Programming in Python

Testing, debugging, exceptions and errors
lecture 4

Department of Cybernetics and Artificial Intelligence
Technical University of Košice
Ing. Ján Magyar, PhD.

Exceptions and errors

- indicate faulty code
- two main types
  - syntax (errors)
  - runtime (exceptions)

Syntax errors

- interpreter identifies them before running the code
- easy to solve
- the error might not be in the place indicated by the interpreter

```
  File "C:\Users\Ian\Desktop\test.py", line 2
    if condition print("okay")
                  ^
SyntaxError: invalid syntax
```

Runtime errors/exceptions

- occur during runtime
- usually caused only by selected input

```
number = 10
divide_by = 0
print(number / divide_by)
```

```
  File "C:\Users\Ian\Desktop\test.py", line 3, in
<module>
    print(number / divide_by)
ZeroDivisionError: division by zero
```

Handling errors and exceptions

- keywords `try / except`

```
number = 10
divide_by = 0
try:
    print(number / divide_by)
except ZeroDivisionError:
    print("You're trying to divide by zero")
```

Handling multiple exceptions

```
number = 10
divide_by = 0
try:
    print(number / divide_by)
except (ZeroDivisionError, NameError, ValueError):
    print("You're trying to divide by zero")
```

Handling multiple exceptions

```
number = 10
divide_by = 0
try:
    print(number / divide_by)
except ZeroDivisionError:
    print("You're trying to divide by zero")
except NameError:
    print("Undefined variable")
except ValueError:
    print("Invalid value")
```

Working with exceptions

```
my_dict = dict()
if 'a' not in my_dict:
    my_dict['a'] = 1
else:
    my_dict['a'] += 1

print(my_dict)
```

```
my_dict = dict()
try:
    my_dict['a'] += 1
except KeyError:
    my_dict['a'] = 1

print(my_dict)
```

Executing code only when no errors occurred

```
number = 10
divide_by = 0
try:
    print(number / divide_by)
except ZeroDivisionError:
    print("You're trying to divide by zero")
else:
    print("Everything went well")
```

Finally block

- the code executes regardless of the existence of any error

```
try:
    print(5 / 0)
finally:
    print("I still get printed")


I still get printed
Traceback (most recent call last):
  File "C:\Users\Ian\Desktop\test.py", line 2, in <module>
    raise ValueError
ValueError
```

Creating exceptions

- keyword `raise`
- defines any error message
- finishes execution if error is not handled elsewhere

```
number = 10
divide_by = 0
if divide_by == 0:
    raise ZeroDivisionError("You're trying to divide by zero")
print(number / divide_by)
```

Defining own exceptions

- errors and exceptions are a subclass of `Exception`

```
class MyError(Exception):
    def __init__(self, expression, message):
        self.expression = expression
        self.message = message
```

Most common exceptions

- `AttributeError`
- `IndexError`
- `KeyError`
- `NameError`
- `TypeError`
- `ValueError`
- `RuntimeError`

AttributeError

- connected to object oriented programming
- we want to access a non-existent attribute (method or field)

```
my_lst = [1, 2, 3]
my_lst.add(4)

Traceback (most recent call last):
  File "C:\error_examples.py", line 10, in <module>
    my_lst.add(4)
AttributeError: 'list' object has no attribute 'add'
```

IndexError

- we want to access an element under a non-existent index
- usually with lists, tuples, and arrays

```
my_lst = [1, 2, 3]
print(my_lst[3])

Traceback (most recent call last):
  File "C:\error_examples.py", line 15, in <module>
    print(my_lst[3])
IndexError: list index out of range
```

KeyError

- similar to `IndexError`
- using a non-existent key for a dictionary

```
my_dct = {"one": 1, "two": 2, "three": 3}
print(my_dct["four"])
```

```
Traceback (most recent call last):
  File "C:\error_examples.py", line 20, in <module>
    print(my_dct["four"])
KeyError: 'four'
```

NameError

- we want to use a non-existent component, e.g. variable or method

```
my_lst = [1, 2, 3]
print(my_list[0])

Traceback (most recent call last):
  File "C:\error_examples.py", line 25, in <module>
    print(my_list[0])
NameError: name 'my_list' is not defined. Did you mean:
'my_lst'?
```

TypeError

- we want to execute an operation with arguments of incorrect types

```
a = "abc"
print(a ** 2)

Traceback (most recent call last):
  File "C:\error_examples.py", line 30, in <module>
    print(a ** 2)
TypeError: unsupported operand type(s) for ** or pow():
'str' and 'int'
```

ValueError

- an operation or function receives an argument with an incorrect type or value

```
import math

print(math.sqrt(-4))

Traceback (most recent call last):
  File "C:\error_examples.py", line 36, in <module>
    print(math.sqrt(-4))
ValueError: math domain error
```

RuntimeError

- non-specific error during runtime
- the error does not fit any of the standard categories
- the error message elaborates on what happened

Program testing

- **process**, where the goal is to find errors and faults in the code
- for now we are not interested in the reason nor possible solutions
- verification and validation

Verification and validation

- verification
  - checks if the program or its part meets design requirements (from the programmer’s side)
  - does not deal only with what a program does, but also whether it does it in the desired way
- validation
  - checks if the program or its part meets user requirements – what it does
  - the goal is to increase the reliability of code – we never reach 100%

Test types

- unit tests
  - independent testing of code blocks
  - testing functions one by one
- integration tests
  - testing the program as a whole
  - used to uncover faults in communication between modules and code blocks
- it’s always best to start with unit tests

Designing tests

- testing the program on every possible input – impractical and often impossible
- testing on an appropriately selected set of test cases
  - the set must be small enough to run the tests in an acceptable time period
  - the set must be big enough to represent every possible input

Test set

- frequent input examples (expected input)
- extreme input examples
- invalid input examples (maybe most important)

Testing in Python

- usually in a separate file
- importing the tested functions from the primary file
- keyword **`assert`** with a condition
- throws **`AssertionError`** if test is not passed

```
def is_even(number):
    return not number % 2

assert is_even(7) is False
```

- module `unittest` for automated testing

Debugging

- we found a fault in our code, we need to remove it
- process, during which our goal is to remove **every** bug from the code
- the fault must not manifest in the code being non-functional
- sometimes during debugging we also want to optimize code execution, i.e. increase performance

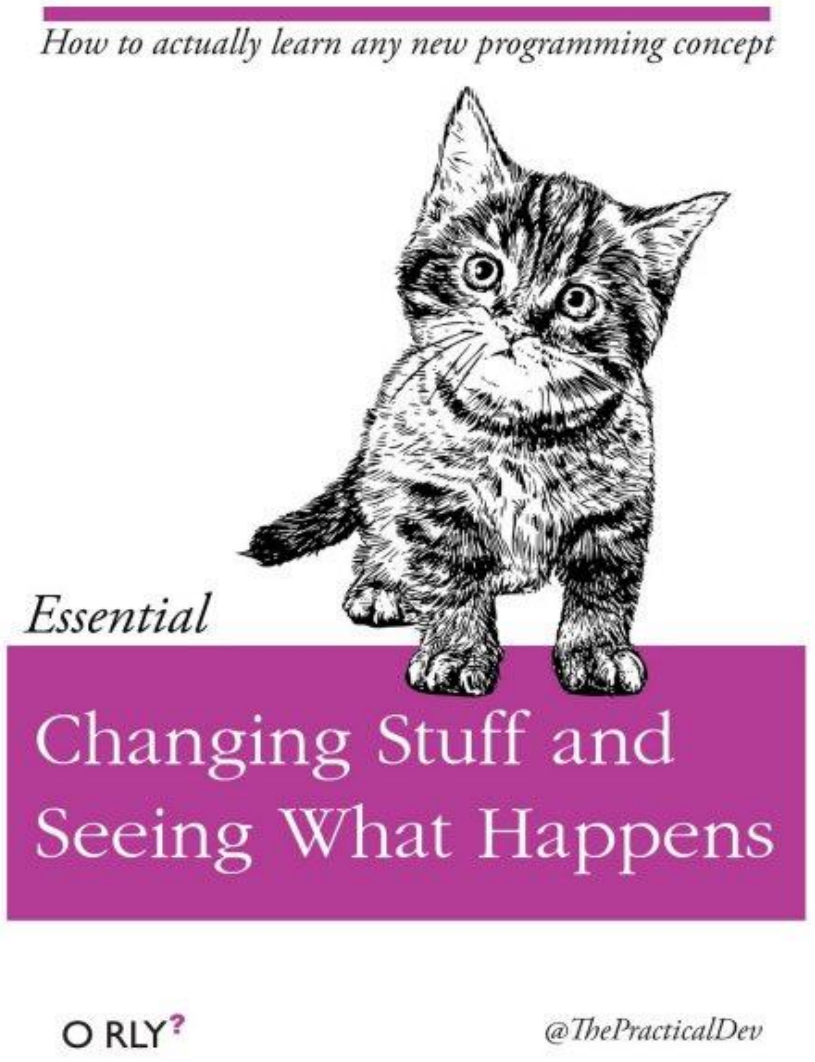
Other myths about bugs

1. bugs appear in code
   - the bug is in the code because the programmer put it there (unwillingly)
   - the bug is nothing more than the programmer’s mistake
2. bugs multiply – if you correct one bug, two take its place
   - if you have to correct more bugs, it means that you made more mistakes than you thought
   - the goal is never to correct one bug, but make non-functional code into functional

Debugging tools

- multiple tools for different IDEs
- the best way to debug
  - read the code
  - use `print`

The debugging process



The debugging process

1. Where is the fault in your code?
2. How can the code produce incorrect output?
3. Is this a unique fault or did we make the same mistake multiple times in the program?
4. How do we correct the fault?

- always test on the simplest possible input
- when identifying the fault’s position use binary search

Example

```
def is_palindrome():
    original_list = list()
    done = False
    while not done:
      elem = input("Enter element. Return when done. ")
      if elem == '':
            done = True
      else:
            original_list.append(elem)

    test_list = original_list
    test_list.reverse()
    return test_list == original_list
```

Typical mistakes

- incorrect parameter/argument order
- grammar, typos, uppercase/lowercase letters
- initialization – in loop or outside?
- side effects of called functions
- aliases vs. copies, deep vs. shallow copies
- equality of objects vs. equality of values

+ each programmer has his/her own frequent mistakes

Best practices when debugging

- check if you test the file you are updating
- systemic process
- make notes of what you've already tried
- check if your assumptions are correct
- debug the code and not comments
- get some help
- explain to someone what the code does and what it is supposed to do
- take a break
- simplify code
- save old versions of code

Conclusion

- errors and exceptions
- handling exceptions in Python
- selected main exceptions and their meaning
- testing and its goals
- unit tests in Python
- debugging process