Prof. Dr. Gemma Roig
M.Sc. Alperen Kantarcı
M.Sc. Gamze Akyol

Programmieren für Studierende der Naturwissenschaften

Lecture 5 – Testing error messages, self-help and “OOP“ (extra)

Contents

- L1: Basics of programming
  P1: Exercise 1 and help to installments.

- L2: Elementary data types and control structures
  P2: Exercises

- L3: Aggregated data types
  P3: Exercises

- L4: Aggregated data types and functions
  P4: Exercises

- L5:Testing,error messages, self-help and OOP (extra)
  P5: Exercises

Static and Dynamic Programming

- **Premise:** The later an error is found, the more difficult the correction

- Finding sources of errors in advance saves time during testing:
- Static program analysis

- **Code Review:** Structure, Semantics, Syntax and Logic
    - Rules and specifications

- Dynamic program analysis of an executable implementation

    - Verification - proof of correctness against the specification
        - Are we developing correctly?
    - Validation - meeting expectations
        - Are we developing the right thing?

Time Required

Up to 50% of the development time is testing! Not in frequently even more costly!

- Developers can test in an economically sensible way
  - Psychologically questionable

- Formal test processes can vary in complexity:
  - Sequence planning
  - Sections that do not produce data
  - Sections that produce data
  - Sections that require data
- All these steps cost time!

Planning

- Test preparation
  - Test quantities and target results
  - If applicable, test environment(s) and test object(s)

- (Finally!)Test execution
- Spelling and syntax errors
- Program logic is tested and logged manually
- Test and log test quantities

- Test evaluation
- Localization and elimination of the causes of errors
- Return place holders and test objects to real code
- Test everything again

Selection of test cases

- Work more efficiently through appropriate minimization of test cases
- Do not test twice
- Test border line cases

And/Or

- Generate random test cases

"Program testing can be used to show the presence of bugs, but never show their absence!"
[Edsger Wybe Dijkstra (1930-2002): The Humble Programmer, ACM Turing Lecture 1972]

Selection of test cases

Practice

- Error messages
    - Type
    - Meaning
    - Place in code
- Is the Procedure wrong without an error message?
    - Identify cause of error in source code
    - Define your own error messages (we don’t do that)

- User input as a frequent source of errors (example: checksum) Test!

All Python "built-in" error messages (Built-in Exceptions) can be found in the documentation: https://docs.python.org/3/library/exceptions.html.

Name Error

- At the time of execution the name used is not known

- Common causes:
  - Variable name misspelled
  - Variable has not been defined yet
  - A module,which should be used, was not imported
  - A function is called before it has been defined

```
>>> a = 10
>>> name = 'hello'
>>> print(NAME)
Traceback (most recent call last):
  File "<pyshell#2>", line 1, in <module>
    print(NAME)
NameError: name 'NAME' is not defined
```

Syntax and EOL Errors

ParseError indicates syntax error

- Common causes:
  - Missing brackets
  - Missing quotation marks
  - Missing commas
  - Missing colon

EOL:end-of-line

- Common causes:
  - Missing brackets
  - Missing quotation marks
  - Missing commas
  - Missing colon

```
Type "copyright", "credits" or "license()" for more information.
>>> print("hello world")
hello world
>>> print("hello world)

SyntaxError: EOL while scanning string literal
>>> print("hello world')

SyntaxError: EOL while scanning string literal
>>> print('hello world')

hello world
>>>
print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)
```

Type Error

Type Error

- Indicates wrong data types in an operation
- Common causes:
- Incorrect data types for simple operation
- An incorrect data type was inserted into a function and leads to incorrect calculations there
- A return was forgotten and therefore the function result is evaluated to "None" and further used

More errors

Indentation Error:

- Incorrect indentation

- **Index Error:**
- Use of invalid indices, e.g.
- `a = [1,2] print (a[2])`

- **Key Error**
- An attempt is made to access a key in the dictionary that does not exist

- **IO Error**:
- Operations on files that do not exist

Errors without error messages

- Quite frequently happens
  - Output values and variable types with `print` for control purposes
  - Output branches with `print`

- If necessary, comment out and examine suspicious areas

- Error in loops
  - Index instead of entry or vice versa
  - Note: range(n): Start at 0 and end at n-1
  - Check termination conditions

Errors without error messages

- Quite frequently happens
  - Output values and variable types with `print` for control purposes
  - Output branches with `print`

- If necessary, comment out and examine suspicious areas

- Error in loops
  - Index instead of entry or vice versa
  - Note: range(n): Start at 0 and end at n-1
  - Check termination conditions

How to avoid mistakes?

- Use comments

- Insert Doc strings

- Give sensible variable/function names
- Keep nesting depth low (separate the logic into smaller functions)

Doc Strings

GOETHE UNIVERSITÄT FRANKFURT AM MAIN

- Adhere to conventions
  - First sentence short
  - After that blank line
  - Further explanations
  - Blank line
  - Language:English

```python
def fact (n):
  """Computes the factorial of n."""
  if (n <= 1):
    return 1
  else:
    return n*fact(n-1)
```

```python
def fact (n):
  """Ein kurzer Satz, der die Funktionsweise erklärt.

  Hier könnten zusätzliche Infos stehen

  """
  if (n <= 1):
    return 1
  else:
    return n*fact(n-1)
```

Doc Strings (If you don‘t want to write documentation)

The docstring of a module m is stored in m.doc and can thus also be read out.

```
>>> import testfile
>>> testfile.fact.__doc__
'Ein kurzer Satz, der die Funktionsweise erklärt.\n\n  Hier könnten zusätzliche
Infos stehen\n\n  '
>>> help(testfile.fact)
Help on function fact in module testfile:

fact(n)
    Ein kurzer Satz, der die Funktionsweise erklärt.

    Hier könnten zusätzliche Infos stehen
```

- On-the-fly maintenance of docstrings can make documentation work much easier and ensures that your code is always up to date
- Especially recommended for collaborative work

Unavoidable mistakes

- In some situations, mistakes cannot be avoided
  - Or even belong to the conception
  - Solution: Exceptions
    - A powerful tool
    - Use with caution
    - Ignore error messages
      - ...or have them react to it specifically:

```
try:
    <instructions>
except <TypeOfError>: #or all errors by except
    <AlternativeInstructions>
```

- Simplification:try this and if an error occurs, execute that

Try except

try:
    <instructions>
except <TypeOfError>:
    <AlternativeInstructions>

- Execute try block
  - No error: except block is skipped
  - Error: Execute except block
- A try can be followed by several excepts, which handle different errors accordingly

Try except

```
while True:
    try:
        n = input("Bitte eine Ganzzahl (integer) eingeben: ")
        n = int(n)
        break
    except ValueError:
        print("Keine Integer! Bitte nochmals versuchen ...")
print('Super! Das wars!')
```

Conclusion

Conclusion

- Testing is time-consuming and is part of software development
- Error messages can be bypassed

- However, ignoring them can have fatal consequences
- Comments can massively facilitate testing
- Not seeing error messages does not mean an error-free implementation
- Testing is not a panacea!
- However, as a developer you can’t do without it!

Object Oriented Programming

- It is the most commonly used paradigm in big software projects
- Think everything like an object and their relation to the world
- Makes the abstraction easier -> maintenance is easier too
- You evolve your classes by inheriting other classes (like evolution of species)

Object Oriented Programming

Object-oriented programming (OOP) is a programming paradigm that uses "objects" to design software. Objects are instances of classes, which can contain data (attributes) and functions (methods) related to that object.

OOP has several principles, including:

1. **Encapsulation**: Binding the data (attributes) and functions (methods) into a single unit called a class.
2. **Abstraction**: Hiding complex implementation details and showing only the essentials.
3. **Inheritance**: Allows one class to inherit properties and methods from another class.
4. **Polymorphism**: Allows one interface to be used for a general class of actions.

Class and Object

```
class Dog:
    def __init__(self, name, breed):
        self.name = name
        self.breed = breed

    def bark(self):
        print(f"{self.name} barks!")

# Create an instance of the Dog class
max = Dog("Max", "Golden Retriever")
max.bark()  # Output: Max barks!
```

Encapsulation

```
class Dog:
    def __init__(self, name, breed):
        self.name = name
        self.breed = breed

    def bark(self):
        print(f"{self.name} barks!")

# Create an instance of the Dog class
max = Dog("Max", "Golden Retriever")
max.bark()  # Output: Max barks!
```

In the above example, name and breed are attributes and bark is a method. They're encapsulated within the Dog class.

Abstraction

GOETHE UNIVERSITÄT FRANKFURT AM MAIN

```
class Calculator:
    def add(self, x, y):
        return x + y

    def subtract(self, x, y):
        return x - y

calc = Calculator()
print(calc.add(5, 3))  # Output: 8
```

We don't need to know how the add or subtract functions work internally. We just use them.

Inheritance and Polymorphism (Only needed if you want to go advanced levels)

```
class Poodle(Dog):
    def show_off(self):
        print(f"{self.name}, the Poodle, is showing off!")

lucy = Poodle("Lucy", "Poodle")
lucy.bark()  # Output: Lucy barks!
lucy.show_off()  # Output: Lucy, the Poodle, is showing off!
```

Inheritance and Polymorphism (Only needed if you want to go advanced levels)

```
class Cat:
    def speak(self):
        print("Meow!")

class Dog:
    def speak(self):
        print("Woof!")

def animal_speak(animal):
    animal.speak()

tom = Cat()
max = Dog()

animal_speak(tom)  # Output: Meow!
animal_speak(max)  # Output: Woof!
```

One example to test

- We save this file as circle.py

```
class Circle:
def __init__(self, radius):
    self.radius = radius

def calculate_area(self):
    return round(3.141* self.radius ** 2, 2)
```

One example to test

- Now we can use our classes in other Python files

- >>> from circle import Circle
- >>> circle_1 = Circle(42)
- >>> circle_2 = Circle(7)
- >>> circle_1
- <__main__.Circle object at 0x102b835d0>
- >>> circle_2
- <__main__.Circle object at 0x1035e3910>

One example to test

- We can access to object variables and functions

- >>> from circle import Circle
- >>> circle_1 = Circle(42)
- >>> circle_2 = Circle(7)
- >>> circle_1.radius 42
- >>> circle_1.calculate_area() 5541.77
- >>> circle_2.radius 7
- >>> circle_2.calculate_area() 153.94
- >>> circle_1.radius = 100
- >>> circle_1.radius 100
- >>> circle_1.calculate_area() 31415.93

Conclusion

- Object Oriented Programming is good for keeping a structured code.
- More useful in big projects (makes it easier to maintenance and expansion)
- If you use modules, or create small projects. You need to learn it.
- Makes it easier to understand external modules for your own area (BioPy, ChemPy, PyTorch, NumPy, SciPy..)
- Also makes it easier to read documentation or even look at the source codes
- There is no ”correct way”. It is your own imagination and decision. There are guidelines but everyone codes differently.