PYTHON TESTS

ERRORS AND TESTS

Usually, programmers and program developers spend a great deal of their time with debugging and testing. It's hard to give exact percentages, because it highly depends among other factors on the individual programming style, the problems to be solved and of course on the qualification of a programmer. Of course, the programming language is another important factor.

You don't have to program to get pestered by errors, as even the ancient Romans knew. The philosopher Cicero coined more than 2000 years ago an unforgettable aphorism, which is often quoted: "errare humanum est"[1]
This aphorism is often used as an excuse for failure. Even though it's hardly possible to completely eliminate all errors in a software product, we should always work ambitiously to this end, i.e. to keep the number of errors minimal.

KINDS OF ERRORS

There are various kinds of errors. During program development there are lots of "small errors", mostly typos. Whether a colon is missing - for example behind an "if" or an "else - or the keyword "True" is wrongly written with a lower case "t". These errors are called syntactical errors.[2]
In most cases, syntactical errors can be easily found, but another type of errors is harder to be solved. A semantic error is syntactically correct code, but the program doesn't behave in the intended way. Imagine somebody wants to increment the value of a variable x by one, but instead of `"x += 1"` he or she writes `"x = 1"`.
The following longer code example may harbour another semantic error:

```
x = int(input("x? "))
y = int(input("y? "))

if x > 10:
    if y == x:
        print("Fine")
else:
    print("So what?")
```

We can see two if statements. One nested inside of the other. The code is definitely syntactically correct. But it may be, that the writer of the program only wanted to output "So what?", if the value of the variable x is both greater than 10 and x is not equal to y. In this case, the code should look like this:

```
x = int(input("x? "))
y = int(input("y? "))

if x > 10:
    if y == x:
        print("Fine")
    else:
        print("So what?")
```

Both code versions are syntactically correct, but one of them violates the intended semantics. Let's look at another example:

```
>>> for i in range(7):
...     print(i)
...
0
1
2
3
4
5
6
>>>
```

The statement ran without raising an exception, so we know that it is syntactically correct. Though it is not possible to decide, if the statement is semantically correct, as we don't know the problem. It may be that the programmer wanted to output the numbers from 1 to 7, i.e. 1,2,...7
In this case, he or she does not properly understand the range function.

So we can divide semantic errors into two categories.

- Errors caused by lack of understanding of a language construct.
- Errors due to logically incorrect code conversion.

UNIT TESTS

This paragraph is about unit tests. As the name implies they are used for testing units or components of the code, typically, classes or functions. The underlying concept is to simplify the testing of large programming systems by testing "small" units. To accomplish this the parts of a program have to be isolated into independent testable "units". One can define "unit testing" as a method whereby individual units of source code are tested to determine if they meet the requirements, i.e. return the expected

output for all possible - or defined - input data. A unit can be seen as the smallest testable part of a program, which are often functions or methods from classes. Testing one unit should be independent from the other units. As a unit is "quite" small, i.e. manageable to ensure complete correctness. Usually, this is not possible for large scale systems like large software programs or operating systems.

MODULE TESTS WITH __NAME__

Every module has a name, which is defined in the built-in attribute __name__. Let's assume that we have written a module "xyz" which we have saved as "xyz.py". If we import this module with `"import xyz"`, the string "xyz" will be assigned to __name__. If we call the file xyz.py as a standalone program, i.e. in the following way,

```
$python3 xyz.py
```

the value of __name__ will be the string '__main__'.

The following module can be used for calculating fibonacci numbers. But it is not important what the module is doing. We want to demonstrate with it, how it is possible to create a simple module test inside of a module file, - in our case the file "xyz.py", - by using an if statement and checking the value of __name__. We check if the module has been started standalone, in which case the value of __name__ will be '__main__'. We assume that you save the following code as "fibonacci.py":

```
""" Fibonacci Module """

def fib(n):
    """ Calculates the n-th Fibonacci number iteratively """
    a, b = 0, 1
    for i in range(n):
        a, b = b, a + b
    return a

def fiblist(n):
    """ creates a list of Fibonacci numbers up to the n-th
generation """
    fib = [0,1]
    for i in range(1,n):
```

```
            fib += [fib[-1]+fib[-2]]
        return fib
```

It's possible to test this module manually in the interactive Python shell:

```
>>> from fibonacci import fib, fiblist
>>> fib(0)
0
>>> fib(1)
1
>>> fib(10)
55
>>> fiblist(10)
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
>>> fiblist(-8)
[0, 1]
>>> fib(-1)
0
>>> fib(0.5)
Traceback (most recent call last):
  File "", line 1, in
  File "fibonacci.py", line 6, in fib
    for i in range(n):
TypeError: 'float' object cannot be interpreted as an integer
>>>
```

We can see that the functions make only sense, if the input consists of positive integers. The function fib returns 0 for a negative input and fiblist returns always the list [0,1], if the input is a negative integer. Both functions raise a TypeError exception, because the range function is not defined for floats. We can test our module by checking the return values for some characteristic calls to fib() and fiblist(). So we can add e.g. the following if statement to our module:

```
if fib(0) == 0 and fib(10) == 55 and fib(50) == 12586269025:
    print("Test for the fib function was successful!")
else:
    print("The fib function is returning wrong values!")
```

If our program will be called standalone, we see the following output:

```
$ python3 fibonacci.py
Test for the fib function was successful!
```

We will deliberately add an error into our code now.

We change the following line

```
a, b = 0, 1
```

into

```
    a, b = 1, 1
```

Principally, the function fib is still calculating the Fibonacci values, but fib(n) is returning the Fibonacci value for the argument "n+1" If we call our changed module, we receive this error message:

```
$ python3 fibonacci.py
"The fib function is returning wrong values!
```

This approach has a crucial disadvantage. If we import the module, we will get output, saying the test was okay. Something we don't want to see, when we import the module.

```
>>> import fibonacci
Test for the fib function was successful!
```

Apart from being disturbing it is not common practice. Modules should be silent when being imported.

Our module should only print out error messages, warnings or other information, if it is started standalone. If it is imported, it should be silent. The solution for this problem consists in using the built-in attribute __name__ in a conditional statement. If the module is started standalone, the value of this attribute is "__main__". Otherwise the value is the filename of the module without the extension.
Let's rewrite our module in the above mentioned way:

```
""" Fibonacci Module """

def fib(n):
    """ Calculates the n-th Fibonacci number iteratively """
    a, b = 0, 1
    for i in range(n):
        a, b = b, a + b
    return a

def fiblist(n):
    """ creates a list of Fibonacci numbers up to the n-th
generation """
    fib = [0,1]
    for i in range(1,n):
        fib += [fib[-1]+fib[-2]]
    return fib

if __name__ == "__main__":
    if fib(0) == 0 and fib(10) == 55 and fib(50) == 12586269025:
        print("Test for the fib function was successful!")
    else:
        print("The fib function is returning wrong values!")
```

We have squelched our module now. There will be no messages, if the module is imported.
This is the simplest and widest used method for unit tests. But it is definitely not the best one.

DOCTEST MODULE

The doctest module is often considered easier to use than the unittest, though the later is more suitable for more complex tests. doctest is a test framework that comes prepackaged with Python. The doctest module searches for pieces of text that look like interactive Python sessions inside of the documentation parts of a module, and then executes (or reexecutes) the commands of those sessions to verify that they work exactly as shown, i.e. that the same results can be achieved. In other words: The help text of the module is parsed for example python sessions. These examples are run and the results are compared against the expected value.

Usage of doctest:
To use "doctest" it has to be imported. The part of an interactive Python sessions with the examples and the output has to be copied inside of the docstring the corresponding function.

We demonstrate this way of proceeding with the following simple example. We have slimmed down the previous module, so that only the function fib is left:

```
import doctest

def fib(n):
    """ Calculates the n-th Fibonacci number iteratively """
    a, b = 0, 1
    for i in range(n):
        a, b = b, a + b
    return a
```

We now call this module in an interactive Python shell and do some calculations:

```
>>> from fibonacci import fib
>>> fib(0)
0
>>> fib(1)
1
>>> fib(10)
55
>>> fib(15)
610
>>>
```

We copy the complete session of the interactive shell into the docstring of our function. To start the module doctest we have to call the method testmod(), but only if the module is called standalone. The complete module looks like this now:

```
import doctest

def fib(n):
    """
    Calculates the n-th Fibonacci number iteratively
```

```
    >>> fib(0)
    0
    >>> fib(1)
    1
    >>> fib(10)
    55
    >>> fib(15)
    610
    >>>

    """
    a, b = 0, 1
    for i in range(n):
        a, b = b, a + b
    return a

if __name__ == "__main__":
    doctest.testmod()
```

If we start our module directly like this

```
$ python3 fibonacci_doctest.py
```

we get no output, because everything is okay.

To see how doctest works, if something is wrong, we place an error in our code:
We change again

```
a, b = 0, 1
```

into

```
a, b = 1, 1
```

Now we get the following, if we start our module:

```
$ python3 fibonacci_doctest.py
**********************************************************************
****
File "fibonacci_doctest.py", line 8, in __main__.fib
Failed example:
    fib(0)
Expected:
    0
Got:
    1
```

```
**********************************************************************
****
File "fibonacci_doctest.py", line 12, in __main__.fib
Failed example:
    fib(10)
Expected:
    55
Got:
    89
**********************************************************************
****
File "fibonacci_doctest.py", line 14, in __main__.fib
Failed example:
    fib(15)
Expected:
    610
Got:
    987
**********************************************************************
****
1 items had failures:
   3 of   4 in __main__.fib
***Test Failed*** 3 failures.
```

The output depicts all the calls, which return faulty results. We can see the call with the arguments in the line following "Failed example:". We can see the expected value for the argument in the line following "Expected:". The output shows us the newly calculated value as well. We can find this value behind "Got:"

TEST-DRIVEN DEVELOPMENT (TDD)

In the previous chapters, we tested functions, which we had already been finished. What about testing code you haven't yet written? You think that this is not possible? It is not only possible, it is the underlying idea of test-driven development. In the extreme case, you define tests before you start coding the actual source code. The program developer writes an automated test case which defines the desired "behaviour" of a function. This test case will - that's the idea behind the approach - initially fail, because the code has still to be written.

The major problem or difficulty of this approach is the task of writing suitable tests. Naturally, the perfect test would check all possible inputs and validate the output. Of course, this is generally not always feasible.

We have set the return value of the fib function to 0 in the following example:

```
import doctest

def fib(n):
    """
    Calculates the n-th Fibonacci number iteratively

    >>> fib(0)
    0
    >>> fib(1)
    1
    >>> fib(10)
    55
    >>> fib(15)
    610
    >>>

    """

    return 0

if __name__ == "__main__":
    doctest.testmod()
```

It hardly needs mentioning that the function returns except for fib(0) only wrong return values:

```
$ python3 fibonacci_TDD.py
**********************************************************************
****
File "fibonacci_TDD.py", line 10, in __main__.fib
Failed example:
    fib(1)
Expected:
    1
Got:
    0
**********************************************************************
****
File "fibonacci_TDD.py", line 12, in __main__.fib
Failed example:
    fib(10)
Expected:
    55
Got:
    0
**********************************************************************
****
File "fibonacci_TDD.py", line 14, in __main__.fib
Failed example:
```

```
    fib(15)
Expected:
    610
Got:
    0
**********************************************************************
****
1 items had failures:
   3 of   4 in __main__.fib
***Test Failed*** 3 failures.
```

Now we have to keep on writing and changing the code for the function fib until it passes the test.

This test approach is a method of software development, which is called test-driven development.

UNITTEST

The Python module unittest is a unit testing framework, which is based on Erich Gamma's JUnit and Kent Beck's Smalltalk testing framework. The module contains the core framework classes that form the basis of the test cases and suites (TestCase, TestSuite and so on), and also a text-based utility class for running the tests and reporting the results (TextTestRunner).
The most obvious difference to the module "doctest" consists in the fact that the test cases of the module "unittest" are not defined inside of the module, which has to be tested. The major advantage is clear: program documentation and test descriptions are separate from each other. The price you have to pay on the other hand consists in an increase of work to create the test cases.

We will use our module fibonacci once more to create a test case with unittest. To this purpose we create a file fibonacci_unittest.py. In this file we have to import unittest and the module which has to be tested, i.e. fibonacci.

Furthermore, we have to create a class with an arbitrary name - we will call it "FibonacciTest" - which inherits from unittest.TestCase. The test cases are defined in this class by using methods. The name of these methods is arbitrary, but has to start with test. In our method "testCalculation" we use the method assertEqual from the class TestCase. assertEqual(first, second, msg = None) checks, if expression "first" is equal to the expression "second". If the two expressions are not equal, msg will be output, if msg is not None.

```
import unittest
from fibonacci import fib

class FibonacciTest(unittest.TestCase):

    def testCalculation(self):
        self.assertEqual(fib(0), 0)
        self.assertEqual(fib(1), 1)
        self.assertEqual(fib(5), 5)
```

```
        self.assertEqual(fib(10), 55)
        self.assertEqual(fib(20), 6765)

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

If we call this test case, we get the following output:

```
$ python3 fibonacci_unittest.py
.
----------------------------------------------------------------------
Ran 1 test in 0.000s

OK
```

This is usually the desired result, but we are now interested what happens in the error case. Therefore we will create our previous error again. We change again the well known line:

```
a, b = 0, 1
```

will be changed in

```
a, b = 1, 1
```

Now the test result looks like this:

```
$ python3 fibonacci_unittest.py
F
======================================================================
FAIL: testCalculation (__main__.FibonacciTest)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "fibonacci_unittest.py", line 7, in testCalculation
    self.assertEqual(fib(0), 0)
AssertionError: 1 != 0

----------------------------------------------------------------------
Ran 1 test in 0.000s

FAILED (failures=1)
```

The first statement in testCalculation has created an exception. The other assertEqual calls had not been executed. We correct our error and create a new one. Now all the values will be correct, except if the

input argument is 20:

```
def fib(n):
    """ Iterative Fibonacci Function """
    a, b = 0, 1
    for i in range(n):
        a, b = b, a + b
    if n == 20:
        a = 42
    return a
```

The output of a test run looks now like this:

```
$ python3 fibonacci_unittest.py
blabal
F
======================================================================
====
FAIL: testCalculation (__main__.FibonacciTest)
----------------------------------------------------------------------
----
Traceback (most recent call last):
  File "fibonacci_unittest.py", line 12, in testCalculation
    self.assertEqual(fib(20), 6765)
AssertionError: 42 != 6765

----------------------------------------------------------------------
----
Ran 1 test in 0.000s

FAILED (failures=1)
```

All the statements of testCalculation have been executed, but we haven't seen any output, because everython was okay:

```
        self.assertEqual(fib(0), 0)
        self.assertEqual(fib(1), 1)
        self.assertEqual(fib(5), 5)
```

METHODS OF THE CLASS TESTCASE

We now have a closer look at the class TestCase.

Method	Meaning

Method	Meaning
`setUp()`	Hook method for setting up the test fixture before exercising it. This method is called before calling the implemented test methods.
`tearDown()`	Hook method for deconstructing the class fixture after running all tests in the class.
`assertEqual(self, first, second, msg=None)`	The test fails if the two objects are not equal as determined by the '==' operator.
`assertAlmostEqual(self, first, second, places=None, msg=None, delta=None)`	The test fails if the two objects are unequal as determined by their difference rounded to the given number of decimal places (default 7) and comparing to zero, or by comparing that the between the two objects is more than the given delta. Note that decimal places (from zero) are usually not the same as significant digits (measured from the most significant digit). If the two objects compare equal then they will automatically compare almost equal.
`assertCountEqual(self, first, second, msg=None)`	An unordered sequence comparison asserting that the same elements, regardless of order. If the same element occurs more than once, it verifies that the elements occur the same number of times. `self.assertEqual(Counter(list(first)), Counter(list(second)))` Example: `[0, 1, 1]` and `[1, 0, 1]` compare equal, because the number of ones and zeroes are the same. `[0, 0, 1]` and `[0, 1]` compare unequal, because zero appears twice in the first list and only once in the second list.
`assertDictEqual(self, d1, d2, msg=None)`	Both arguments are taken as dictionaries and they are checked if they are equal.
`assertTrue(self, expr, msg=None)`	Checks if the expression "expr" is True.
`assertGreater(self, a, b, msg=None)`	Checks, if a > b is True.
`assertGreaterEqual(self, a, b, msg=None)`	Checks if a ≥ b
`assertFalse(self, expr, msg=None)`	Checks if expression "expr" is False.
`assertLess(self, a, b, msg=None)`	Checks if a < b
`assertLessEqual(self, a, b, msg=None)`	Checks if a ≤ b

Method	Meaning
`assertIn(self, member, container, msg=None)`	Checks if a in b
`assertIs(self, expr1, expr2, msg=None)`	Checks if "a is b"
`assertIsInstance(self, obj, cls, msg=None)`	Checks if isinstance(obj, cls).
`assertIsNone(self, obj, msg=None)`	Checks if "obj is None"
`assertIsNot(self, expr1, expr2, msg=None)`	Checks if "a is not b"
`assertIsNotNone(self, obj, msg=None)`	Checks if obj is not equal to None
`assertListEqual(self, list1, list2, msg=None)`	Lists are checked for equality.
`assertMultiLineEqual(self, first, second, msg=None)`	Assert that two multi-line strings are equal.q
`assertNotRegexpMatches(self, text, unexpected_regexp, msg=None)`	Fails, if the text Text "text" of the regular expression unexpected_regexp matches.
`assertTupleEqual(self, tuple1, tuple2, msg=None)`	Analogous to assertListEqual

We expand our previous example by a setUp and a tearDown method:

```
import unittest
from fibonacci import fib

class FibonacciTest(unittest.TestCase):

    def setUp(self):
        self.fib_elems = ( (0,0), (1,1), (2,1), (3,2), (4,3),
(5,5) )
        print ("setUp executed!")

    def testCalculation(self):
        for (i,val) in self.fib_elems:
            self.assertEqual(fib(i), val)

    def tearDown(self):
        self.fib_elems = None
        print ("tearDown executed!")
```

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

A call returns the following results:

```
$ python3 fibonacci_unittest2.py
setUp executed!
tearDown executed!
.
----------------------------------------------------------------------
Ran 1 test in 0.000s

OK
```

Most of the TestCase methods have an optional parameter "msg". It's possible to return an additional description of an error with "msg".

EXERCISES

1. Exercise:

Can you find a problem in the following code?

```
import doctest

def fib(n):
    """ Calculates the n-th Fibonacci number iteratively

    >>> fib(0)
    0
    >>> fib(1)
    1
    >>> fib(10)
    55
    >>> fib(40)
    102334155
    >>>

    """
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-1) + fib(n-2)
```

```
if __name__ == "__main__":
    doctest.testmod()
```

Answer:

The doctest is okay. The problem is the implementation of the fibonacci function. This recursive approach is "highly" inefficient. You need a lot of patience to wait for the termination of the test. The number of hours, days or weeks depend on your computer. ☺

Footnotes:

[1]The aphorism in full length: "Errare (Errasse) humanum est, sed in errare (errore) perseverare diabolicum." (To err is human, but to persist in it is diabolic")

[2]In computer science, the syntax of a computer language is the set of rules that defines the combinations of symbols that are considered to be a correctly structured document or fragment in that language. Writing "if" as "iff" is an example for syntax error, both in programming and in the English language.