INNOVATE

DATA AND AI/ML EDITION

22 February 2023

aws INNOVATE
DATA AND AI/ML EDITION

Bringing software engineering best practices to data science and machine learning

Joshua Goyder
Senior Data Science Consultant
Amazon Web Services

Dr. Marcel Vonlanthen
Senior Data Science Consultant
Amazon Web Services

Agenda

1. Why bring software engineering best practices to data science (DS)/machine learning (ML)?

2. Software testing

3. Debugging

4. Recap

Why bring software engineering practices to data science/machine learning?

The opportunities and challenges of ML

Optimizing businesses with new efficiencies

Adding new capabilities to existing products

Inventing new areas and products

The opportunities and challenges of ML

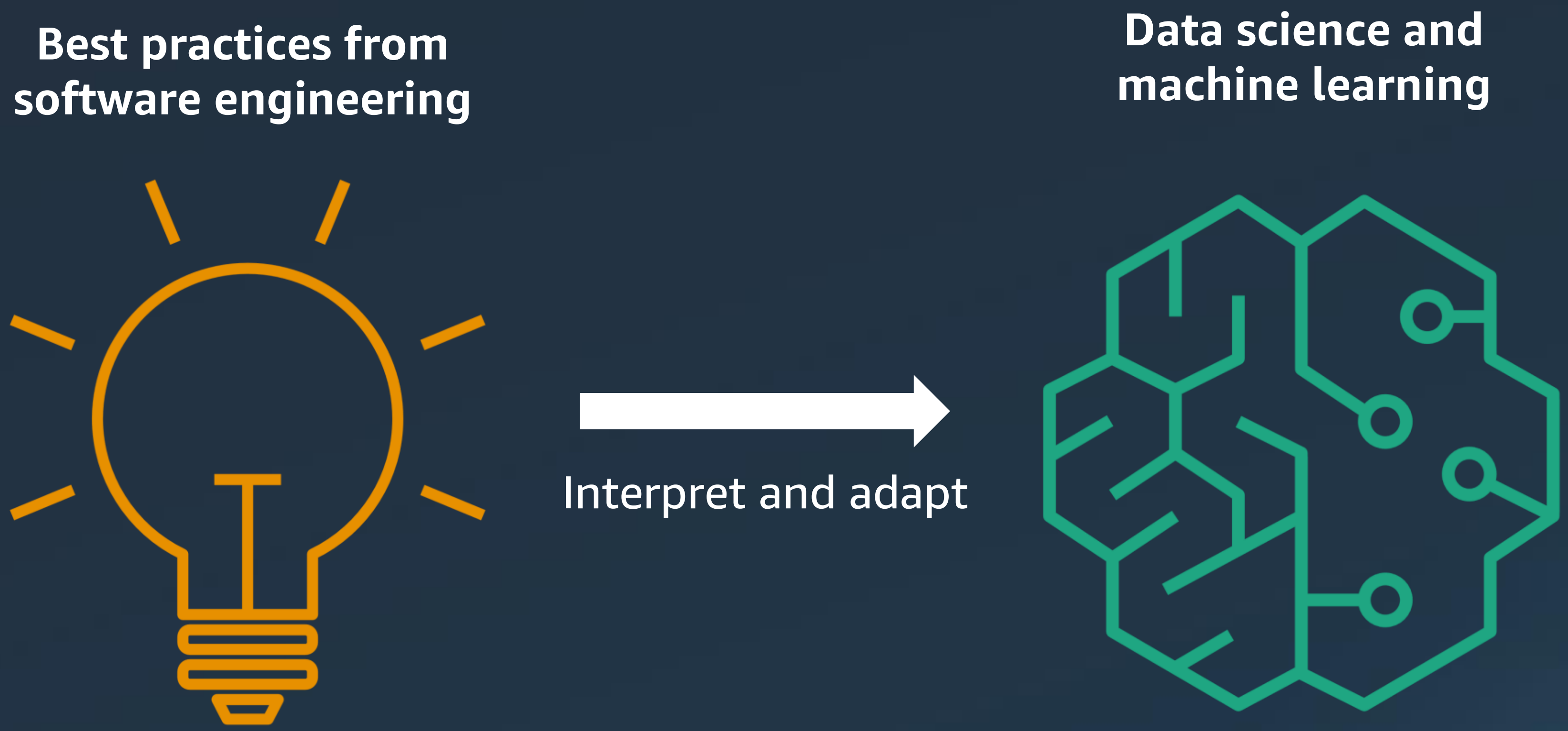
Optimizing businesses with new efficiencies

Adding new capabilities to existing products

Inventing new areas and products

Encountering new (and old) challenges

What can we do about it?



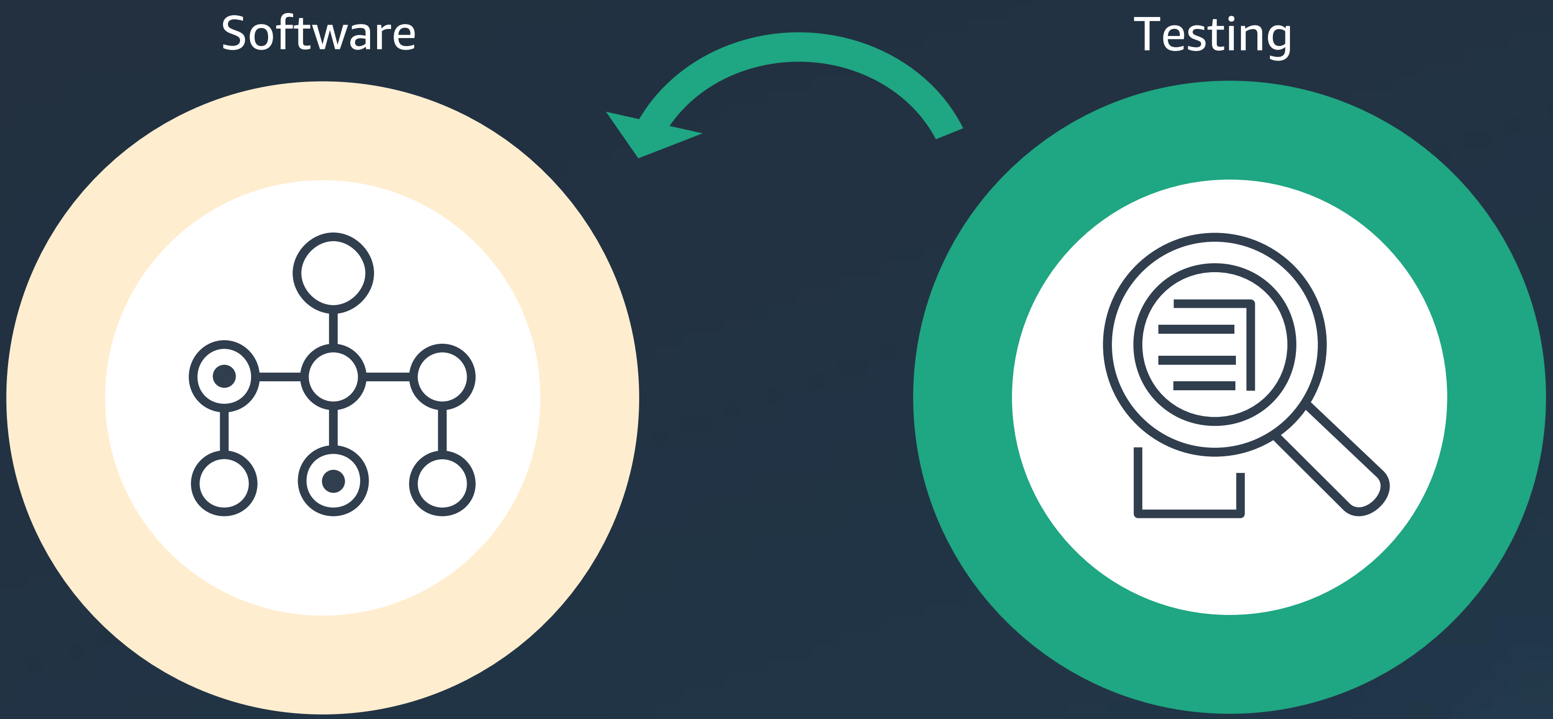
Best practices from software engineering

Interpret and adapt

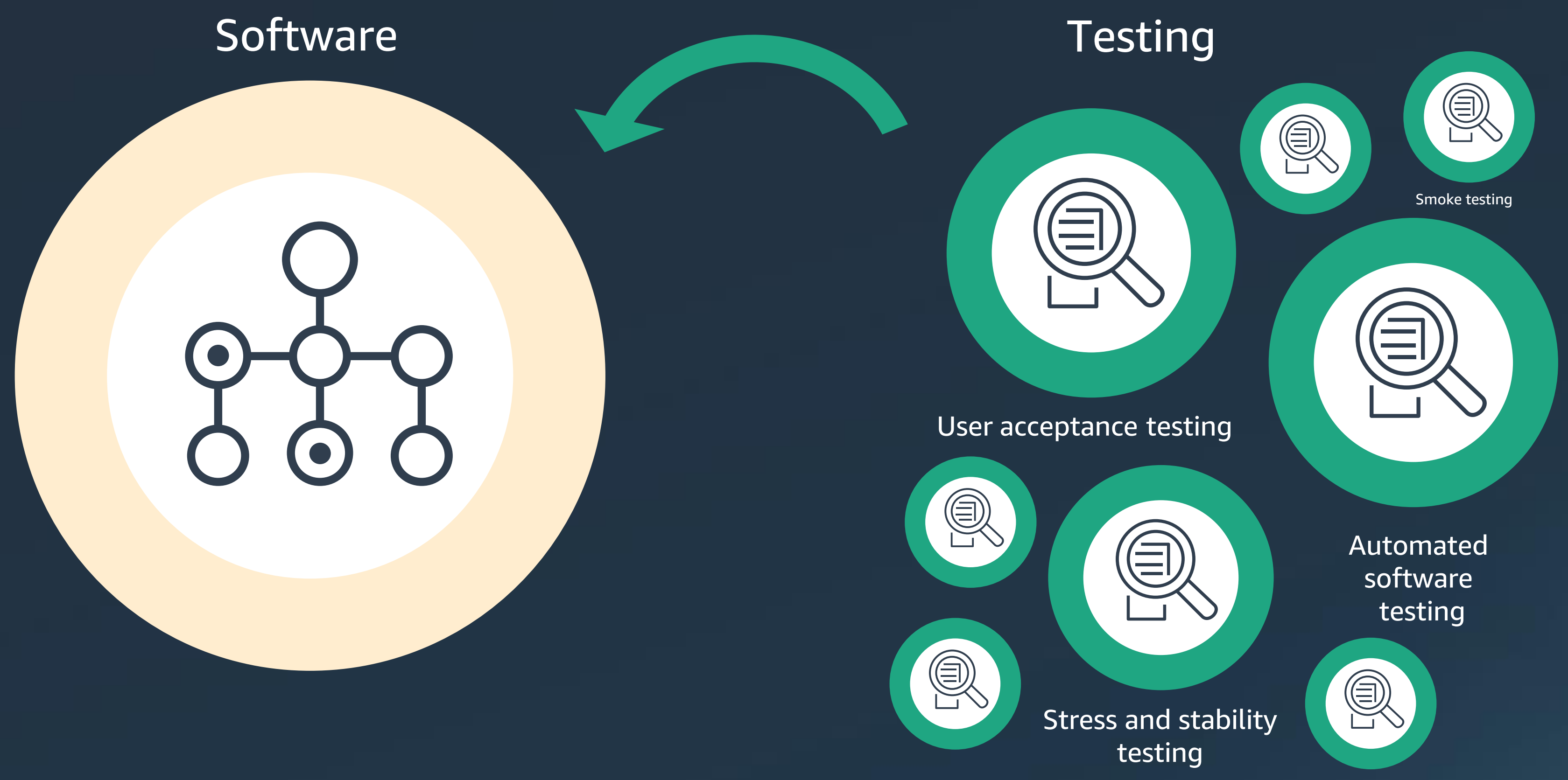
Data science and machine learning

Testing

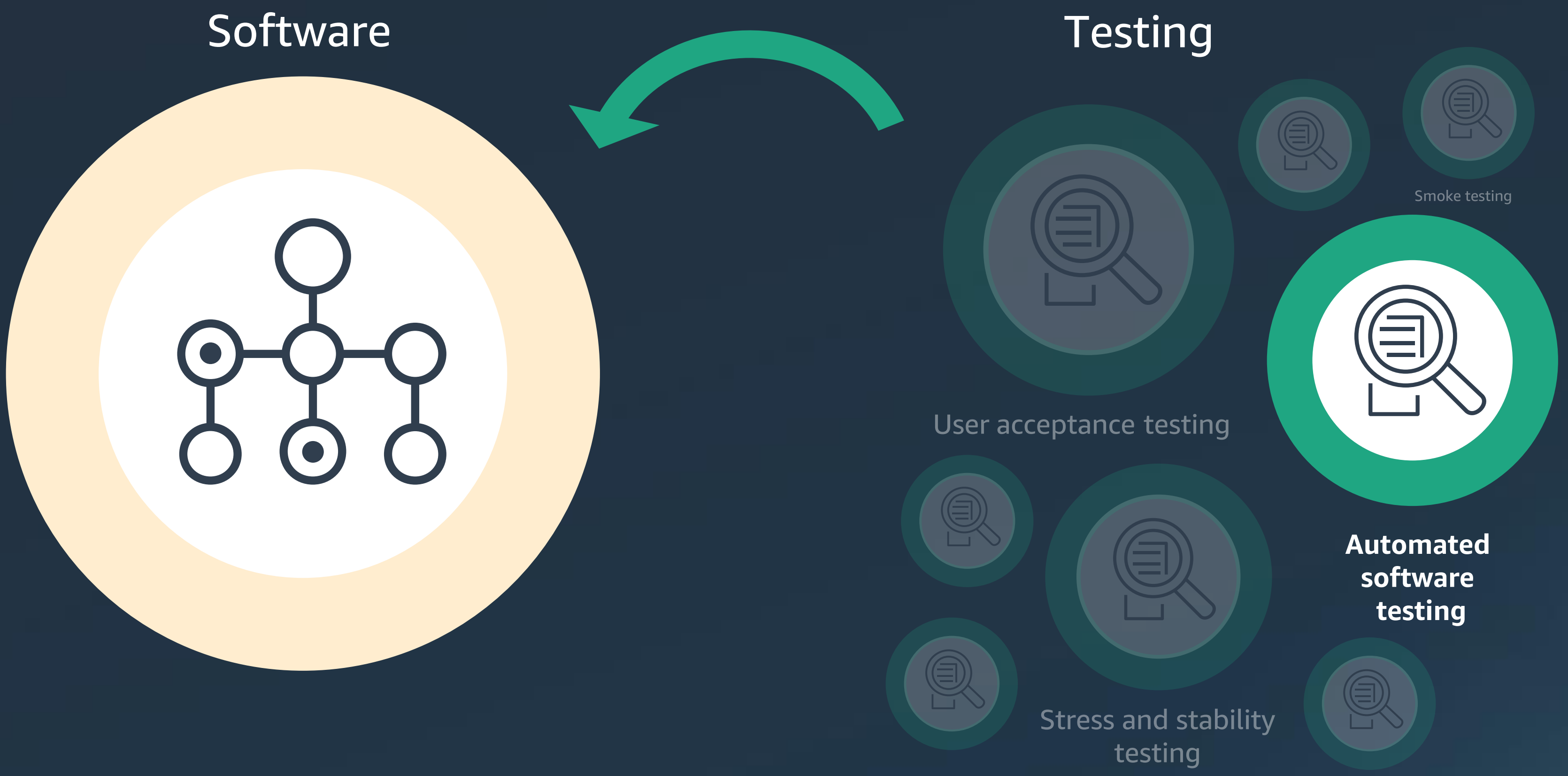
A brief overview of software testing



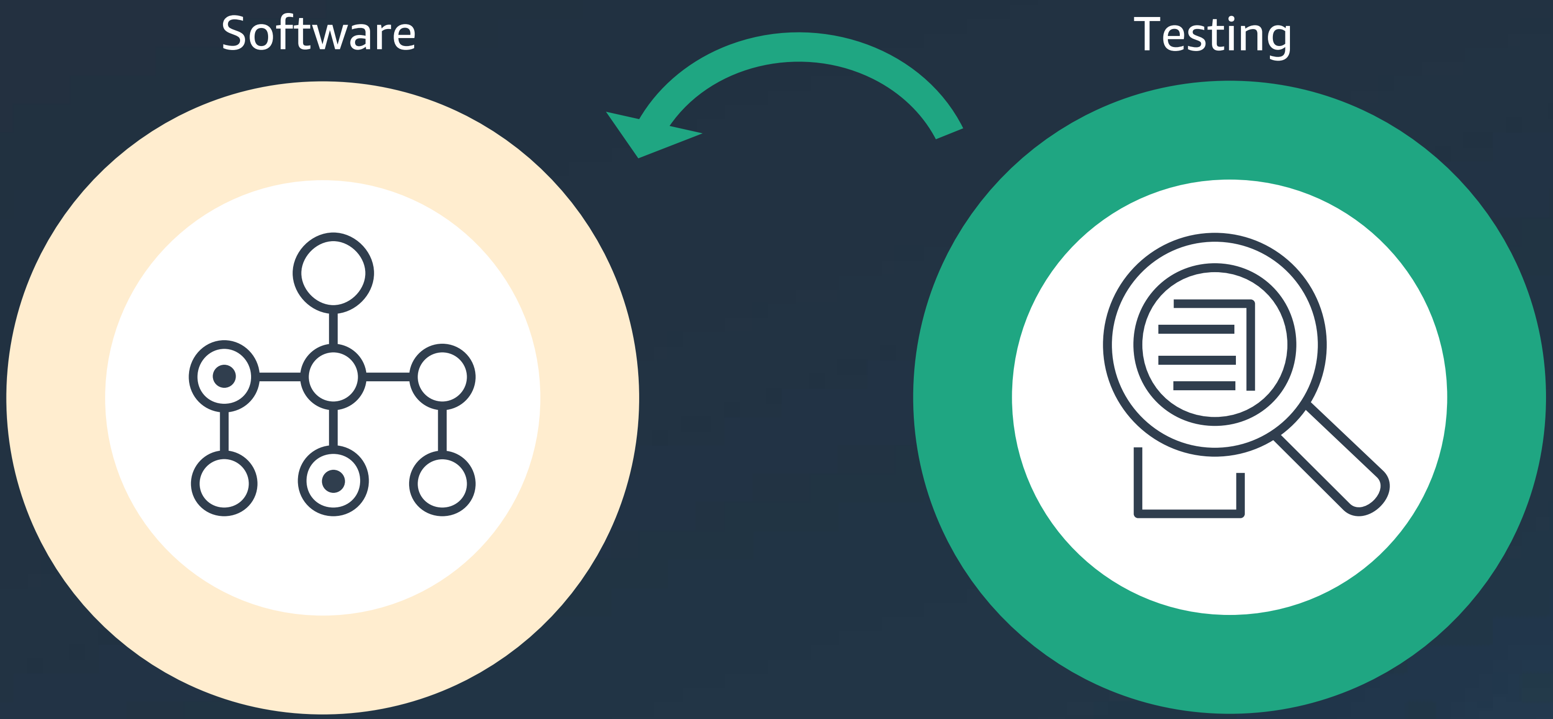
A brief overview of software testing



A brief overview of software testing



A brief overview of software testing



Software

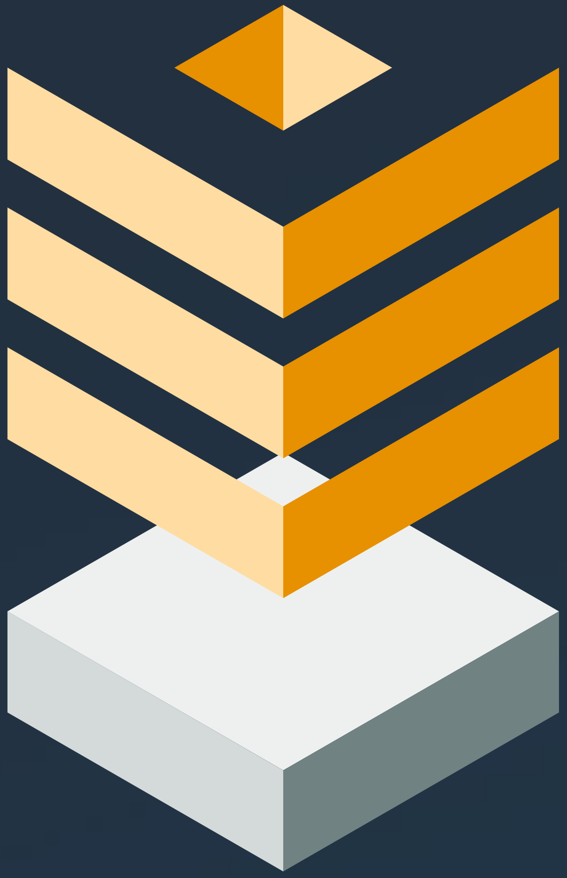
Testing

Can we stop our software doing the wrong thing?

A brief overview of software testing

Our project

A brief overview of software testing



Our project

Tests of our product features

Tests of our external integrations

Tests of our functions and classes

A brief overview of software testing

Our project

Tests of our product features

Tests of our external integrations

Tests of our functions and classes

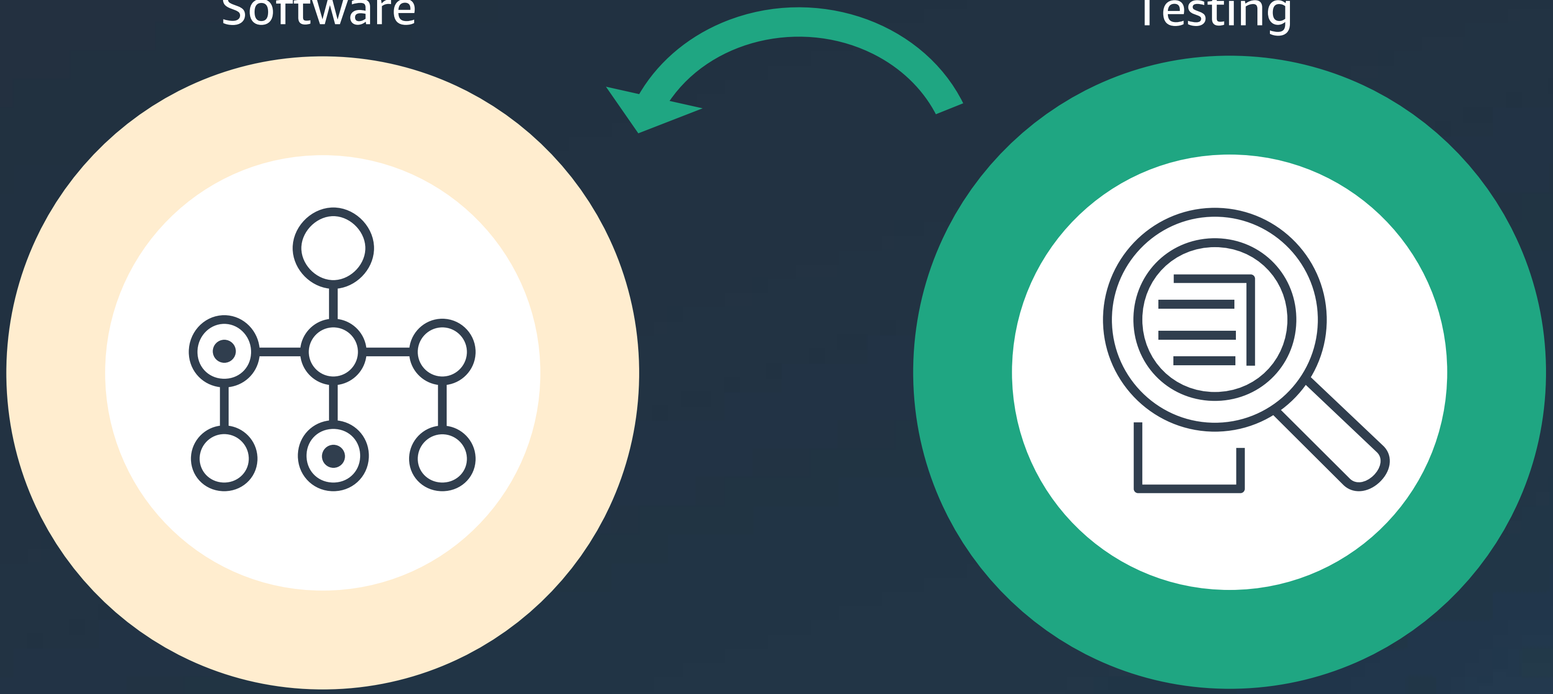
Builds a living spec of what your code does

An analytical tool to assess the health of your code

…and significantly reduces bugs in your code!

A brief overview of software testing

Our project



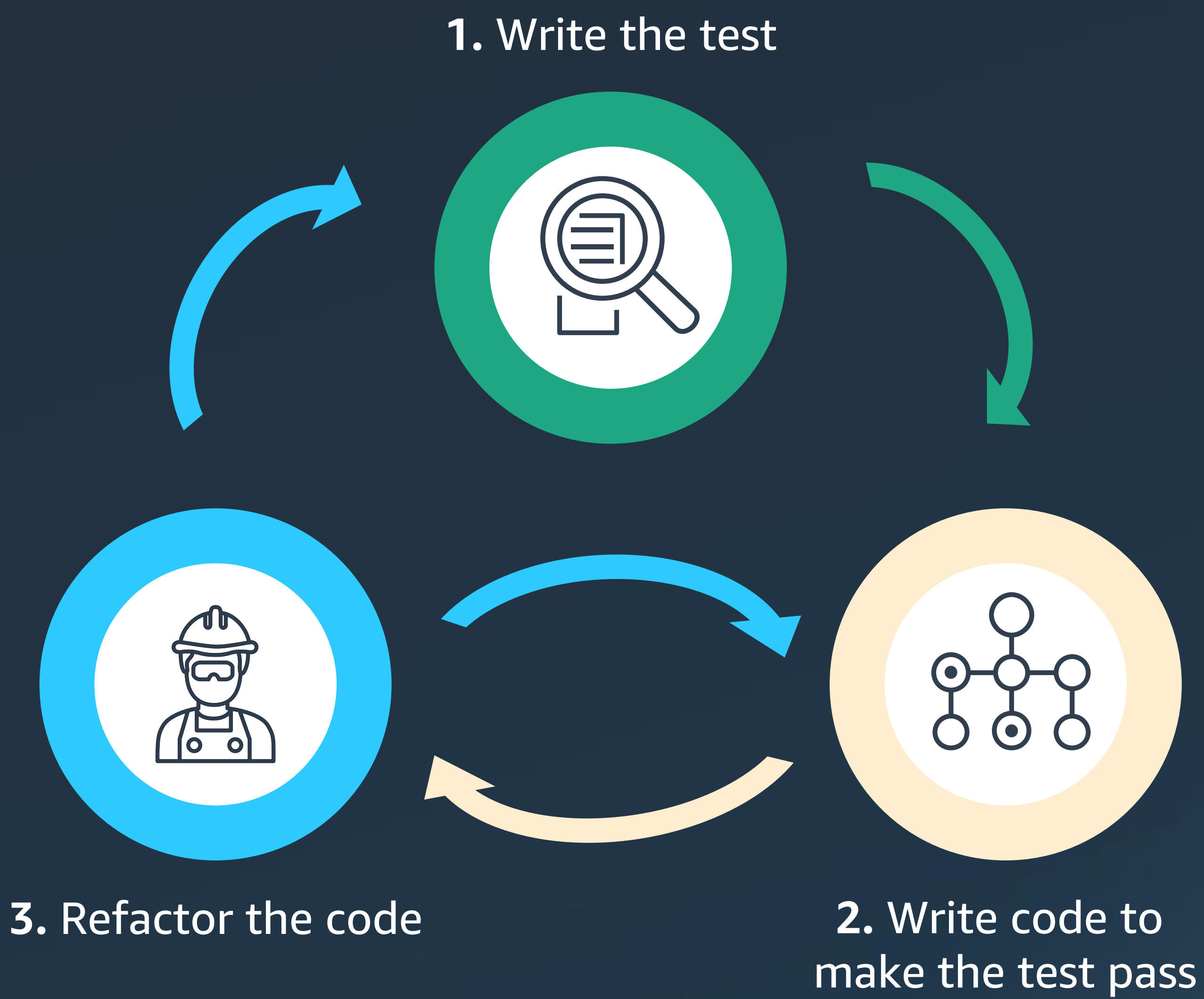
Software

Testing

How the computer does something

What *should* the computer be doing?

Test driven development



1. Write the test

2. Write code to make the test pass

3. Refactor the code

Applying tests to DS/ML

Types of tests

Standard test structure

1. Arrange

Define a situation

Specify the inputs

Specify the desired outputs

2. Act

Execute an action that we want to assess

3. Assert

Check that the outcome of our action is what we expected it to be

Types of tests

	Unit tests	Integration tests	Acceptance tests
	Assess the core components of our code, e.g. functions and classes	*Assess whether our code works with other code*	*Assess whether features of our code do what they are defined to do*
Example			
Arrange			
Act			
Assert			

Scale of component being tested

Types of tests

	Unit tests	Integration tests	Acceptance tests
	Assess the core components of our code, e.g. functions and classes	*Assess whether our code works with other code*	*Assess whether features of our code do what they are defined to do*
Example	**Testing for Pandas DataFrame column transformation**		
Arrange			
Act			
Assert			

Scale of component being tested

Unit testing

Standard test structure

1. Arrange

Define a situation

Specify the inputs

Specify the desired outputs

2. Act

Execute an action that we want to assess

3. Assert

Check that the outcome of our action is what we expected it to be

```python
import pytest
import pandas as pd
import ourcode

def test_upper_limit_transformation():
    # Arrange
    df_input = pd.DataFrame({"input_col": [1, 2, 3]})
    upper_limit = 1
    df_expected_output = pd.DataFrame(
        {"expected_output_col": [True, False, False]})
```

Unit testing

Standard test structure

1. Arrange

Define a situation

Specify the inputs

Specify the desired outputs

2. Act

Execute an action that we want to assess

3. Assert

Check that the outcome of our action is what we expected it to be

```python
import pytest
import pandas as pd
import ourcode

def test_upper_limit_transformation():
    # Arrange
    df_input = pd.DataFrame({"input_col": [1, 2, 3]})
    upper_limit = 1
    df_expected_output = pd.DataFrame(
        {"expected_output_col": [True, False, False]})

    # Act
    df_output = ourcode.values_are_below_limit(
        df=df_input,
        column="input_col",
        upper_limit=upper_limit
        )
```

Unit testing

Standard test structure

1. Arrange

Define a situation

Specify the inputs

Specify the desired outputs

2. Act

Execute an action that we want to assess

3. Assert

Check that the outcome of our action is what we expected it to be

```python
import pytest
import pandas as pd
import ourcode

def test_upper_limit_transformation():
    # Arrange
    df_input = pd.DataFrame({"input_col": [1, 2, 3]})
    upper_limit = 1
    df_expected_output = pd.DataFrame(
        {"expected_output_col": [True, False, False]})

    # Act
    df_output = ourcode.values_are_below_limit(
        df=df_input,
        column="input_col",
        upper_limit=upper_limit
        )

    # Assert
    assert (
        df_output["output_col"]
        .equals(df_expected_output["df_expected_output"]))
```

Types of tests

	Unit tests	Integration tests	Acceptance tests
	Assess the core components of our code, e.g. functions and classes	*Assess whether our code works with other code*	*Assess whether features of our code do what they are defined to do*
Example	**Testing for Pandas DataFrame column transformation**		**Testing a Lambda that gathers data and outputs a training set to a S3**
Arrange	Define inputs and outputs to a DataFrame transformation function		
Act	Call the function		
Assert	Check that the outputs match expected outputs		

Scale of component being tested

Acceptance testing

1. Arrange

Define a situation

Specify the inputs

Specify the desired outputs

2. Act

Execute an action that we want to assess

3. Assert

Check that the outcome of our action is what we expected it to be

```python
import boto3
import json

lambda_client = boto3.client("lambda")
s3_client = boto3.client("s3")

def test_dataset_generation_lambda():
    # Arrange
    output_key = "test/test_dataset.csv"
    function_params = {
        "output_key": output_key
        }
```

Acceptance testing

1. Arrange

Define a situation

Specify the inputs

Specify the desired outputs

2. Act

Execute an action that we want to assess

3. Assert

Check that the outcome of our action is what we expected it to be

```python
import boto3
import json

lambda_client = boto3.client("lambda")
s3_client = boto3.client("s3")

def test_dataset_generation_lambda():
    # Arrange
    output_key = "test/test_dataset.csv"
    function_params = {
        "output_key": output_key
        }

    # Act
    lambda_client.invoke(
        FunctionName="DatasetGenerationFunction",
        Payload=json.dumps(function_params)
        )
```

Acceptance testing

1. Arrange

Define a situation

Specify the inputs

Specify the desired outputs

2. Act

Execute an action that we want to assess

3. Assert

Check that the outcome of our action is what we expected it to be

```python
import boto3
import json

lambda_client = boto3.client("lambda")
s3_client = boto3.client("s3")

def test_dataset_generation_lambda():
    # Arrange
    output_key = "test/test_dataset.csv"
    function_params = {
        "output_key": output_key
        }

    # Act
    lambda_client.invoke(
        FunctionName="DatasetGenerationFunction",
        Payload=json.dumps(function_params)
        )

    # Assert
    assert output_key in s3_client.list_objects(
        Bucket="ExampleBucket", Prefix=output_key)
    # Can include further tests to assess format of produced dataset
```

Types of tests

	Unit tests	Integration tests	Acceptance tests
	Assess the core components of our code, e.g. functions and classes	*Assess whether our code works with other code*	*Assess whether features of our code do what they are defined to do*
Example	**Testing for Pandas DataFrame column transformation**	**Testing that a SageMaker training job executes successfully**	**Testing a Lambda that gathers data and outputs a training set to S3**
Arrange	Define inputs and outputs to a DataFrame transformation function		Define the parameters to our Lambda
Act	Call the function		Invoke the Lambda
Assert	Check that the outputs match expected outputs		Check that the Lambda ran and produced our outputs

Scale of component being tested

Integration testing

Standard test structure

1. Arrange

Define a situation

Specify the inputs

Specify the desired outputs

2. Act

Execute an action that we want to assess

3. Assert

Check that the outcome of our action is what we expected it to be

```python
from sagemaker.pytorch import PyTorch

def test_pytorch_training():
    # Arrange
    pytorch_estimator = PyTorch(
        "pytorch-train.py",
        instance_type="local",
        instance_count=1,
        framework_version="1.5.0",
        hyperparameters={"epochs": 2, "row_limit": 100}
    )
```

Integration testing

Standard test structure

1. Arrange

Define a situation

Specify the inputs

Specify the desired outputs

2. Act

Execute an action that we want to assess

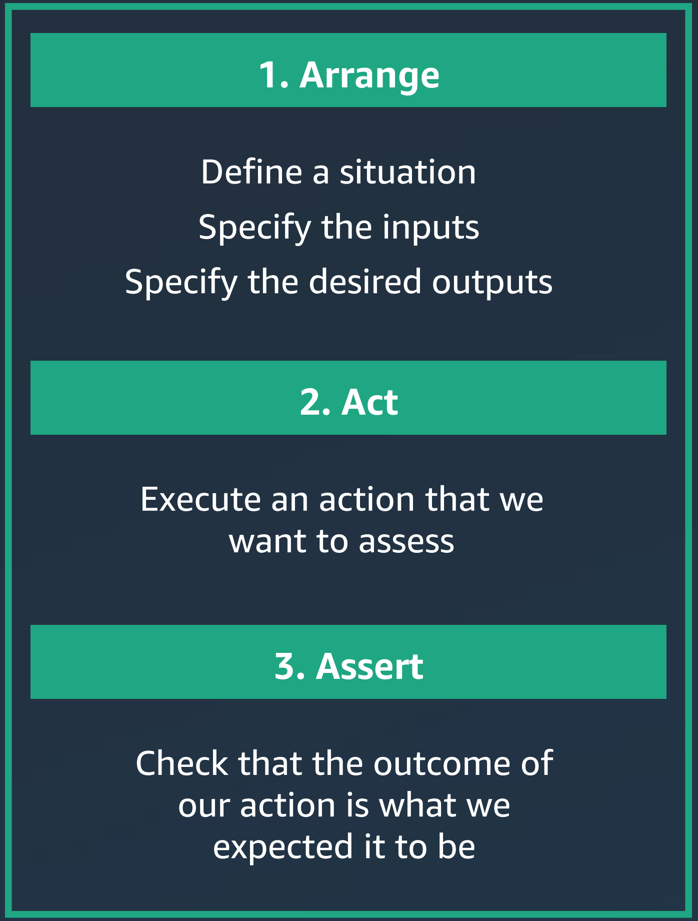
3. Assert

Check that the outcome of our action is what we expected it to be

```python
from sagemaker.pytorch import PyTorch

def test_pytorch_training():
    # Arrange
    pytorch_estimator = PyTorch(
        "pytorch-train.py",
        instance_type="local",
        instance_count=1,
        framework_version="1.5.0",
        hyperparameters={"epochs": 2, "row_limit": 100}
    )

    # Act
    pytorch_estimator.fit({
        "train": "s3://my-bucket/path/to/training/data.csv",
        "test": "s3://my-bucket/path/to/test/data.csv",
    })
```

Integration testing

Standard test structure



1. Arrange

Define a situation
Specify the inputs
Specify the desired outputs

2. Act

Execute an action that we want to assess

3. Assert

Check that the outcome of our action is what we expected it to be

```python
from sagemaker.pytorch import PyTorch

def test_pytorch_training():
    # Arrange
    pytorch_estimator = PyTorch(
        "pytorch-train.py",
        instance_type="local",
        instance_count=1,
        framework_version="1.5.0",
        hyperparameters={"epochs": 2, "row_limit": 100}
    )

     # Act
     pytorch_estimator.fit({
         "train": "s3://my-bucket/path/to/training/data.csv",
         "test": "s3://my-bucket/path/to/test/data.csv",
     })

     # Assert
     # No explicit assert – the job running to completion is a pass.
     # We could also deploy an endpoint and test we get a response.
```

Types of tests

	Unit tests	Integration tests	Acceptance tests
	Assess the core components of our code, e.g. functions and classes	*Assess whether our code works with other code*	*Assess whether features of our code do what they are defined to do*
Example	**Testing for Pandas DataFrame column transformation**	**Testing that a SageMaker training job executes successfully**	**Testing a Lambda that gathers data and outputs a training set to S3**
Arrange	Define inputs and outputs to a DataFrame transformation function	Define the input parameters to a SageMaker training job	Define the parameters to our Lambda
Act	Call the function	Call the training job	Invoke the Lambda
Assert	Check that the outputs match expected outputs	Check that the training job ran successfully	Check that the Lambda ran and produced our outputs

Scale of component being tested

First steps

First steps

How do I start including tests?

Unit tests are often the most straightforward place to start.

Specifically, use *pytest* and write tests for Pandas transformations.

First steps

How do I start including tests?

Unit tests are often the most straightforward place to start.

Specifically, use *pytest* and write tests for Pandas transformations.

How do I structure my tests?

One approach: write and install your own code as a Python package, and then have a separate “test” directory.

Structuring your tests

One approach: write and install
your own code as a Python package,
and then have a separate “test” directory.

```
mycode
├── LICENSE.md
├── README.md
├── setup.py
├── mycode
|    ├── __init__.py
|    ├── models.py
|    ├── settings.py
|    └── transformations.py
└── tests
     ├── unit_tests
     |    ├── test_models.py
     |    ├── test_settings.py
     |    └── test_transformations.py
     └── acceptance_tests
          └── test_model_deployment.py
```

First steps

How do I start including tests?

Unit tests are often the most straightforward place to start.

Specifically, use *pytest* and write tests for Pandas transformations.

How do I structure my tests?

One approach: write and install your own code as a Python package, and then have a separate "test" directory.

If in doubt, copy from the greats!

First steps

How do I start including tests?

Unit tests are often the most straightforward place to start.

Specifically, use *pytest* and write tests for Pandas transformations.

How do I structure my tests?

One approach: write and install your own code as a Python package, and then have a separate “test” directory.

If in doubt, copy from the greats!

Testing is making me write code differently.

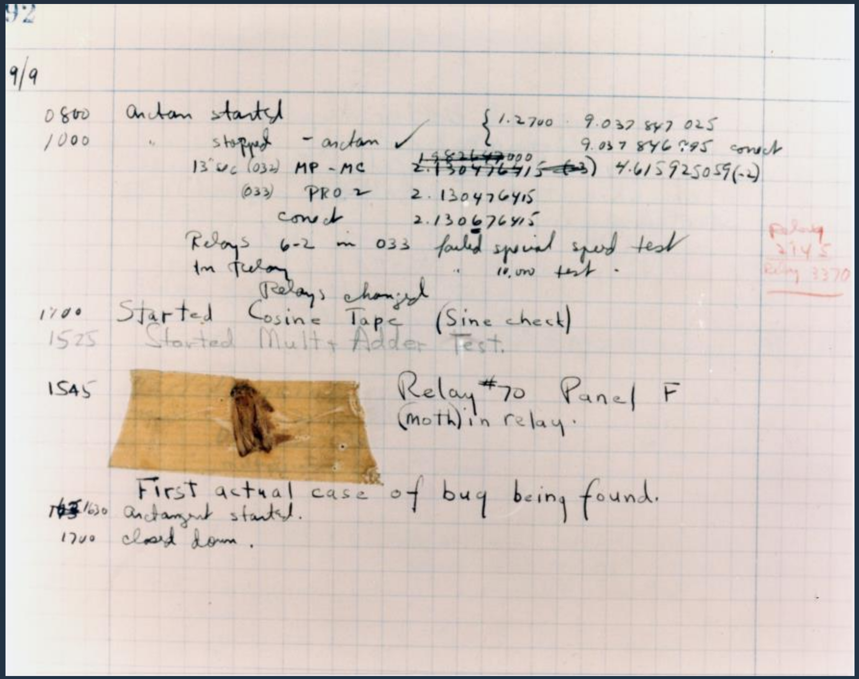
Great!

Writing testable code encourages good practices, and makes deploying your code much easier.

Debugging

What can we do about?

Debugging is the process of finding and resolving defects that prevent correct operation of a computer programs or a software. These defects are call bugs.



92

9/9

0800 Antan started
1000 " stopped - antan ✓ {1.2700 9.037 847 025
9.037 846 795 conect
13"uc (032) MP - MC ~~1.982147000~~ ~~2.130476415 (-3)~~ 4.615925059(-2)
(033) PRO 2 2.130476415
conect 2.130676415
Relays 6-2 in 033 failed special speed test
in relay " 10,000 test.
Relays changed
1100 Started Cosine Tape (Sine check)
1525 Started Mult + Adder Test.
1545 Relay #70 Panel F
(moth) in relay.
First actual case of bug being found.
~~143~~1630 Antangent started.
1700 closed down.

Relay 2145
Relay 3370

An actual bug (i.e. a moth) found in 1947 in one of the early computer, preventing correct operation. “bug” recorded and correct by Grace Hopper, a pioneer in computer programming

What can we do about?

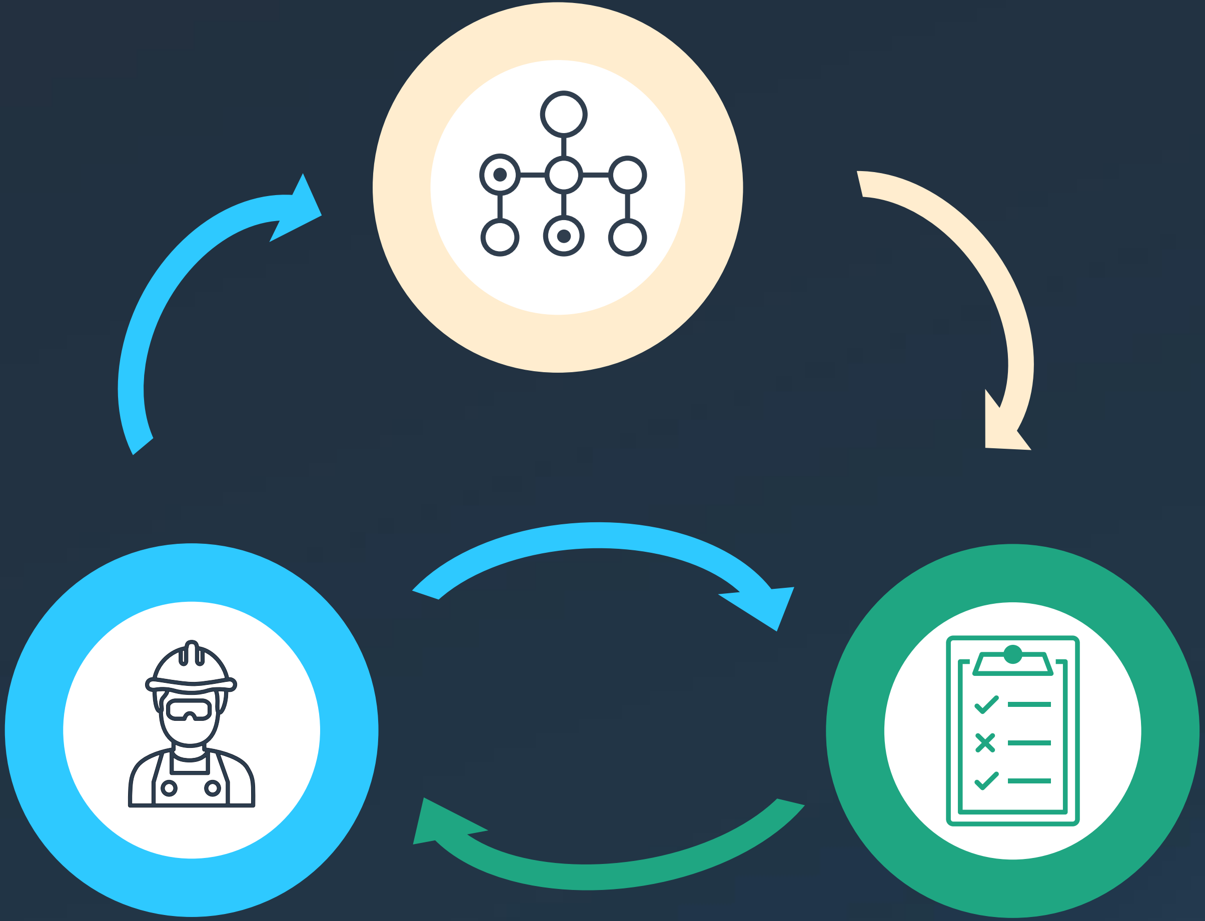
Debugging includes (among the most common):

- Interactive debugging
- Testing
- Profiling

In this presentation, we are cover *interactive debugging* and *testing.*

aws

The code writing loop



1. Write the code

2. Execute and debug the code

3. Refactor the code With the insights from the previous phase

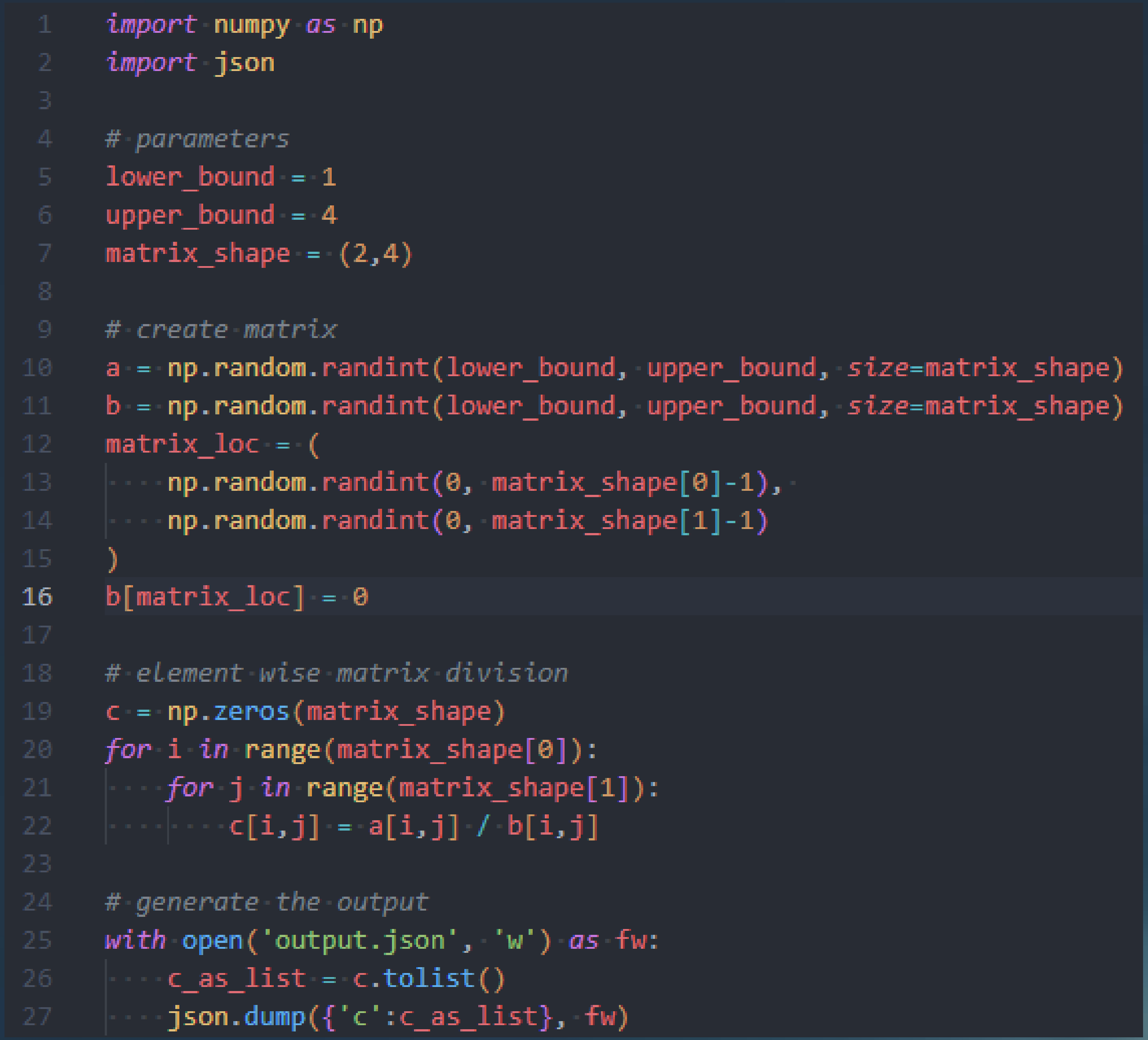
Write the code

Code purpose:

- Element wise matrix division between matrix a and b

Expected output:

- c matrix written in a JSON file where c is defined as $c_{ij} = a_{ij}/b_{ij}$, without using the Einstein summation rule.

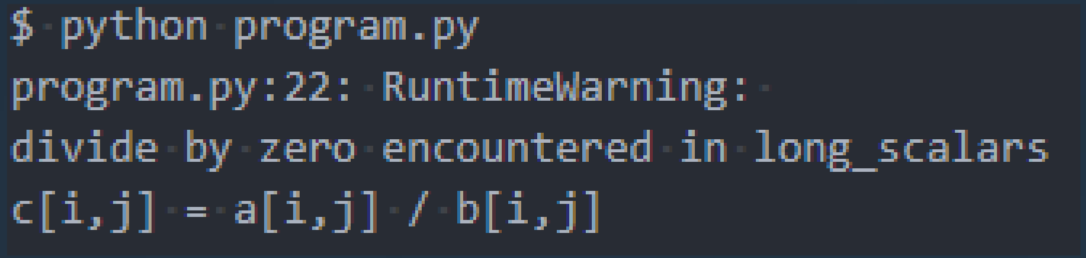


```python
import numpy as np
import json

# parameters
lower_bound = 1
upper_bound = 4
matrix_shape = (2,4)

# create matrix
a = np.random.randint(lower_bound, upper_bound, size=matrix_shape)
b = np.random.randint(lower_bound, upper_bound, size=matrix_shape)
matrix_loc = (
    np.random.randint(0, matrix_shape[0]-1), 
    np.random.randint(0, matrix_shape[1]-1)
)
b[matrix_loc] = 0

# element wise matrix division
c = np.zeros(matrix_shape)
for i in range(matrix_shape[0]):
    for j in range(matrix_shape[1]):
        c[i,j] = a[i,j] / b[i,j]

# generate the output
with open('output.json', 'w') as fw:
    c_as_list = c.tolist()
    json.dump({'c':c_as_list}, fw)
```

Write the code

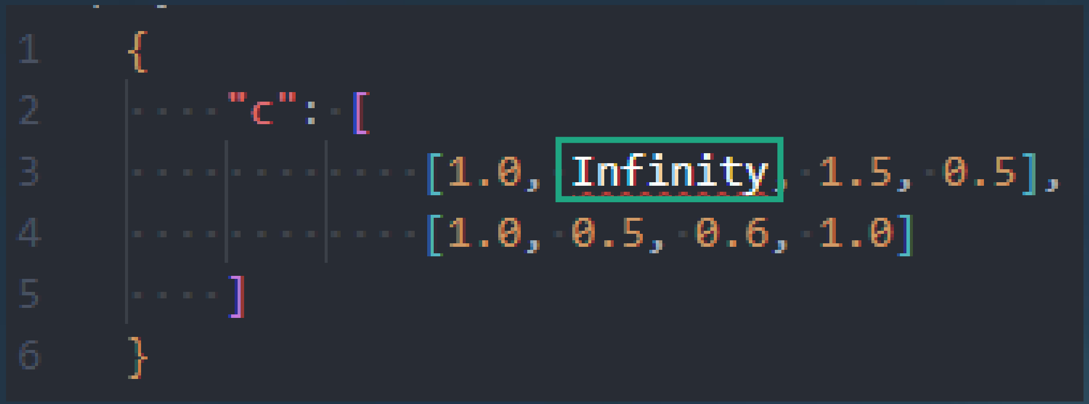
Actual output

- The execution returns a *divide by zero runtime warning* (but still executes)



```
$ python program.py
program.py:22: RuntimeWarning:
divide by zero encountered in long_scalars
c[i,j] = a[i,j] / b[i,j]
```

- The execution generates a non valid JSON with a list mixing data types



```
{
    "c": [
            [1.0, Infinity, 1.5, 0.5],
            [1.0, 0.5, 0.6, 1.0]
    ]
}
```

How to debug - Static debugging

Static debugging

1. How to

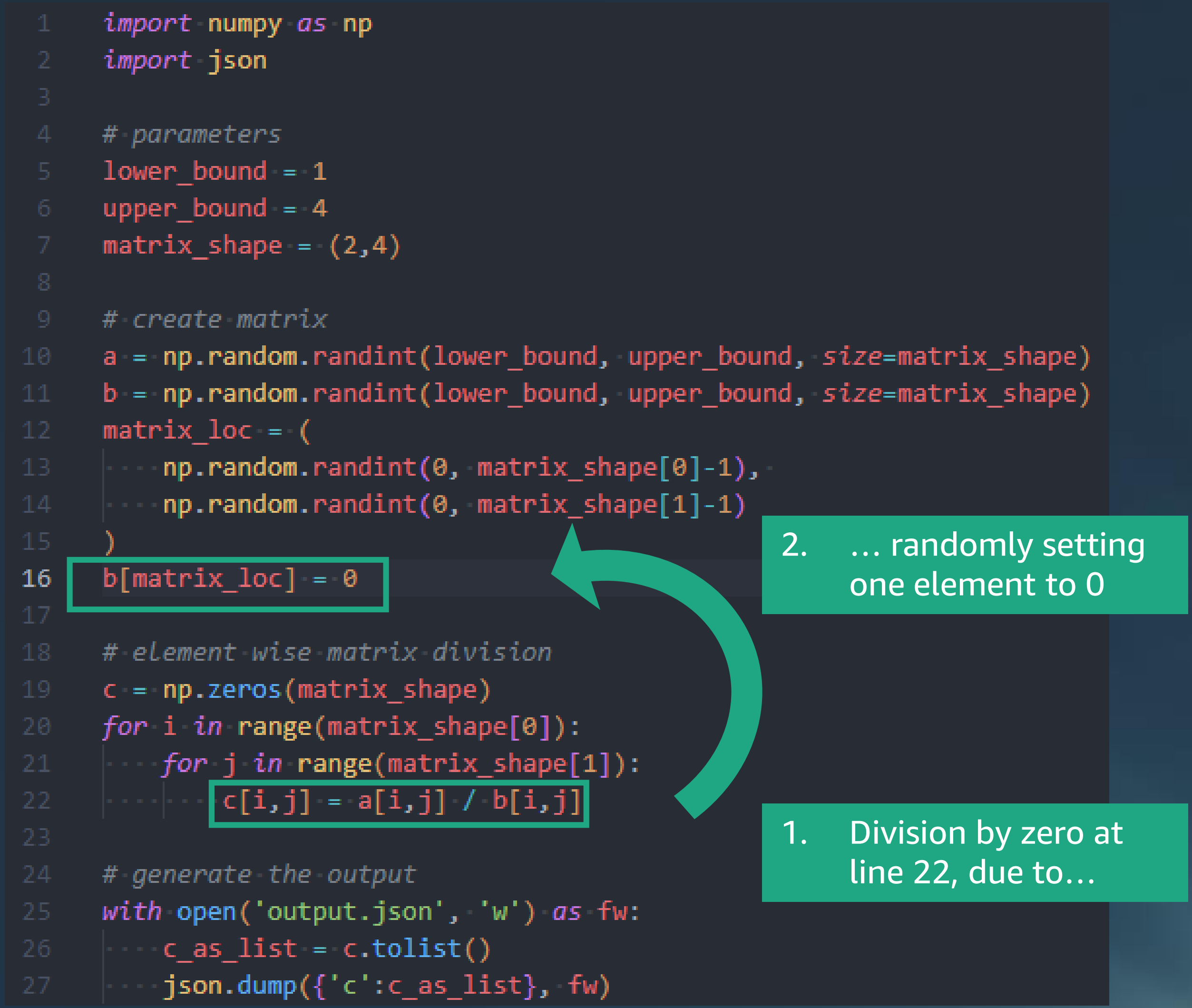
- Debugging by reading the code

2. Pros

- Generally fast

2. Cons

- Works only with very simple cases
- Generally only feasible by highly experienced programmer
- Requires deep knowledge of the language syntax and mechanism



```
 1  import numpy as np
 2  import json
 3
 4  # parameters
 5  lower_bound = 1
 6  upper_bound = 4
 7  matrix_shape = (2,4)
 8
 9  # create matrix
10  a = np.random.randint(lower_bound, upper_bound, size=matrix_shape)
11  b = np.random.randint(lower_bound, upper_bound, size=matrix_shape)
12  matrix_loc = (
13      np.random.randint(0, matrix_shape[0]-1),
14      np.random.randint(0, matrix_shape[1]-1)
15  )
16  b[matrix_loc] = 0
17
18  # element wise matrix division
19  c = np.zeros(matrix_shape)
20  for i in range(matrix_shape[0]):
21      for j in range(matrix_shape[1]):
22          c[i,j] = a[i,j] / b[i,j]
23
24  # generate the output
25  with open('output.json', 'w') as fw:
26      c_as_list = c.tolist()
27      json.dump({'c':c_as_list}, fw)
```

2. … randomly setting one element to 0

1. Division by zero at line 22, due to…

How to debug – printf() debugging

Printf() debugging

1. How to

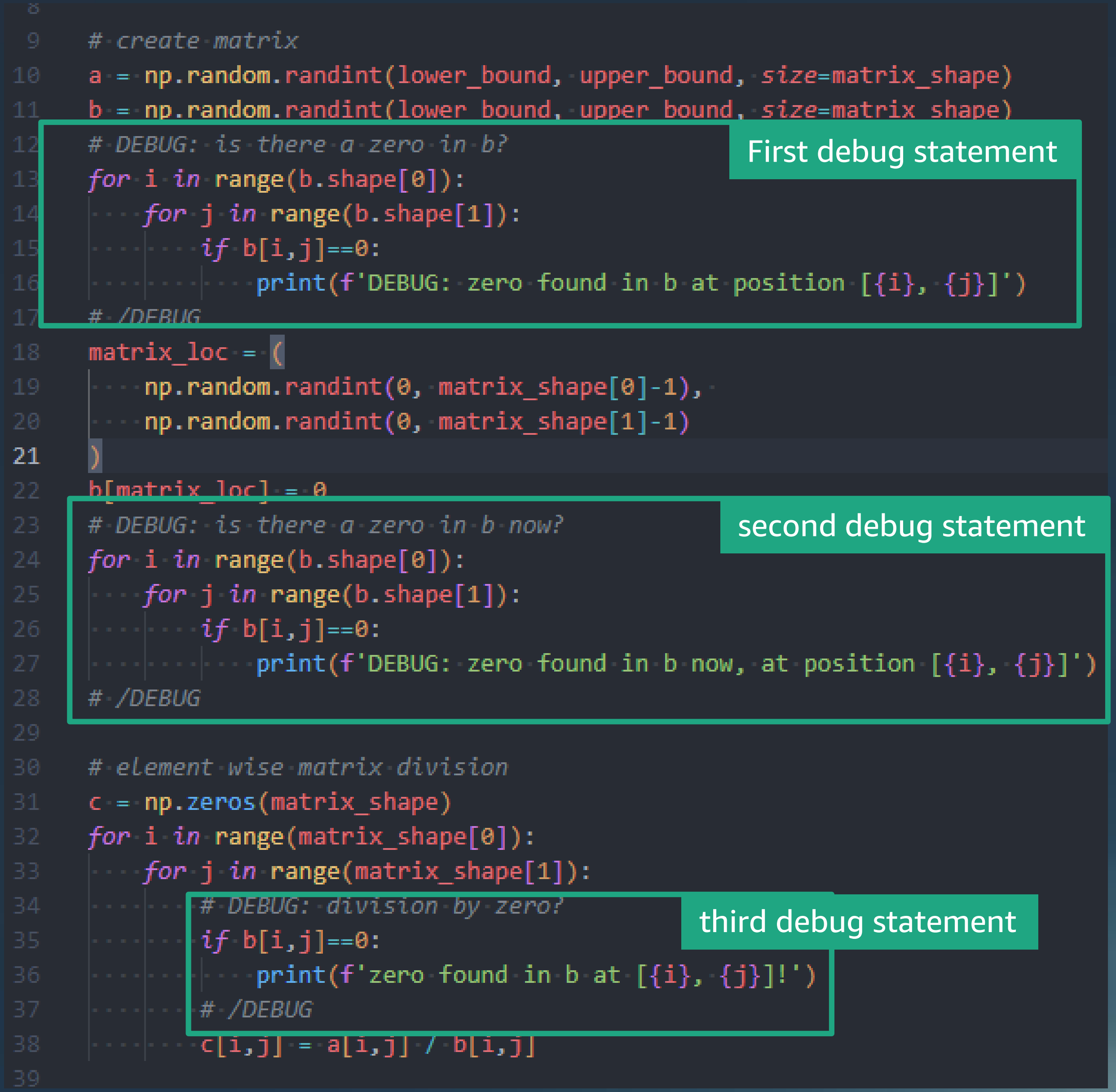
- Debugging by adding print statement to the code and identify which print output is not correct

2. Pros

- Relatively simple to implement

2. Cons

- Requires typing non necessary code
- Clean up required after debugging
- No interaction with the execution
- No debugging access to installed library



```python
# create matrix
a = np.random.randint(lower_bound, upper_bound, size=matrix_shape)
b = np.random.randint(lower_bound, upper_bound, size=matrix_shape)
# DEBUG: is there a zero in b?
for i in range(b.shape[0]):
    for j in range(b.shape[1]):
        if b[i,j]==0:
            print(f'DEBUG: zero found in b at position [{i}, {j}]')
# /DEBUG
matrix_loc = (
    np.random.randint(0, matrix_shape[0]-1),
    np.random.randint(0, matrix_shape[1]-1)
)
b[matrix_loc] = 0
# DEBUG: is there a zero in b now?
for i in range(b.shape[0]):
    for j in range(b.shape[1]):
        if b[i,j]==0:
            print(f'DEBUG: zero found in b now, at position [{i}, {j}]')
# /DEBUG

# element wise matrix division
c = np.zeros(matrix_shape)
for i in range(matrix_shape[0]):
    for j in range(matrix_shape[1]):
        # DEBUG: division by zero?
        if b[i,j]==0:
            print(f'zero found in b at [{i}, {j}]!')
        # /DEBUG
        c[i,j] = a[i,j] / b[i,j]
```

How to debug – debugger

Debugger

What is a debugger?

A debugger is a program used to analyze the execution and debug another target program. A debugger generally can:

- Execute the code line by line (important for compiled language)
- Halt the target program at user demand or under condition (e.g. at a raised exception)
- Display memory content and modify it

How to debug – debugger

Debugger

What is a debugger?

A debugger is a program used to analyze the execution and debug another target program. A debugger generally can:

- Execute the code line by line (important for compiled language)
- Halt the target program add user demand or under condition (e.g. at a raised exception)
- Display memory content and modify it

How does it work?

- A debugger compiles (if required) and runs the target code on an Instruction Set Simulator (ISS) which allows halt and analysis
- ISS is complex for complied language (e.g. C or Rust), but much simpler for scripting language (e.g. Python or Typescript)

How to debug – debugger

Debugger

What is a debugger?

A debugger is a program used to analyze the execution and debug another target program. A debugger generally can:

- Execute the code line by line (important for compiled language)
- Halt the target program add user demand or under condition (e.g. at a raised exception)
- Display memory content and modify it

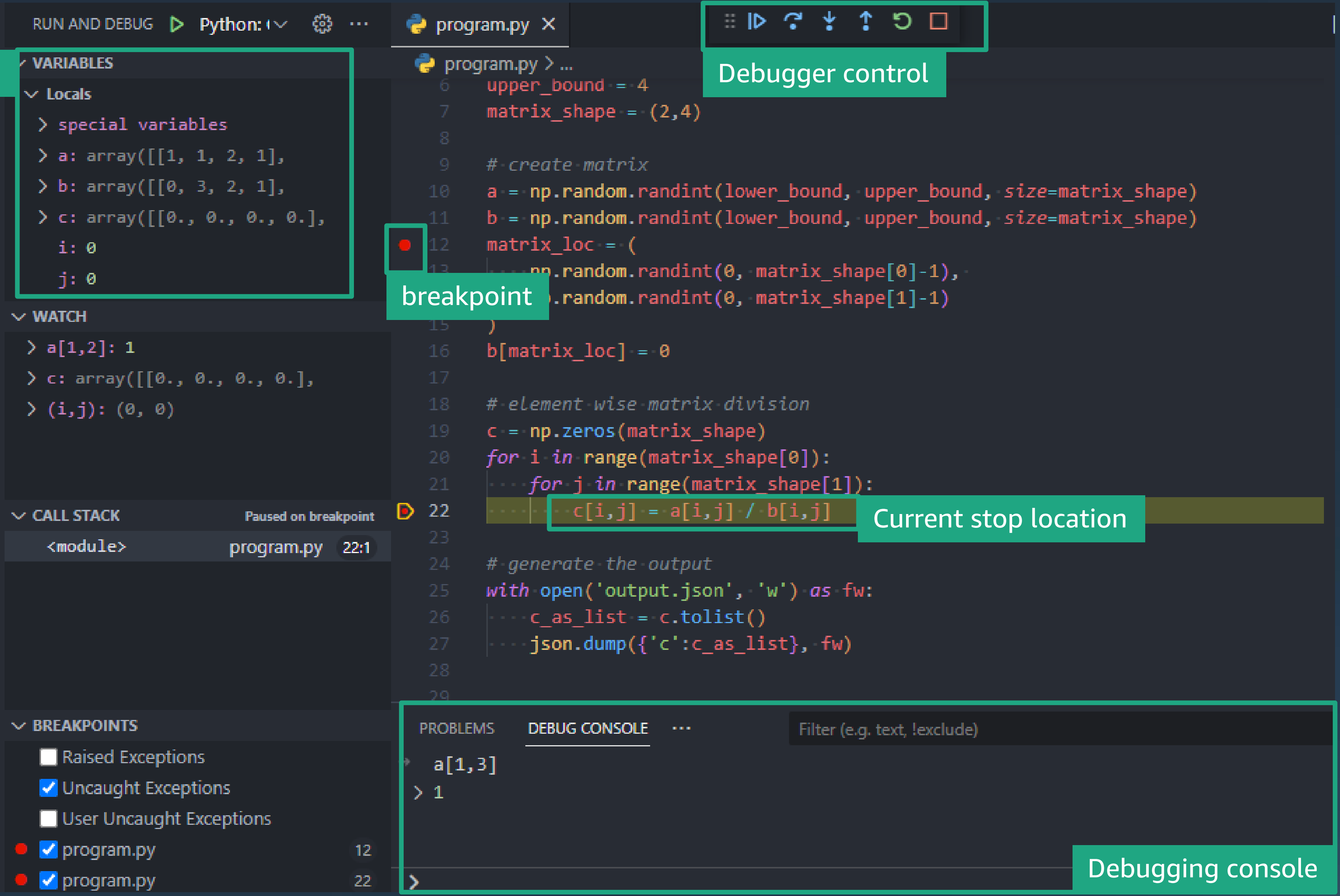
How does it work?

- A debugger compiles (if required) and runs the target code on an Instruction Set Simulator (ISS) which allows halt and analysis
- ISS is complex for complied language (e.g. C or Rust), but much simpler for scripting language (e.g. Python or Typescript)

How to use it?

- Debugger program is overwhelmingly used within an Integrated Development Environment (IDE) such as Visual Studio Code, PyCharm or Vim
- In rare instances, debugger program can be in a Command Line Interface (CLI)

Anatomy of an IDE debugger



Variables

```
special variables
a: array([[1, 1, 2, 1],
b: array([[0, 3, 2, 1],
c: array([[0., 0., 0., 0.],
i: 0
j: 0
```

Watch

```
a[1,2]: 1
c: array([[0., 0., 0., 0.],
(i,j): (0, 0)
```

Call stack

```
<module>    program.py 22:1
```

Breakpoints

- □ Raised Exceptions
- ☑ Uncaught Exceptions
- □ User Uncaught Exceptions
- ● ☑ program.py 12
- ● ☑ program.py 22

Debugger control

breakpoint

Current stop location

```python
upper_bound = 4
matrix_shape = (2,4)

# create matrix
a = np.random.randint(lower_bound, upper_bound, size=matrix_shape)
b = np.random.randint(lower_bound, upper_bound, size=matrix_shape)
matrix_loc = (
    np.random.randint(0, matrix_shape[0]-1),
    .random.randint(0, matrix_shape[1]-1)
)
b[matrix_loc] = 0

# element wise matrix division
c = np.zeros(matrix_shape)
for i in range(matrix_shape[0]):
    for j in range(matrix_shape[1]):
        c[i,j] = a[i,j] / b[i,j]

# generate the output
with open('output.json', 'w') as fw:
    c_as_list = c.tolist()
    json.dump({'c':c_as_list}, fw)
```

Debugging console

```
a[1,3]
> 1
>
```

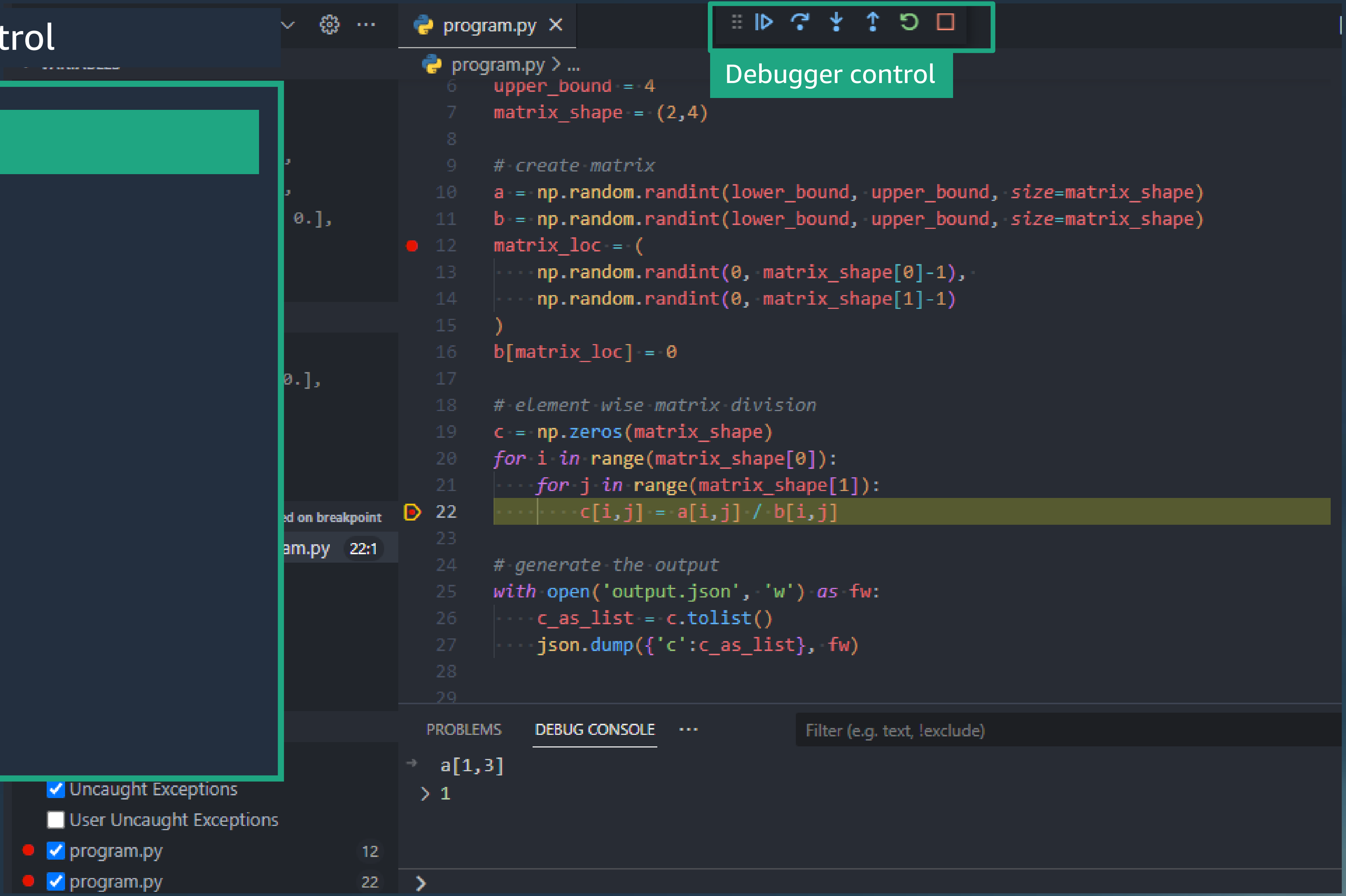
Anatomy of an IDE debugger

Debugger control

What is it?

The debugger control allows you to:

- Execute to the next breakpoint
- Execute the next line of code
- Jump inside the function



```python
upper_bound = 4
matrix_shape = (2,4)

# create matrix
a = np.random.randint(lower_bound, upper_bound, size=matrix_shape)
b = np.random.randint(lower_bound, upper_bound, size=matrix_shape)
matrix_loc = (
    np.random.randint(0, matrix_shape[0]-1),
    np.random.randint(0, matrix_shape[1]-1)
)
b[matrix_loc] = 0

# element wise matrix division
c = np.zeros(matrix_shape)
for i in range(matrix_shape[0]):
    for j in range(matrix_shape[1]):
        c[i,j] = a[i,j] / b[i,j]

# generate the output
with open('output.json', 'w') as fw:
    c_as_list = c.tolist()
    json.dump({'c':c_as_list}, fw)
```

```
a[1,3]
> 1
```

☑ Uncaught Exceptions
□ User Uncaught Exceptions
● ☑ program.py 12
● ☑ program.py 22

Anatomy of an IDE debugger

Debugger control

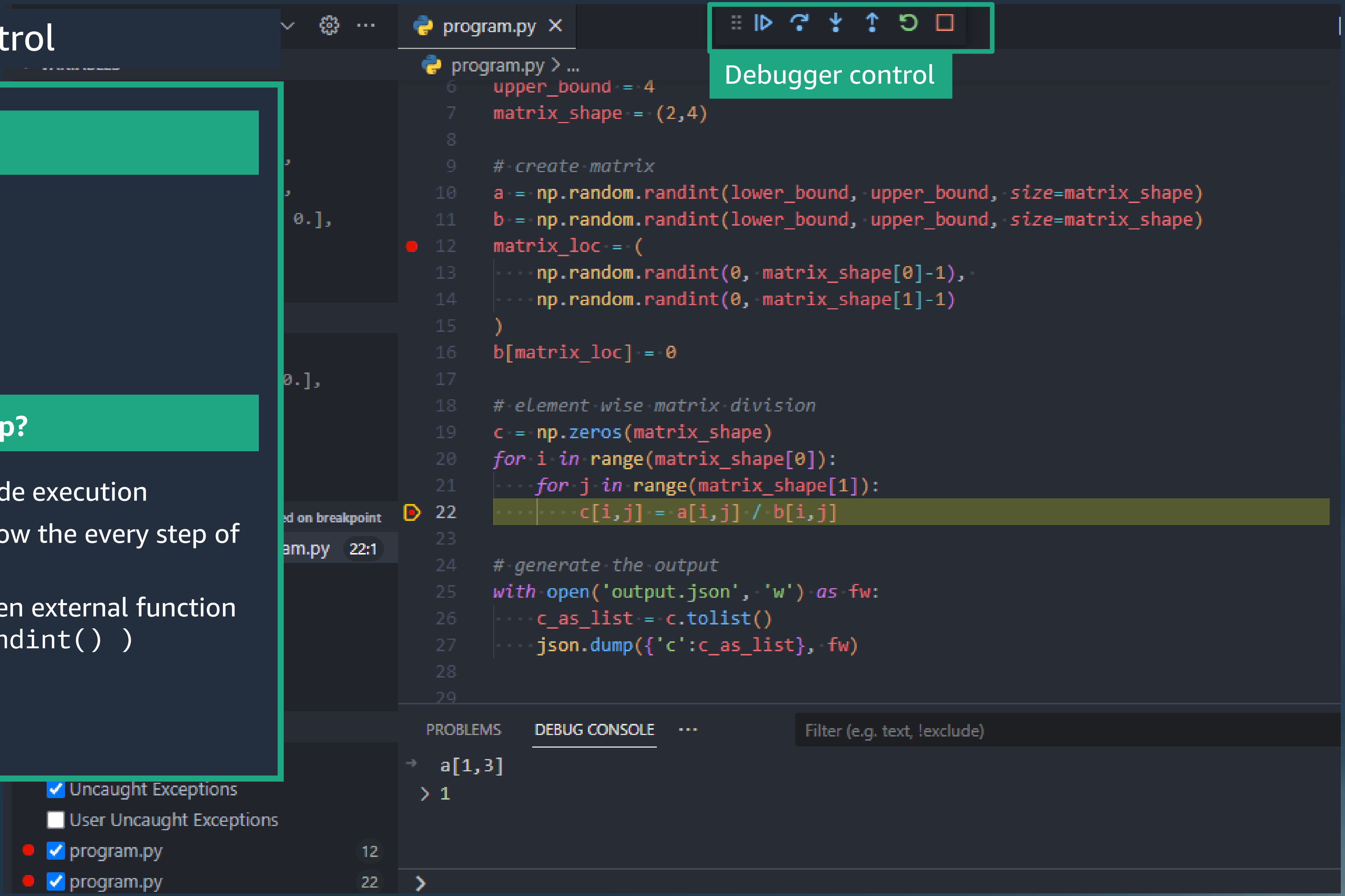
What is it?

The debugger control allows you to:

- Execute to the next breakpoint
- Execute the next line of code
- Jump inside the function

How does it help?

- It provides fine control of the code execution
- It allows the programmer to follow the every step of the execution
- It allow jumping in functions, even external function (e.g. jumping in `np.random.randint( )`)

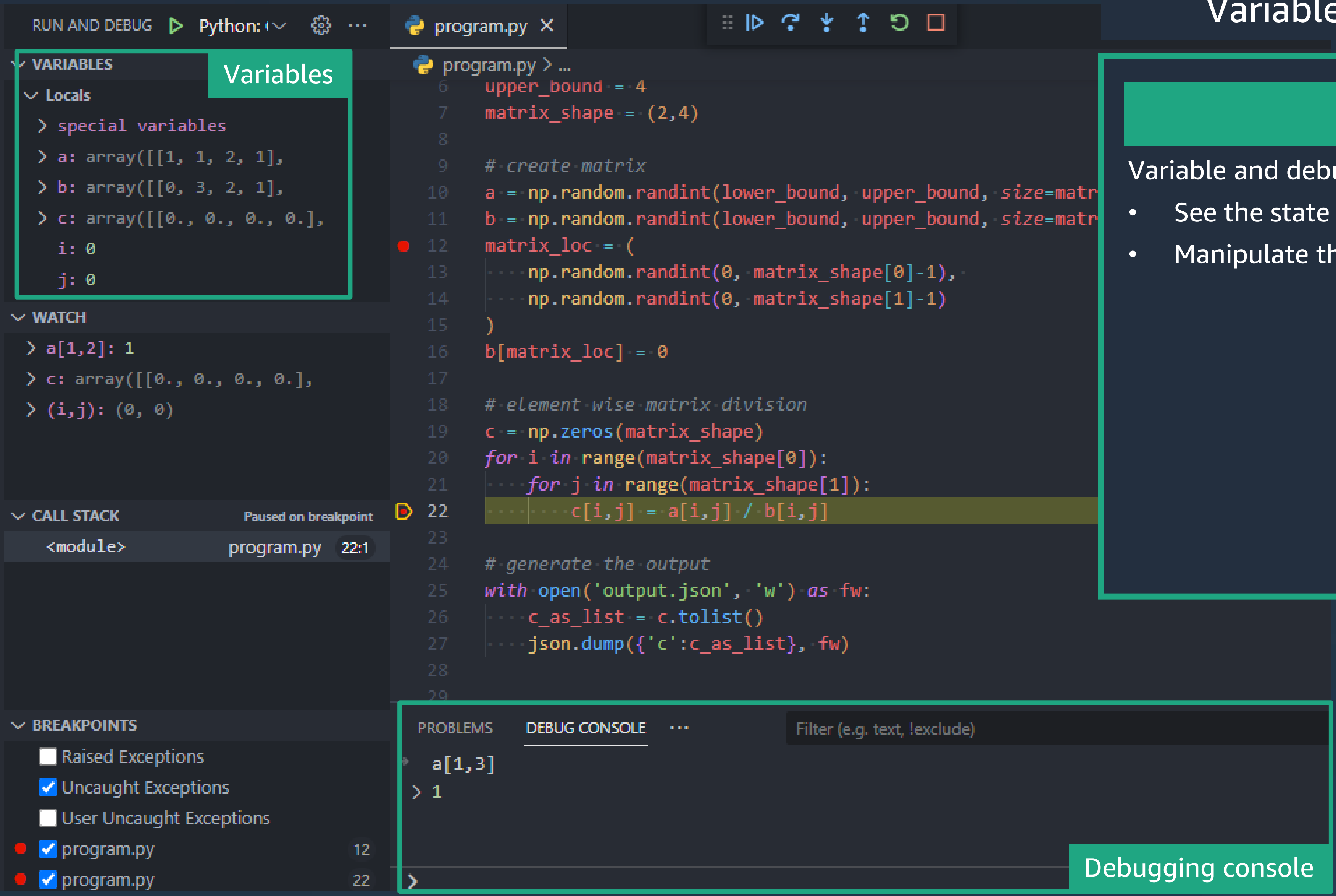


```python
upper_bound = 4
matrix_shape = (2,4)

# create matrix
a = np.random.randint(lower_bound, upper_bound, size=matrix_shape)
b = np.random.randint(lower_bound, upper_bound, size=matrix_shape)
matrix_loc = (
    np.random.randint(0, matrix_shape[0]-1),
    np.random.randint(0, matrix_shape[1]-1)
)
b[matrix_loc] = 0

# element wise matrix division
c = np.zeros(matrix_shape)
for i in range(matrix_shape[0]):
    for j in range(matrix_shape[1]):
        c[i,j] = a[i,j] / b[i,j]

# generate the output
with open('output.json', 'w') as fw:
    c_as_list = c.tolist()
    json.dump({'c':c_as_list}, fw)
```

```
a[1,3]
> 1
```

- ☑ Uncaught Exceptions
- □ User Uncaught Exceptions
- ☑ program.py 12
- ☑ program.py 22

Anatomy of an IDE debugger



Variables

```
VARIABLES
Locals
  > special variables
  > a: array([[1, 1, 2, 1],
  > b: array([[0, 3, 2, 1],
  > c: array([[0., 0., 0., 0.],
    i: 0
    j: 0
```

```
WATCH
  > a[1,2]: 1
  > c: array([[0., 0., 0., 0.],
  > (i,j): (0, 0)
```

```
CALL STACK                Paused on breakpoint
  <module>                program.py   22:1
```

```
BREAKPOINTS
  ☐ Raised Exceptions
  ☑ Uncaught Exceptions
  ☐ User Uncaught Exceptions
● ☑ program.py                12
● ☑ program.py                22
```

```python
upper_bound = 4
matrix_shape = (2,4)

# create matrix
a = np.random.randint(lower_bound, upper_bound, size=matr
b = np.random.randint(lower_bound, upper_bound, size=matr
matrix_loc = (
    np.random.randint(0, matrix_shape[0]-1),
    np.random.randint(0, matrix_shape[1]-1)
)
b[matrix_loc] = 0

# element wise matrix division
c = np.zeros(matrix_shape)
for i in range(matrix_shape[0]):
    for j in range(matrix_shape[1]):
        c[i,j] = a[i,j] / b[i,j]

# generate the output
with open('output.json', 'w') as fw:
    c_as_list = c.tolist()
    json.dump({'c':c_as_list}, fw)
```

```
a[1,3]
> 1
```

Debugging console

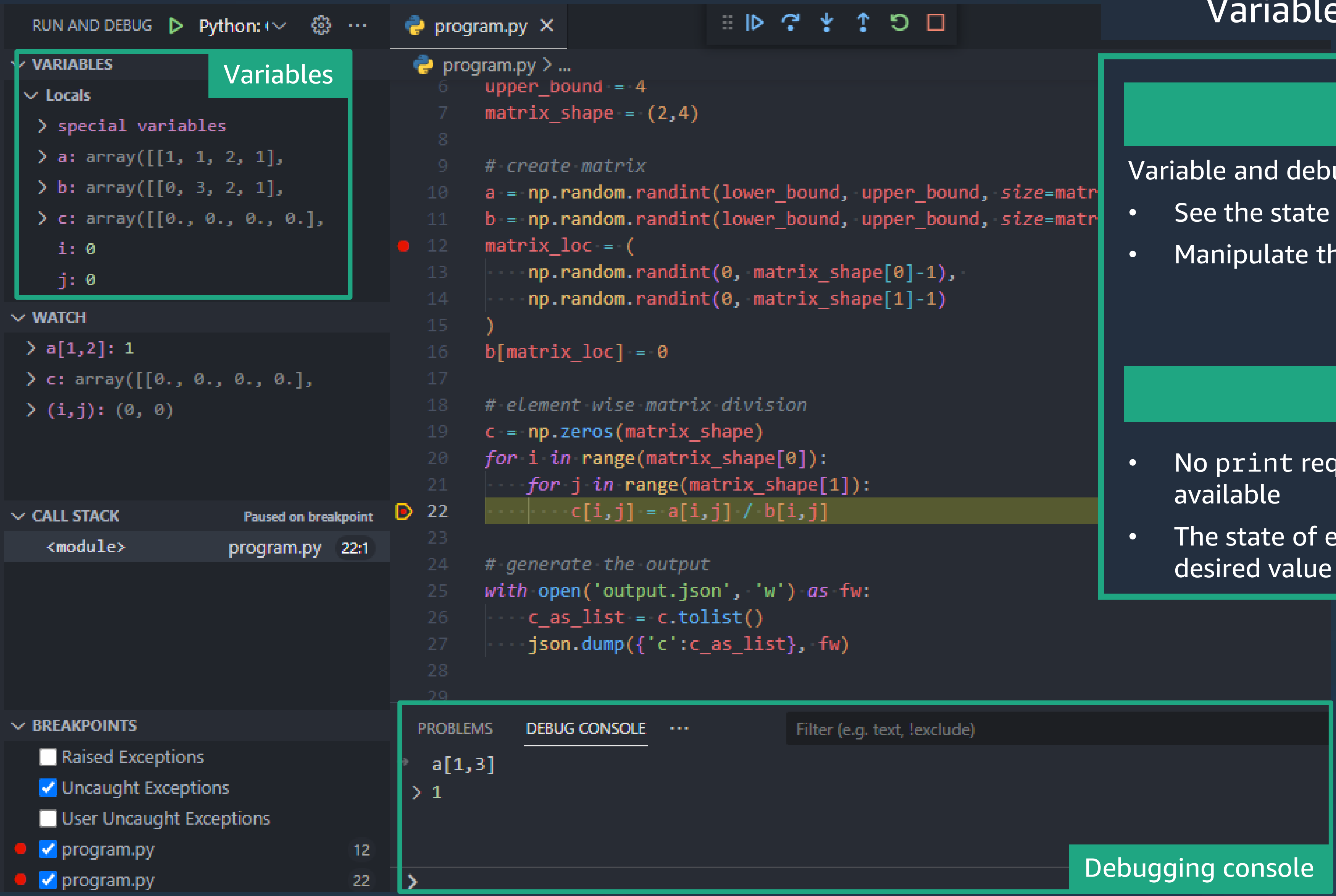
Variable and debugging console

What is it?

Variable and debugging console allows you to:

- See the state of each variable
- Manipulate the value of each variable

Anatomy of an IDE debugger



VARIABLES

Locals

```
> special variables
> a: array([[1, 1, 2, 1],
> b: array([[0, 3, 2, 1],
> c: array([[0., 0., 0., 0.],
  i: 0
  j: 0
```

WATCH

```
> a[1,2]: 1
> c: array([[0., 0., 0., 0.],
> (i,j): (0, 0)
```

CALL STACK — Paused on breakpoint

```
<module>    program.py  22:1
```

BREAKPOINTS

- ☐ Raised Exceptions
- ☑ Uncaught Exceptions
- ☐ User Uncaught Exceptions
- ● ☑ program.py 12
- ● ☑ program.py 22

```
upper_bound = 4
matrix_shape = (2,4)

# create matrix
a = np.random.randint(lower_bound, upper_bound, size=matr
b = np.random.randint(lower_bound, upper_bound, size=matr
matrix_loc = (
    np.random.randint(0, matrix_shape[0]-1),
    np.random.randint(0, matrix_shape[1]-1)
)
b[matrix_loc] = 0

# element wise matrix division
c = np.zeros(matrix_shape)
for i in range(matrix_shape[0]):
    for j in range(matrix_shape[1]):
        c[i,j] = a[i,j] / b[i,j]

# generate the output
with open('output.json', 'w') as fw:
    c_as_list = c.tolist()
    json.dump({'c':c_as_list}, fw)
```

```
a[1,3]
> 1
>
```

Variable and debugging console

What is it?

Variable and debugging console allows you to:

- See the state of each variable
- Manipulate the value of each variable

How does it help?

- No `print` required: the state of each variable is available
- The state of each variable can be manipulated to a desired value (real-time alteration of the execution)

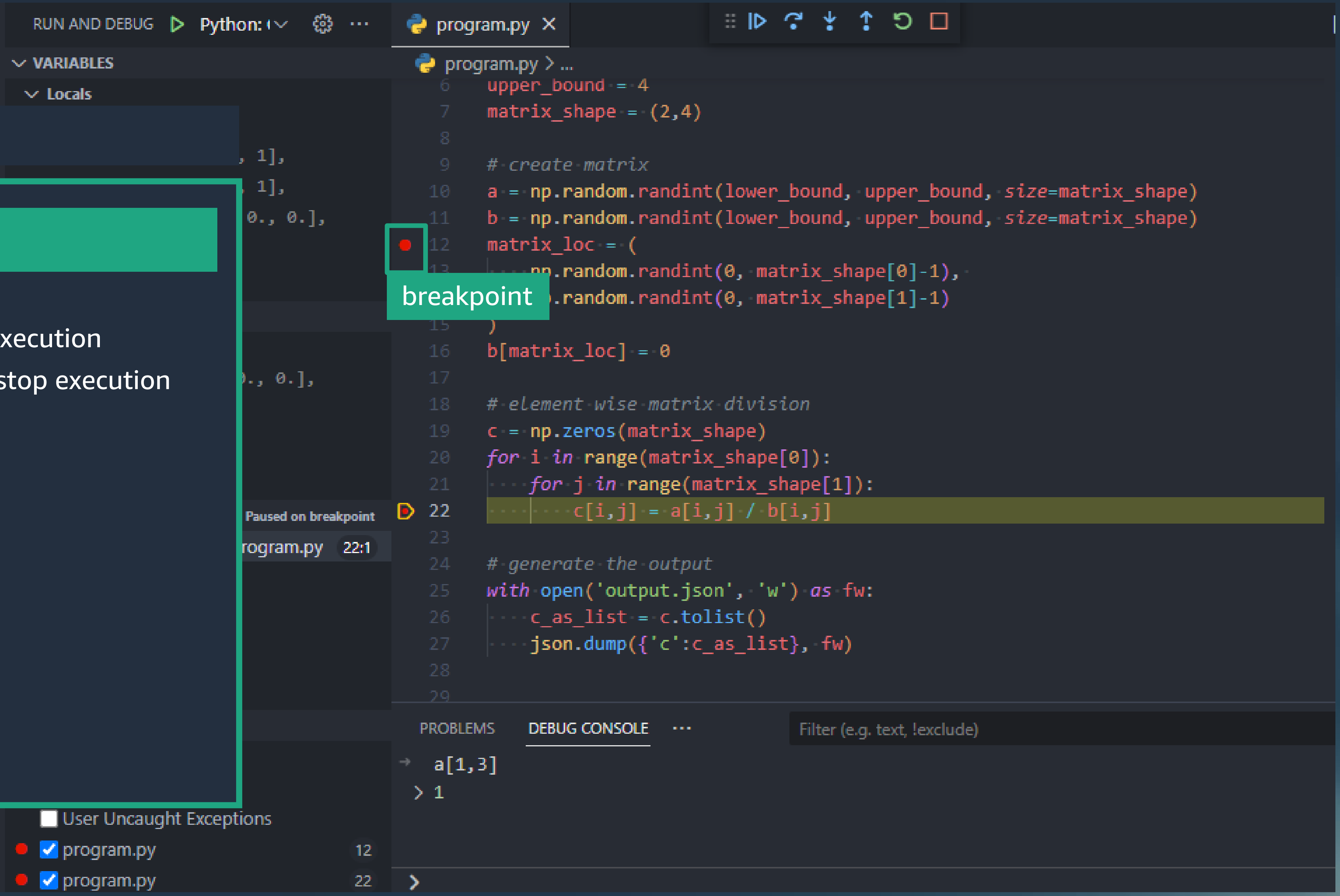
Anatomy of an IDE debugger

Breakpoint

What is it?

Breakpoints allow you to:

- To define at which line to stop the execution
- To define under which condition to stop execution



```
, 1],
, 1],
0., 0.],
)., 0.],
```

```python
upper_bound = 4
matrix_shape = (2,4)

# create matrix
a = np.random.randint(lower_bound, upper_bound, size=matrix_shape)
b = np.random.randint(lower_bound, upper_bound, size=matrix_shape)
matrix_loc = (
    np.random.randint(0, matrix_shape[0]-1),
    .random.randint(0, matrix_shape[1]-1)
)
b[matrix_loc] = 0

# element wise matrix division
c = np.zeros(matrix_shape)
for i in range(matrix_shape[0]):
    for j in range(matrix_shape[1]):
        c[i,j] = a[i,j] / b[i,j]

# generate the output
with open('output.json', 'w') as fw:
    c_as_list = c.tolist()
    json.dump({'c':c_as_list}, fw)
```

```
a[1,3]
> 1
```

□ User Uncaught Exceptions

☑ program.py 12

☑ program.py 22

Anatomy of an IDE debugger

Breakpoint

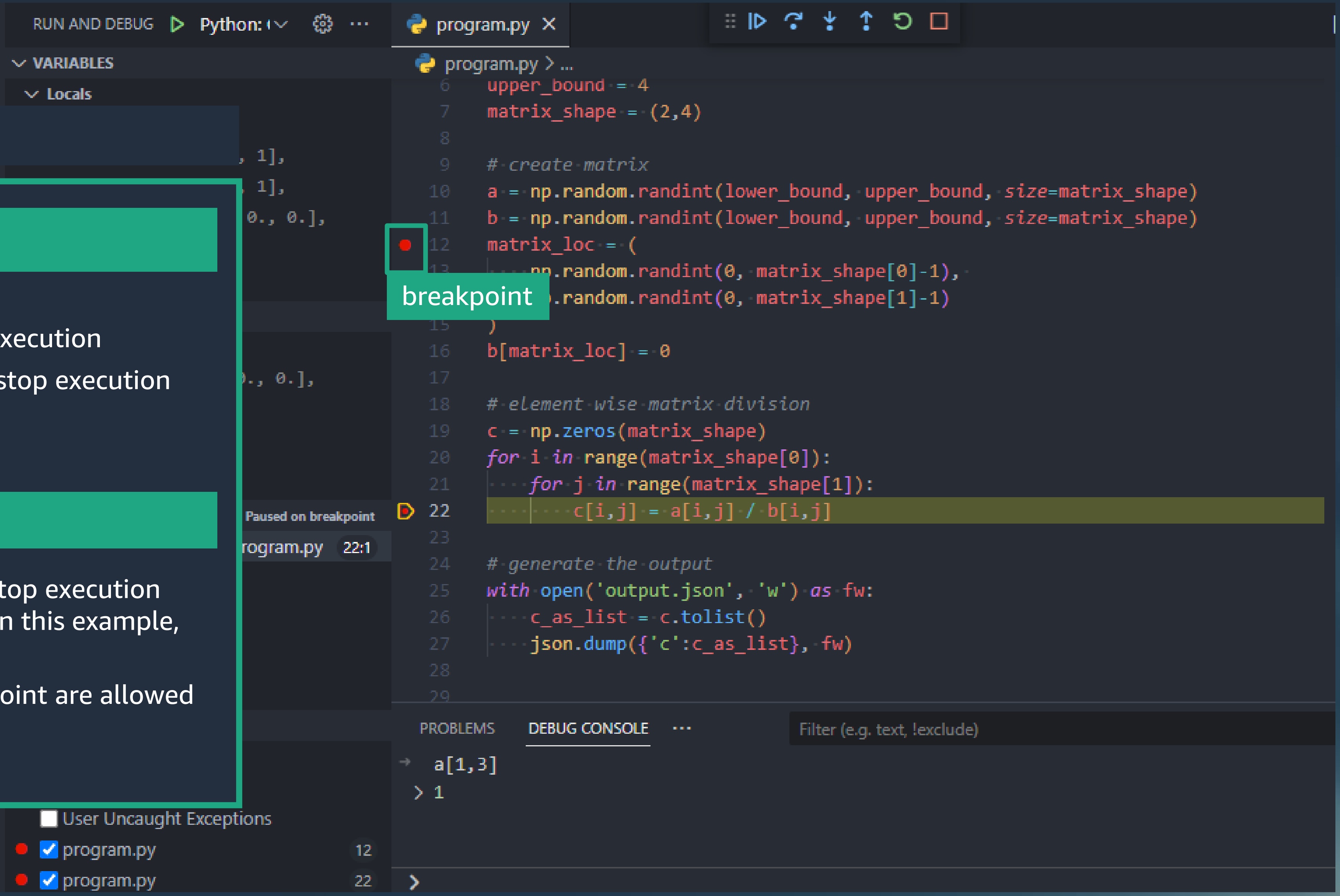
What is it?

Breakpoints allow you to:

- To define at which line to stop the execution
- To define under which condition to stop execution

How does it help?

- Condition breakpoint allow you to stop execution when a critical moment is reached. In this example, when `b[i,j]==0`
- Dynamic execution: multiple breakpoint are allowed and can be added during execution



```
, 1],
, 1],
0., 0.],
)., 0.],
```

```
upper_bound = 4
matrix_shape = (2,4)

# create matrix
a = np.random.randint(lower_bound, upper_bound, size=matrix_shape)
b = np.random.randint(lower_bound, upper_bound, size=matrix_shape)
matrix_loc = (
    np.random.randint(0, matrix_shape[0]-1),
    .random.randint(0, matrix_shape[1]-1)
)
b[matrix_loc] = 0

# element wise matrix division
c = np.zeros(matrix_shape)
for i in range(matrix_shape[0]):
    for j in range(matrix_shape[1]):
        c[i,j] = a[i,j] / b[i,j]

# generate the output
with open('output.json', 'w') as fw:
    c_as_list = c.tolist()
    json.dump({'c':c_as_list}, fw)
```

breakpoint

Paused on breakpoint

rogram.py 22:1

☐ User Uncaught Exceptions

● ☑ program.py 12

● ☑ program.py 22

```
a[1,3]
> 1
```

Debugging a library

How to debug a library

Why debugging a library is different?

- A debugger can only run on a target program
- A library is not a program: it is a collection of statements (e.g. function class) which can be used in a program

Debugging a library

How to debug a library

Why debugging a library is different?

- A debugger can only run on a target program
- A library is not a program: it is a collection of statements (e.g. function class) which can be used in a program

How to not debug a library

- Write a specific program with the only purpose to be executed within a debugger (this not better than printf() debugging)

Debugging a library

How to debug a library

Why debugging a library is different?

- A debugger can only run on a target program
- A library is not a program: it is a collection of statements (e.g. function class) which can be used in a program

How to not debug a library

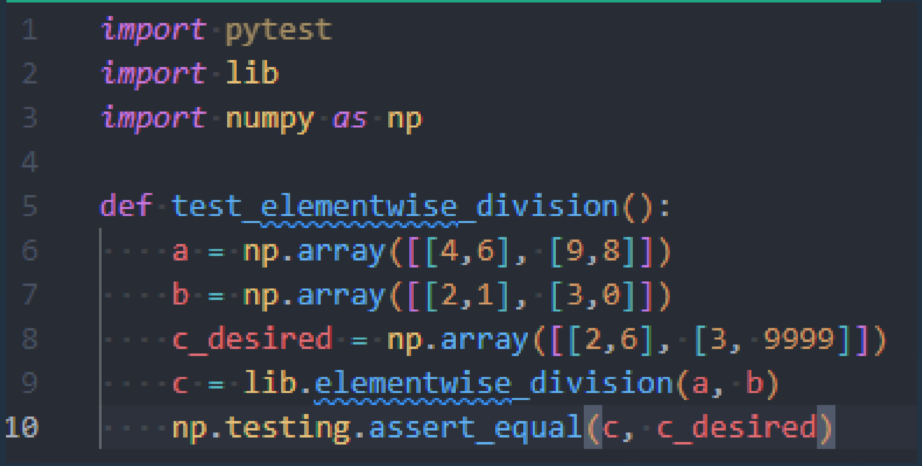
- Write a specific program with the only purpose to be executed within a debugger (this not better than printf() debugging)

How to debug a library

- Use unit tests as the target program for the debugger
- Use breakpoints in the test code to jump into the library function to debug

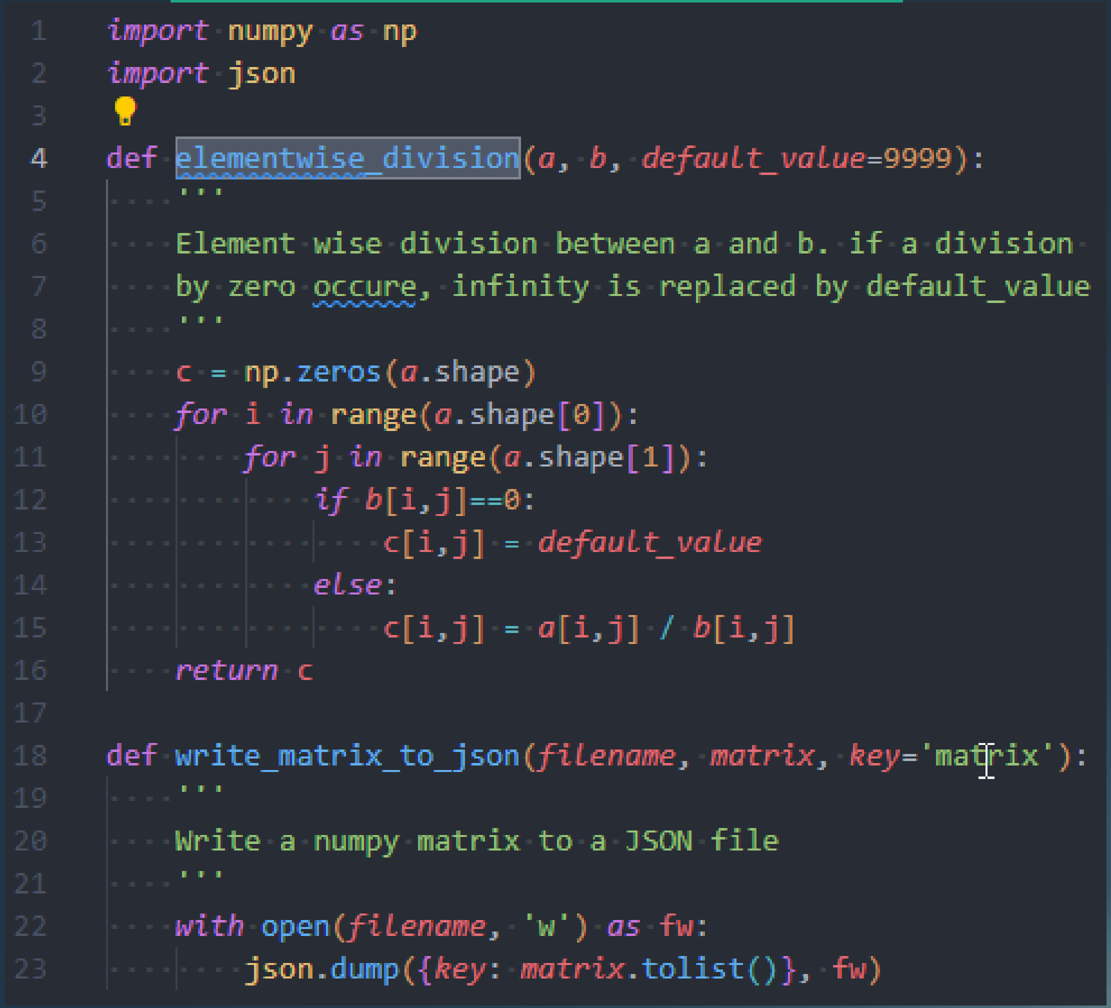
Refactoring our example into a library

Test_lib.py
Unit test for the `elementwise_division` function



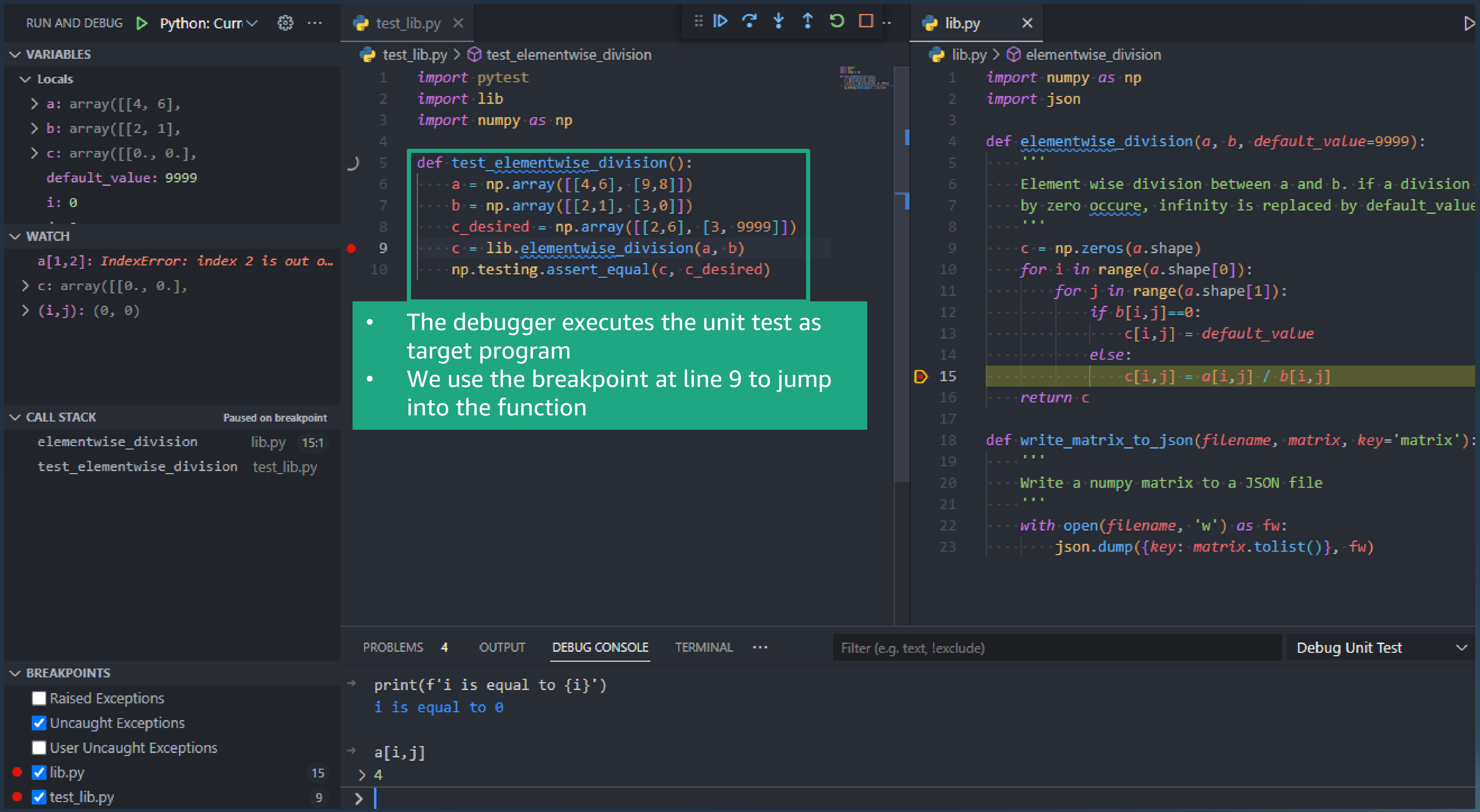
```python
import pytest
import lib
import numpy as np

def test_elementwise_division():
    a = np.array([[4,6], [9,8]])
    b = np.array([[2,1], [3,0]])
    c_desired = np.array([[2,6], [3, 9999]])
    c = lib.elementwise_division(a, b)
    np.testing.assert_equal(c, c_desired)
```

lib.py
`elementwise_division` implementation



```python
import numpy as np
import json

def elementwise_division(a, b, default_value=9999):
    '''
    Element wise division between a and b. if a division
    by zero occure, infinity is replaced by default_value
    '''
    c = np.zeros(a.shape)
    for i in range(a.shape[0]):
        for j in range(a.shape[1]):
            if b[i,j]==0:
                c[i,j] = default_value
            else:
                c[i,j] = a[i,j] / b[i,j]
    return c

def write_matrix_to_json(filename, matrix, key='matrix'):
    '''
    Write a numpy matrix to a JSON file
    '''
    with open(filename, 'w') as fw:
        json.dump({key: matrix.tolist()}, fw)
```

Using a debugger with unit test



RUN AND DEBUG
Python: Curr
test_lib.py
lib.py
test_lib.py > test_elementwise_division
lib.py > elementwise_division

VARIABLES

Locals

```
> a: array([[4, 6],
> b: array([[2, 1],
> c: array([[0., 0.],
  default_value: 9999
  i: 0
```

WATCH

```
a[1,2]: IndexError: index 2 is out o…
> c: array([[0., 0.],
> (i,j): (0, 0)
```

CALL STACK — Paused on breakpoint

```
elementwise_division        lib.py  15:1
test_elementwise_division   test_lib.py
```

BREAKPOINTS

- ☐ Raised Exceptions
- ☑ Uncaught Exceptions
- ☐ User Uncaught Exceptions
- ● ☑ lib.py 15
- ● ☑ test_lib.py 9

test_lib.py

```python
1   import pytest
2   import lib
3   import numpy as np
4
5   def test_elementwise_division():
6       a = np.array([[4,6], [9,8]])
7       b = np.array([[2,1], [3,0]])
8       c_desired = np.array([[2,6], [3, 9999]])
9       c = lib.elementwise_division(a, b)
10      np.testing.assert_equal(c, c_desired)
```

- The debugger executes the unit test as target program
- We use the breakpoint at line 9 to jump into the function

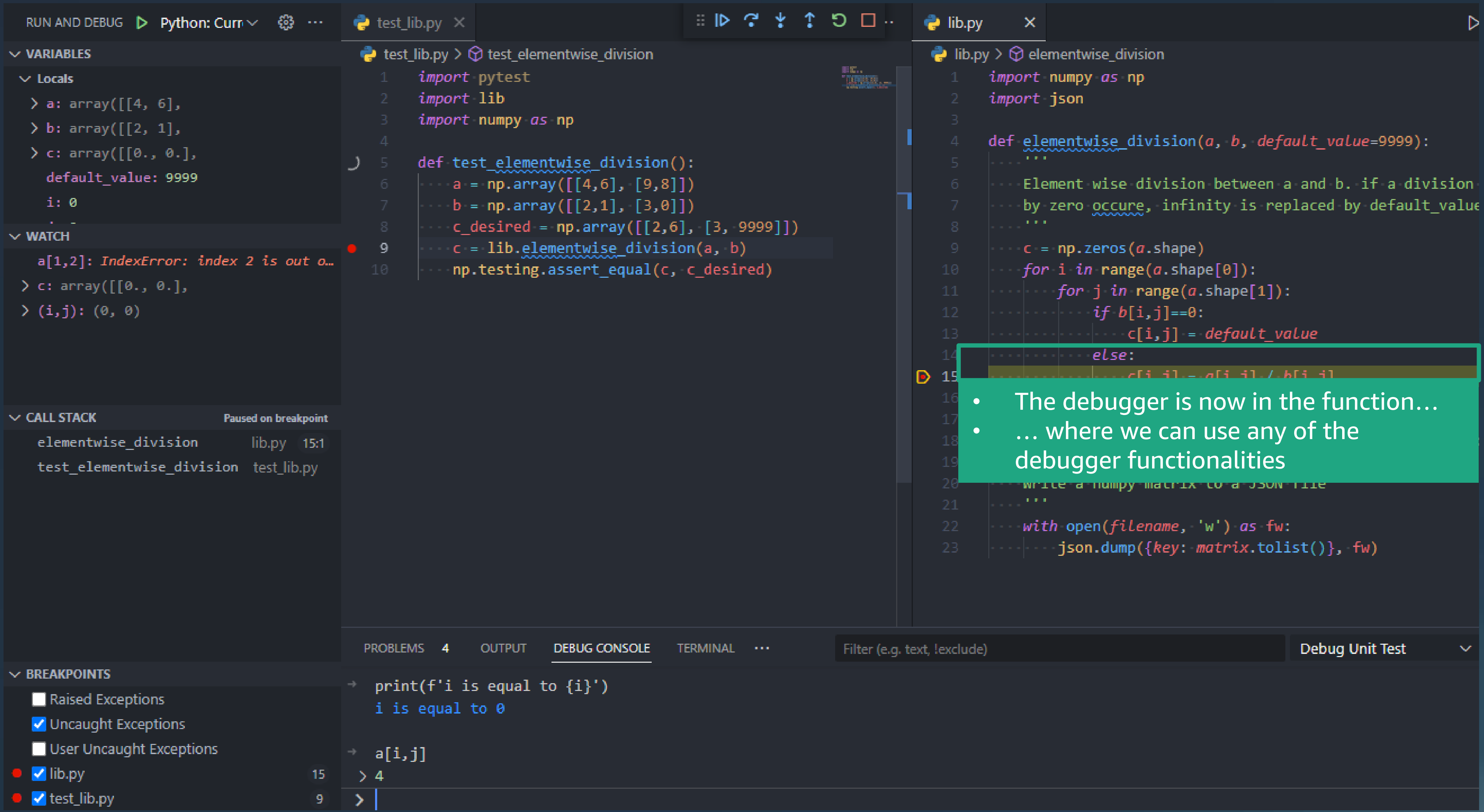
lib.py

```python
1   import numpy as np
2   import json
3
4   def elementwise_division(a, b, default_value=9999):
5       '''
6       Element wise division between a and b. if a division
7       by zero occure, infinity is replaced by default_value
8       '''
9       c = np.zeros(a.shape)
10      for i in range(a.shape[0]):
11          for j in range(a.shape[1]):
12              if b[i,j]==0:
13                  c[i,j] = default_value
14              else:
15                  c[i,j] = a[i,j] / b[i,j]
16      return c
17
18  def write_matrix_to_json(filename, matrix, key='matrix'):
19      '''
20      Write a numpy matrix to a JSON file
21      '''
22      with open(filename, 'w') as fw:
23          json.dump({key: matrix.tolist()}, fw)
```

PROBLEMS 4 | OUTPUT | DEBUG CONSOLE | TERMINAL | Filter (e.g. text, !exclude) | Debug Unit Test

```
print(f'i is equal to {i}')
i is equal to 0

a[i,j]
> 4
>
```

Using a debugger with unit test



VARIABLES

Locals

- a: array([[4, 6],
- b: array([[2, 1],
- c: array([[0., 0.],
- default_value: 9999
- i: 0

WATCH

- a[1,2]: IndexError: index 2 is out o…
- c: array([[0., 0.],
- (i,j): (0, 0)

CALL STACK — Paused on breakpoint

- elementwise_division lib.py 15:1
- test_elementwise_division test_lib.py

BREAKPOINTS

- □ Raised Exceptions
- ☑ Uncaught Exceptions
- □ User Uncaught Exceptions
- ● ☑ lib.py 15
- ● ☑ test_lib.py 9

test_lib.py > test_elementwise_division

```python
import pytest
import lib
import numpy as np

def test_elementwise_division():
    a = np.array([[4,6], [9,8]])
    b = np.array([[2,1], [3,0]])
    c_desired = np.array([[2,6], [3, 9999]])
    c = lib.elementwise_division(a, b)
    np.testing.assert_equal(c, c_desired)
```

lib.py > elementwise_division

```python
import numpy as np
import json

def elementwise_division(a, b, default_value=9999):
    '''
    Element wise division between a and b. if a division
    by zero occure, infinity is replaced by default_value
    '''
    c = np.zeros(a.shape)
    for i in range(a.shape[0]):
        for j in range(a.shape[1]):
            if b[i,j]==0:
                c[i,j] = default_value
            else:
                c[i,j] = a[i,j] / b[i,j]



    write a numpy matrix to a JSON file
    '''
    with open(filename, 'w') as fw:
        json.dump({key: matrix.tolist()}, fw)
```

- The debugger is now in the function…
- … where we can use any of the debugger functionalities

```
print(f'i is equal to {i}')
i is equal to 0

a[i,j]
> 4
>
```

First steps

How do I start debugging?

Your favorite IDE is very likely to have a visual debugger included.

If not, the debugger might be a extension or your “IDE” is actually a simple text editor

First steps

How do I start debugging?

Your favorite IDE is very likely to have a visual debugger included.

If not, the debugger might be a extension or your “IDE” is actually a simple text editor

When can I start debugging?

From the first line of code!

Debugger can be helpful to trace and debug unit test to

First steps

How do I start debugging?

Your favorite IDE is very likely to have a visual debugger included.

If not, the debugger might be a extension or your “IDE” is actually a simple text editor

When can I start debugging?

From the first line of code!

Debugger can be helpful to trace and debug unit test to

Why should I use a debugger?

Because it’s the most powerful way to debug code…

…And when used with testing, they allow writing of better and more efficient code

Closing out

Recap

Why bring software engineering practices to DS/ML?

Applying testing to DS/ML

Debugging

Visit the Data & AI/ML resource hub

Dive deeper into these resources, get inspired and learn how you can use AI and machine learning to accelerate your business outcomes.

- 6 steps to machine learning success e-book
- 7 leading machine learning use cases e-book
- Machine learning at scale e-book
- Achieving transformative business results with machine learning e-book
- Tackling our world’s hardest problems with machine learning e-book
- Accelerating machine learning innovation through security e-book
- … and more!

https://bityl.co/FqdC

Visit resource hub

AWS Training and Certification

Access the AI & ML learning plan courses built by AWS experts on AWS Skill Builder

- Get started with digital self-paced, on-demand training and ramp-up guides to help you grow your technical skills
- Learn how to apply machine learning, artificial intelligence, and deep learning to unlock new insights and value in your role
- Take the steps today, towards validating your expertise with an AWS Certified Machine Learning – Specialty Certification

https://bit.ly/3FnxDH7

Learn your way explore.skillbuilder.aws »

Thank you for attending AWS Innovate – Data & AI/ML Edition

We hope you found it interesting! A kind reminder to **complete the survey.**
Let us know what you thought of today's event and how we can improve the event experience for you in the future.

- aws-apj-marketing@amazon.com
- twitter.com/AWSCloud
- facebook.com/AmazonWebServices
- youtube.com/user/AmazonWebServices
- slideshare.net/AmazonWebServices
- twitch.tv/aws

Thank you!