

# PREEXERCISE DEBUGGING / CODE TESTING



Algorithms and Data Structures 1  
Exercise – 2023S

Markus Jäger (Computer Science)  
Florian Beck (Artificial Intelligence)  
Raja Zafar (Artificial Intelligence)

Institute of Pervasive Computing  
Johannes Kepler University Linz

markus.jaeger@jku.at  
florian.beck@jku.at  
raja.zafar@pervasive.jku.at

**JOHANNES KEPLER  
UNIVERSITÄT LINZ**  
Altenberger Straße 69  
4040 Linz, Österreich  
jku.at

# MENTAL HEALTH WEEK 23

Montag, 27. März 2023 bis

Freitag, 31. März 2023

Wo: JKU Campus

MO - Fr: ganztägig

Anmeldung &  
weitere Infos:



Das wartet kostenlos auf dich:

ca 30 Workshops und Vorträge von externen Workshop-  
leiter:innen, Professor:innen und ÖH-Referaten

# OVERVIEW

## Debugging and code testing

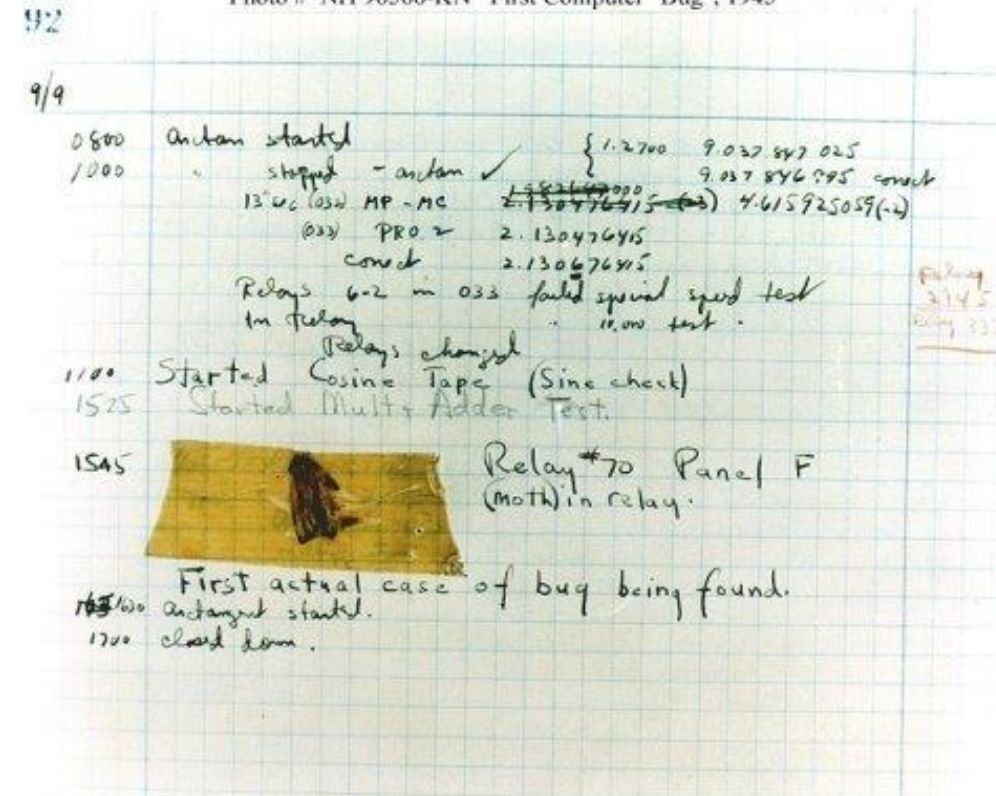
- Debugging
- How to test your code - Unit testing
- Live demo of the unit testing framework **unittest** in **PyCharm**
- Brief introduction to **exceptions**

# BUGS?

- Bug = Software Error
  - Bug = dt. „Wanze“ / „Käfer“
  - First Bug caused by a Bug: 1945 – Moth in Switch
  - Oldest known Bug: 18. November 1878,  
Letter from Thomas Edison to the Inventor  
of the Switchboard

- Debugging = using a debugger (software for error detection and correction)
  - A very powerful tool in software development and essential help for programmers when creating, optimizing and troubleshooting software
  - Possibility to check program functionality step-by-step

Photo # NH 96566-KN First Computer "Bug", 1945



# DEBUGGING / CODE TESTING

Code testing is important to make sure your code is doing what you expect it to do.

There are different approaches:

- Using dedicated debug outputs to e.g., print to terminal (--> cross-platform development)
- Execute code (with any parameters) and analyse result using debugger/breakpoints step by step
- If you have a class without main(), write a test class around it to execute the methods
- Make use of unit test frameworks
- ...

# DEBUGGING / CODE TESTING :: EXAMPLE

**Task:** Write a class Calculator with a method sum\_positive(), that

- takes two arguments a and b which must be integer numbers
- calculates and returns the sum of two positive integer values a and b or
- returns -1 if at least one parameter is negative.

Additionally, store the last computed value in a variable

```
class Calculator:
    last_value = 0

    def sum_positive(self, a, b):
        if a >= 0 and b >= 0:
            self.last_value = a + b
            return a + b
        else:
            self.last_value = -1
            return -1
```

# DEBUGGING / CODE TESTING :: **EXAMPLE**

How can we test that code?

```
class Calculator:
    last_value = 0

    def sum_positive(self, a, b):
        if a >= 0 and b >= 0:
            self.last_value = a + b
            return a + b
        else:
            self.last_value = -1
            return -1
```

Make it  
executable...

```
class Calculator:
    last_value = 0

    def sum_positive(self, a, b):
        if a >= 0 and b >= 0:
            self.last_value = a + b
            return a + b
        else:
            self.last_value = -1
            return -1

calc = Calculator()
print(calc.sum_positive(1, 2))
```

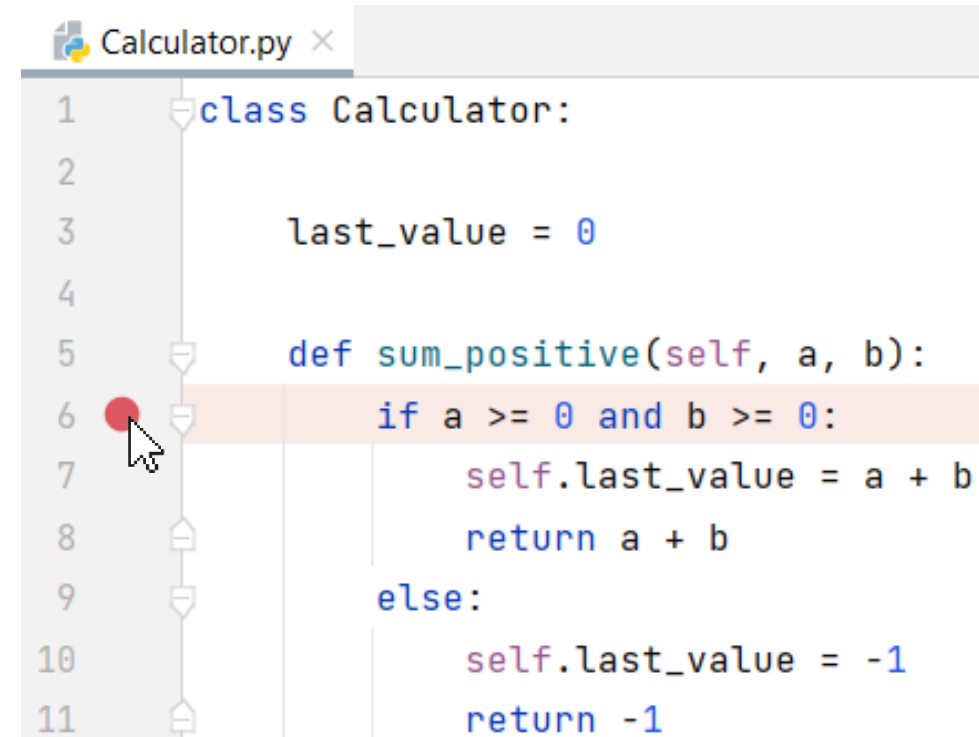
# DEBUGGING / CODE TESTING :: **EXAMPLE**

Create a breakpoint in PyCharm

1. with a left-click next to the code line number
2. using the shortcut *ctrl+8* to toggle a breakpoint on the selected line
3. In the menu:  
'Run'-'>'Toggle Breakpoint'-'>'Line Breakpoint'

A red bullet appears next to the line number

→ Breakpoint set



The screenshot shows a PyCharm IDE window titled 'Calculator.py'. The code is as follows:

```
1 class Calculator:
2
3     last_value = 0
4
5     def sum_positive(self, a, b):
6         if a >= 0 and b >= 0:
7             self.last_value = a + b
8             return a + b
9         else:
10            self.last_value = -1
11            return -1
```

A red bullet breakpoint is set on line 6, next to the line number in the left margin. A mouse cursor is hovering over the bullet. The line containing the breakpoint is highlighted in light red.



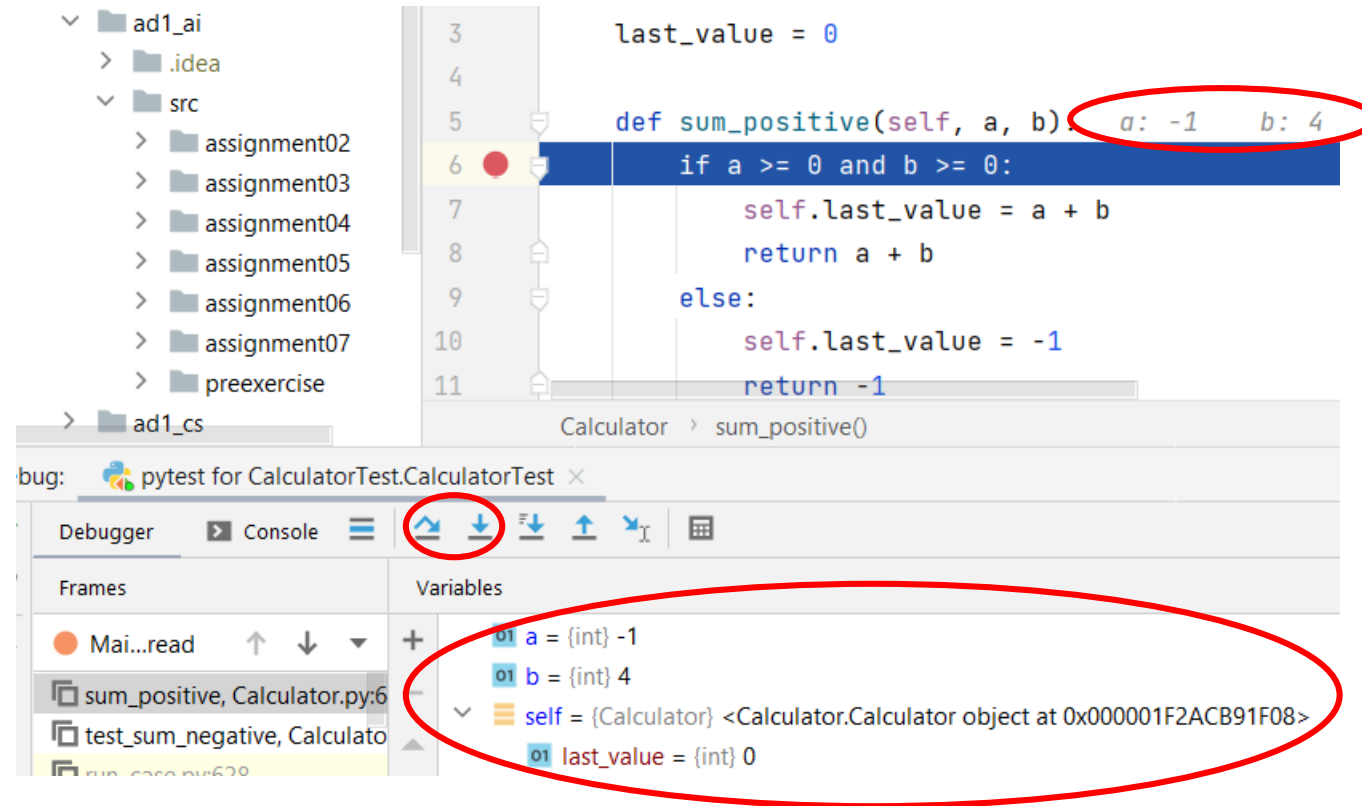
# DEBUGGING / CODE TESTING :: EXAMPLE

Click the Debug-Button  
or create a specific *Debug Configuration*



Program execution will stop at the first breakpoint:

- Use F7 (Step Into) or F8 (Step Over) to execute the highlighted line
- In the “Variables-Box” at the bottom, you see variables used in the current context including their values



# UNIT TESTING

## One definition:

*A unit test is an automated piece of code that invokes another piece of code (the unit of work being tested) and then checks the correctness of some assumptions about an end result of that code. If the assumptions turn out to be wrong, the unit test failed.*

*(slightly adapted from Roy Osherove, "The Art of Unit Testing")*

Unit tests are used in software projects to automate testing of the **behaviour of software** components, apart from its use in a full (complex) software system.

Do not care about implementation details.

A **unit** in our understanding can be e.g. an **interface/class** but also a **single method**.

# UNIT TESTING

**Properties** of good unit tests:

- Full access/control of the unit under test
- Should run quickly
- Should be consistent in its results
- Should be fully isolated (run independently of other tests)
- Can run automatically and repeatable without need of configuration or other resources.  
(in contrast to integration testing)

## Approach

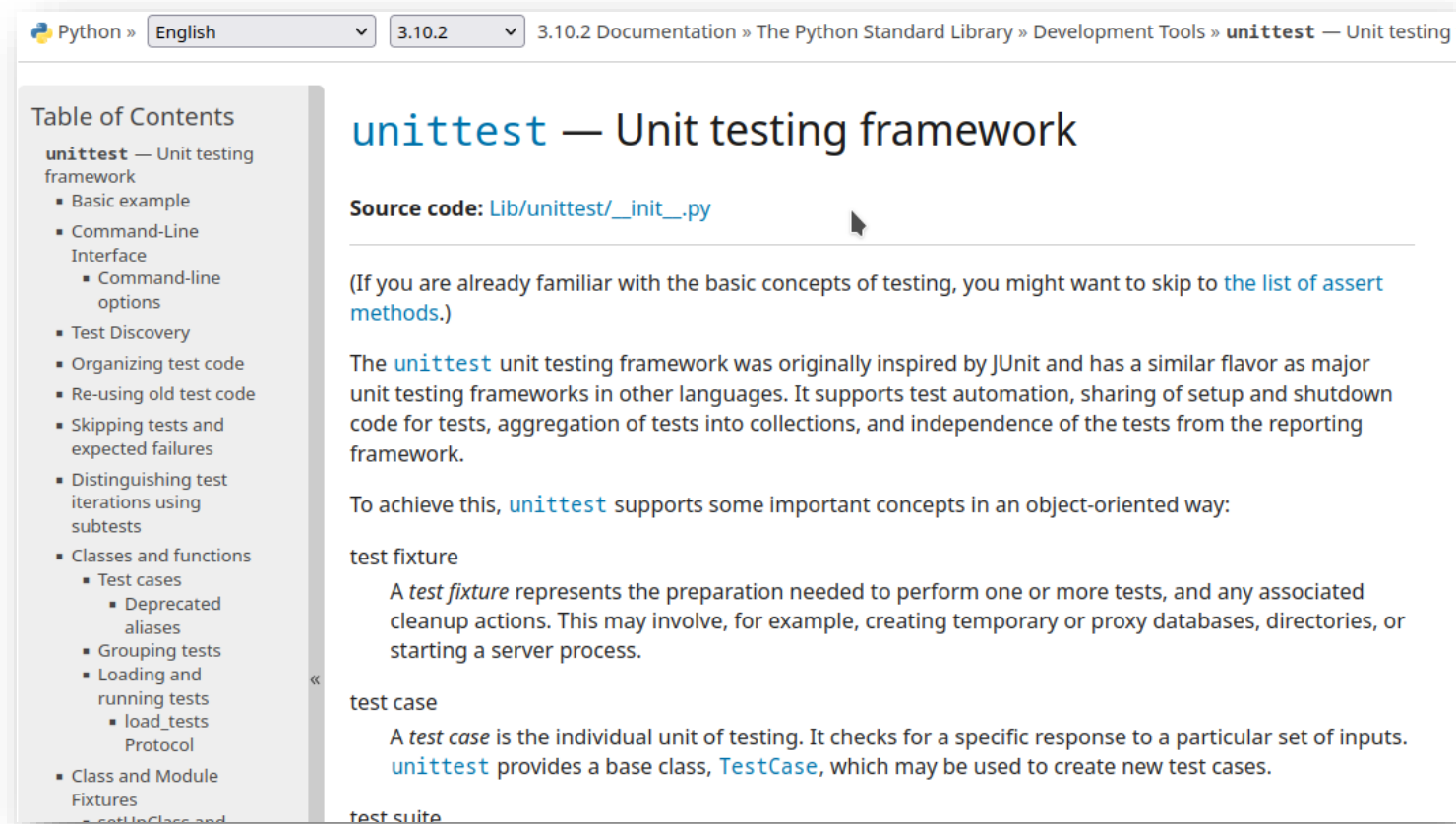
1. Initialisation of the required initial state of the software component to start testing.
2. Execution of the piece of code to be tested on that state.
3. The actual result is compared to the expected result from the specification.

# UNIT TESTING :: UNITTEST FRAMEWORK

## Unittest

- **framework** for writing small tests
- **scales** for complex functional testing
- **applications** and **libraries**

- <https://docs.python.org/3/library/unittest.html>



The screenshot shows the Python 3.10.2 documentation page for the `unittest` module. The page title is "unittest — Unit testing framework". The left sidebar contains a "Table of Contents" with links to various sections: `unittest` — Unit testing framework, Basic example, Command-Line Interface, Test Discovery, Organizing test code, Re-using old test code, Skipping tests and expected failures, Distinguishing test iterations using subtests, Classes and functions, and Class and Module Fixtures. The main content area includes the source code link `Lib/unittest/_init_.py`, a note about skipping to the list of assert methods, and a description of the `unittest` framework. It states that the framework was inspired by JUnit and supports test automation, sharing of setup and shutdown code, aggregation of tests, and independence of tests from the reporting framework. It also mentions that `unittest` supports important concepts in an object-oriented way, such as test fixture, test case, and test suite.

Python » English » 3.10.2 » 3.10.2 Documentation » The Python Standard Library » Development Tools » **unittest** — Unit testing

## unittest — Unit testing framework

Source code: [Lib/unittest/\\_init\\_.py](#)

(If you are already familiar with the basic concepts of testing, you might want to skip to [the list of assert methods](#).)

The `unittest` unit testing framework was originally inspired by JUnit and has a similar flavor as major unit testing frameworks in other languages. It supports test automation, sharing of setup and shutdown code for tests, aggregation of tests into collections, and independence of the tests from the reporting framework.

To achieve this, `unittest` supports some important concepts in an object-oriented way:

**test fixture**

A *test fixture* represents the preparation needed to perform one or more tests, and any associated cleanup actions. This may involve, for example, creating temporary or proxy databases, directories, or starting a server process.

**test case**

A *test case* is the individual unit of testing. It checks for a specific response to a particular set of inputs. `unittest` provides a base class, `TestCase`, which may be used to create new test cases.

**test suite**

# UNIT TESTING :: PRINCIPLE

```
CalculatorTest.py x
1 import unittest
2 from Calculator import Calculator
3
4
5 class CalculatorTest(unittest.TestCase):
6     def setUp(self):
7         print("\n----- setup ----- \n")
8         self.calculator = Calculator()
9
10    def tearDown(self):
11        print("\n----- teardown ----- \n")
12        del self.calculator
13
14    def test_sum_positive(self):
15        self.assertTrue(self.calculator.last_value == 0) # for
16        self.assertEqual(7, self.calculator.sum_positive(3, 4))
17        self.assertEqual(7, self.calculator.last_value)
18        self.assertEqual(6, self.calculator.sum_positive(2, 4))
19        self.assertEqual(6, self.calculator.last_value)
```

- **Import** necessary packages and the tested class
- **setUp()** runs before each test method
- **tearDown()** runs after each test method
- **test\_ + name** of method define your test cases for this method

# UNIT TESTING :: PRINCIPLE

Unittest uses **assert methods**

- to **verify assumptions**
- help to **find bugs** quickly
- systematic way to **check** if the internal state of a program is **as expected**

```
14 ▶ def test_sum_positive(self):  
15     self.assertTrue(self.calculator.last_value == 0) # for  
16     self.assertEqual(7, self.calculator.sum_positive(3, 4))  
17     self.assertEqual(7, self.calculator.last_value)  
18     self.assertEqual(6, self.calculator.sum_positive(2, 4))  
19     self.assertEqual(6, self.calculator.last_value)
```

**assert** (expression)

**assertEqual** (expected, actual, ["message"])

- compares expected vs. actual value
- print message, if not equal

**assertTrue** (condition)

**assertFalse** (condition)

- check boolean condition

# UNIT TESTING :: PRINCIPLE

```
CalculatorTest.py x
1 import unittest
2 from Calculator import Calculator
3
4
5 class CalculatorTest(unittest.TestCase):
6     def setUp(self):
7         print("\n----- setup -----")
8         self.calculator = Calculator()
9
10    def tearDown(self):
11        print("\n----- teardown -----")
12        del self.calculator
13
14    def test_sum_positive(self):
15        self.assertTrue(self.calculator.last_val)
16        self.assertEqual(7, self.calculator.sum_posi)
17        self.assertEqual(7, self.calculator.last_val)
18        self.assertEqual(6, self.calculator.sum_posi)
19        self.assertEqual(6, self.calculator.last_val)
```

Run...

The test framework executes all or selected test cases and creates a summary about the results.

Run: pytest for CalculatorTest.CalculatorTest x

Test Results 14 ms

- CalculatorTest 14 ms
  - CalculatorTest 14 ms
    - test\_div 1 ms
    - test\_sum\_negative 12 ms
    - test\_sum\_positive 1 ms

Tests failed: 1, passed: 2 of 3 tests – 14 ms

```
self = <CalculatorTest.CalculatorTest testMethod=test
```

```
def test_sum_negative(self):
    self.assertEqual(-1, self.calculator.sum_posi)
    self.assertEqual(-1, self.calculator.last_val)
    self.assertEqual(-2, self.calculator.sum_posi)
```

[CalculatorTest.py:24](#): AssertionError

## Summary on executed tests:

- Green OK
- Red/Orange Error/Failure (wrong result)

# EXCEPTIONS :: OVERVIEW

Exceptions cannot be ignored --> In worst case there is an uncontrolled program crash.

Basically, exceptions should be handled either

- to repair the situation
- or at least to avoid an uncontrolled program crash.

## How to deal with it?

1. When an exception has been thrown, an **exception object** is created (including information such as name, description, call stack).
2. An exception is passed back to calling functions until one finally "*catches*" the exception.
3. In **Python** an exception is caught using a **try-except** block.



# EXCEPTIONS :: TYPES

Class	Description
Exception	A base class for most error types
AttributeError	Raised by syntax <code>obj.foo</code> , if <code>obj</code> has no member named <code>foo</code>
EOFError	Raised if “end of file” reached for console or file input
IOError	Raised upon failure of I/O operation (e.g., opening file)
IndexError	Raised if index to sequence is out of bounds
KeyError	Raised if nonexistent key requested for set or dictionary
KeyboardInterrupt	Raised if user types ctrl-C while program is executing
NameError	Raised if nonexistent identifier used
StopIteration	Raised by <code>next(iterator)</code> if no element; see Section 1.8
TypeError	Raised when wrong type of parameter is sent to a function
ValueError	Raised when parameter has invalid value (e.g., <code>sqrt(-5)</code> )
ZeroDivisionError	Raised when any division operator used with 0 as divisor

# EXCEPTIONS :: EXCEPTION HANDLING

## try - except

- place error prone code within a **try block**
- define and catch exceptions in **except blocks**
- consider the exception type **hierarchy**

## `print(e), print(e.__class__)`

- gather **information** about the exception
- print a specific **message**

## finally:

- **always** executed
- **clean-up** resources
- **close** files and database connections

```
ExceptionDemo.py x
1  class ExceptionDemo:
2      try:
3          x = 2
4          y = 0
5          print(x / y)
6
7      except ZeroDivisionError as e:
8          print("Cannot divide by zero")
9          print(e)
10         print("Class: ", e.__class__)
11         print("Additional info", e.__traceback__)
12
13     except (IndexError, ZeroDivisionError):
14         # handle multiple exceptions
15         # a ZeroDivisionError will never execute
16         pass
17
18     finally:
19         # cleanup resources
20         pass
```

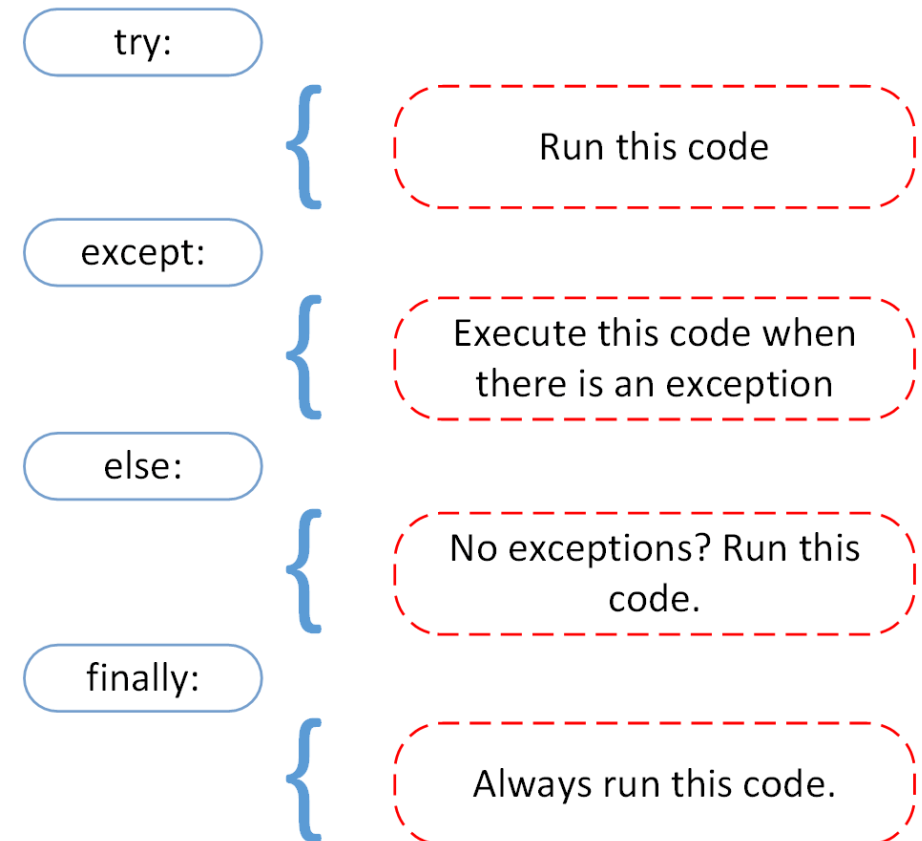
# EXCEPTIONS :: EXCEPTION HANDLING

```
try:
    result = int(input("Please enter a number: "))
except ValueError:
    print("Invalid number. Try again...")
else:
    print("result is", result)
finally:
    print("executing finally clause")
```

**Abort** if a runtime error is not handled by any of the above

**Raise** an exception

- do not abort execution
- force a specified exception to occur
- e.g., **raise AssertionError**
- show which test failed (output vs. expected result)



# PREEXERCISE DEBUGGING / CODE TESTING



Algorithms and Data Structures 1  
Exercise – 2023S

Markus Jäger (Computer Science)  
Florian Beck (Artificial Intelligence)  
Raja Zafar (Artificial Intelligence)

Institute of Pervasive Computing  
Johannes Kepler University Linz

markus.jaeger@jku.at  
florian.beck@jku.at  
raja.zafar@pervasive.jku.at

**JOHANNES KEPLER  
UNIVERSITÄT LINZ**  
Altenberger Straße 69  
4040 Linz, Österreich  
jku.at