

## Documentation

## Testing

## Purpose of documentation

- ▶ Documentation can be aimed at different user groups:
  - ▶ End user needs instructions at how to use your software
  - ▶ Seller of the software needs to know the strengths and abilities
  - ▶ Implementers and maintainers need to understand the inner workings of the program, the technical decisions made, ...
- ▶ Users and developers of the software can be separate groups of people; some way of passing information is needed
- ▶ After time passes a developer of the software may forget what he did or meant when he originally wrote the code
- ▶ A person who didn't participate in coding of the program may later be assigned to maintain the software

## Different kinds of documentation 1

- ▶ In addition to the normal design and implementation documentation and user manual, the following parts can also be considered as part of the documentation:
  - ▶ In the code, well chosen identifiers (function, class, module, .. names) can be thought as part of the documentation
  - ▶ Comments in the code. Comments in Python start with the # sign and continue until the end of the line
  - ▶ One should use comments wisely: where the code itself may not be obvious
  - ▶ Bad example of comments can be seen in the following:

```
for i in L: # iterate through all
           # the elements of the list L
```

## Different kinds of documentation 2

- ▶ Docstrings form an important part of software documentation in Python
- ▶ You should have documentation strings for all functions, classes, methods, and modules
- ▶ For example, the online help system of Python will use these pieces of documentation
- ▶ In addition to plain text, the docstrings can also contain some special notation

## Docstrings

- ▶ For example, you can use the program *epydoc* that reads your modules and also the docstrings, and uses this information to generate pdf or html pages.
- ▶ *epydoc* can understand several special markup languages inside docstrings
  - ▶ `epytext` (native for *epydoc*, quite simple)
  - ▶ `reStructuredText`
  - ▶ `Javadoc`
- ▶ The special notation of these markup languages support, for instance, specifying information about the parameters and return value of a function, and about their types
- ▶ Some markup languages also have the ability of representing more complex structures like arrays

## The doctest module 1

- ▶ Another example of using special notation inside docstrings is the `doctest` module
- ▶ One can embed pieces of code inside the docstrings, and the `doctest` module can then extract these pieces and execute them to see they are functioning properly
- ▶ These examples can be ripped directly from the use of interactive interpreter:
- ▶ Follow the `'>>> '` string by the wanted expression
- ▶ And on the next lines include the evaluated value of the expression

## The doctest module, an example

```
def double(x):
    """Returns the parameter multiplied by two
    Example 1:
    >>> double(4)
    8

    Example 2:
    >>> double("abc")
    'abcabc'"""
```

## The doctest module, an example continues

```
Example 3:
>>> double({"a":1})
Traceback (most recent call last):
...
TypeError: unsupported operand type(s) for *: 'dict' and

return x*2

if __name__ == '__main__':
    import doctest
    doctest.testmod()
```

## The doctest module 2

- ▶ The examples in docstring can show the user some typical usecases of your functions
- ▶ They can also give some negative examples: how the function isn't suppose to work
- ▶ They can be also used to test automatically that your program works
- ▶ Include the following lines in your module, to make the example cases run automatically, when you run your module:

```
if __name__ == '__main__':
    import doctest
    doctest.testmod()
```

## What to include in the document

- ▶ User manual
- ▶ Description of the interfaces of classes, functions, modules, and methods of your program
- ▶ Design choices: used algorithms and data structures. If an algorithm or data structure is better described in some publication, give reference to it, instead of repeating the description in the document
- ▶ Description of the architecture: class hierarchies, dependencies between units, external libraries used, etc
- ▶ What you have tested and how the tests succeeded
- ▶ What flaws your programs has, how could it be improved in the future

## Testing 1

- ▶ Testing is performed on software to make sure it works correctly in all cases before releasing the software to the end users
- ▶ When some testcase notices that a feature of the program doesn't work as it should, the programmer tries to find the reason for this behaviour and fix it
- ▶ After fixing the code, all tests should be run again
- ▶ During the development phase tests are run many times, and therefore running tests should be automated as much as possible.

## Testing 2

- ▶ The tests should also run fast
- ▶ The report that the test programs print should be very brief if everything went ok
- ▶ If a test program gives lots of output, it may be difficult to spot whether all the tests passed or not
- ▶ The testcases should especially check the behaviour in the corner cases
- ▶ For example, if a function gets a list as a parameter, then one should make sure it works sensibly also when we pass an empty list as the parameter
- ▶ Or, if another function gets an integer parameter that denotes a count, then check that count values zero or negative values don't do anything crazy

## Unit testing and system testing

- ▶ Testing is usually divided into two categories: *unit testing* and *system testing*
- ▶ Unit testing involves testing of small units (functions, classes, modules) alone
- ▶ Testing of the program as a whole is called system testing
- ▶ One can start system testing after the major parts of the program are functioning, but unit testing can be done immediately when a new unit is created or some functionality is added to it

## System testing

- ▶ System testing checks whether the units work well together, the integration between units.
- ▶ Inputs to program should be fixed so that tests can be reproduced
- ▶ System testing can also reveal problems in unit testing
- ▶ System testing is usually platform dependent
- ▶ Big problem is testing a program with graphical user interface
- ▶ Another problem may result from accessing some web server from your program. It may be possible that the same test results can't be reproduced because of an external server, that we can't control

## Unit testing

- ▶ We saw earlier that we can write testcases for units as we write code
- ▶ Another way is to reverse this order: testcases are designed first and the program code is written after
- ▶ This is called *test-driven development* (TDD)
- ▶ In TDD one should first specify what the interfaces look like, and what the behaviour should be
- ▶ Then testcases are written, and finally the program code is written
- ▶ A slight contradiction: if a unit A uses some other units, then unit testing of A has some similarities with system testing

## The unittest module 1

- ▶ Python has a module called `unittest` to help you create testcases for your modules
- ▶ Let's suppose you have a module called `mod` that you want to test
- ▶ You should then create a file `test_mod.py` that contains your testcases
- ▶ The testcases in this file are located in classes that inherit from the class `unittest.TestCase`
- ▶ When you call the `unittest.main()` function, the methods of the test classes get executed

## The unittest module 2

- ▶ The methods of the test classes get no parameters
- ▶ These methods should in turn call some of the following methods of the base class `unittest.TestCase`:
  - ▶ `assert_`
  - ▶ `assertAlmostEqual`
  - ▶ `assertEqual`
  - ▶ `assertNotAlmostEqual`
  - ▶ `assertRaises`
- ▶ The above methods all test whether some condition holds, and if not then a message is printed and the test fails

## The unittest module 3

- ▶ Before each test method is called, a new instance of the class is created and the method `setUp` is called
- ▶ After the test method is called, a method `tearDown` is called to finalise the testcase

## Example of unittest usage

```
" This module tests the function double of doublemod.py."
import unittest
import doublemod

class Mytest(unittest.TestCase):

    def test48(self):
        self.assertEqual(doublemod.double(4), 8)

    def testIllegal(self):
        self.assertRaises(TypeError, doublemod.double,
                          {"a":1})

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