(df_tmp['value'][i] - df_tmp['value'][i-1])
    df_tmp['indValue'] = indValue.copy()
    df_tmp.to_csv(newPathFile)
```

The code above is an example of how to implement code coverage in a Python project.
Code Quality

• How it is different form code coverage metric?
  • You don’t need any testing tool to see the code quality

• Answer the following:
  • Do you have any function with more than 6 arguments?
  • Any function with a so many return statements?
  • What is the length of the functions?
  • Any duplicate/similar code?
  • So many nested if-else statements
  • So many methods/functions in a class
Code Quality

Software Tool: **Code climate**

- [https://docs.codeclimate.com/docs/configuring-your-analysis](https://docs.codeclimate.com/docs/configuring-your-analysis)

- Can be easily configured with GitLab CI pipeline.

- Gitlab understands the exported code quality report
Code Quality

Sample code quality reporting by Gitlab after creating a merge request
Sample code quality reporting by Gitlab after creating a merge request:

- Major - Function `many_arg` has 8 arguments (exceeds 4 allowed). Consider refactoring in `dehuny/src/analyze_csv.py:7`
- Major - Avoid too many `return` statements within this function in `dehuny/src/analyze_csv.py:46`
- Major - Avoid too many `return` statements within this function.

Cognitive Complexity is a measure of how difficult a unit of code is to intuitively understand or how difficult the code will be to read and understand.
Code Quality

• Higher quality codes are:
  • Easy to read, better formatting
  • Well documented
  • Easy to change
Summary

• What is testing?
• Different types of testing
• Manual, Automated, and Continuous Testing
• Functional and non functional testing
• {Black,White,Gray} Box Testing
• Alpha, Pilot, Beta Testing
• Unit Testing (using unittest and pytest)
• Testing in GitLab CI
• code coverage
• Code quality
Lab Sessions - Rough

- Do unit testing in CLI
- Unit Testing in GitLab CI
- Test report
- Code coverage report
- Code quality report
References

- https://www.nationalgeographic.org/encyclopedia/Y2K-bug/
- https://www.softwaretestinghelp.com/types-of-software-testing/
- https://www.perfecto.io/blog/automated-testing-vs-manual-testing-vs-continuous-testing
- https://www.xenonstack.com/insights/what-is-continuous-testing
- https://www.xenonstack.com/insights/what-is-unit-testing
- https://www.xenonstack.com/insights/what-is-functional-testing
- https://docs.gitlab.com/ee/user/project/merge_requests/code_quality.html
References

- https://www.softwaretestinghelp.com/python-testing-frameworks/
- https://docs.python.org/3/library/unittest.html
- https://www.geeksforgeeks.org/types-software-testing/
- https://www.atlassian.com/continuous-delivery/software-testing/code-coverage
- https://php.watch/articles/php-code-coverage-comparison
- https://www.ruby-toolbox.com/categories/code_coverage
- https://about.codecov.io/blog/the-best-code-coverage-tools-by-programming-language/
- https://www.codegrip.tech/productivity/everything-you-need-to-know-about-code-coverage/
- https://courses.cs.ut.ee/2021/SWT2021/spring/Main/Loengud
- https://www.tricentis.com/products/what-is-continuous-testing/
Any Question?

THANK YOU