UNIVERSITÄT
BERN

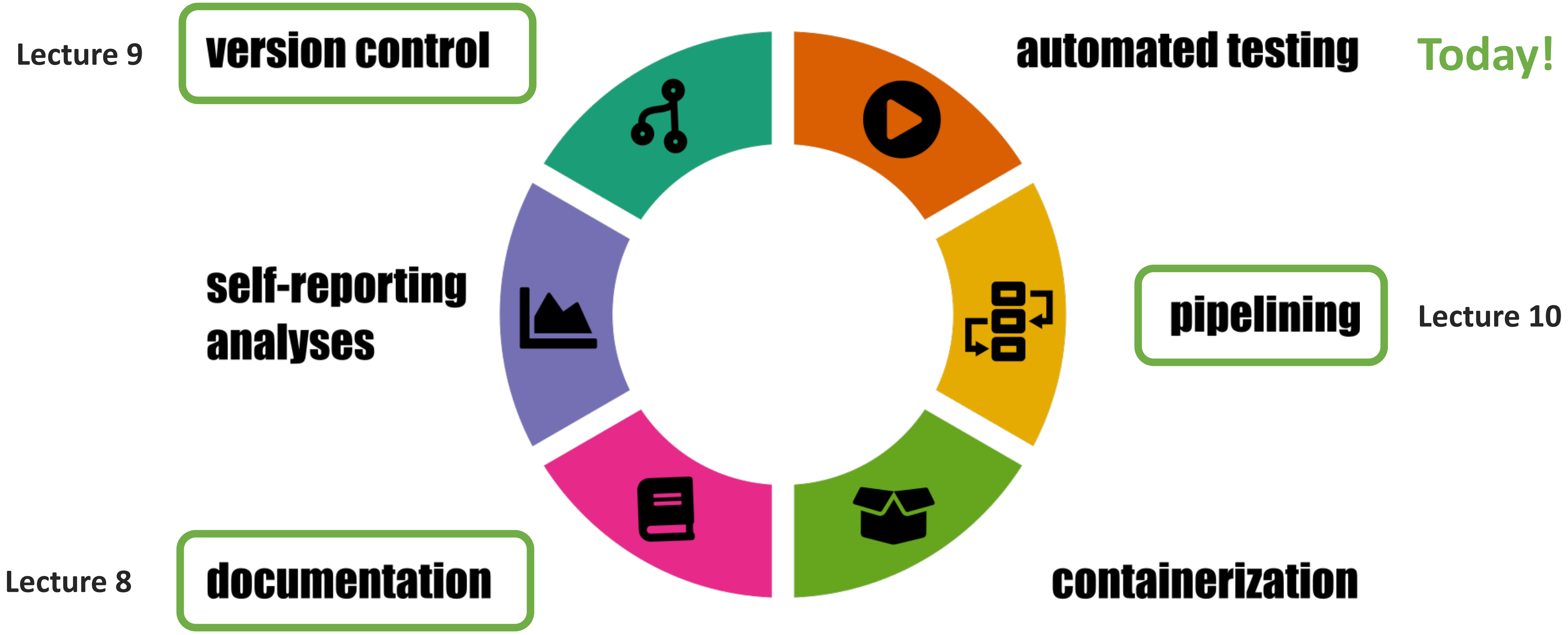
# 11. Writing robust python code

**Athina Tzovara**

University of Bern

Athina.Tzovara@unibe.ch

# Hallmarks of good scientific software




Source: https://grst.github.io/bioinformatics/2020/07/16/hallmarks-scientific-software.html

# Today

1. How do we know if our code works?

2. Exceptions and Assertions

3. Unit testing

# A little bit of history: Therac – 25

June 1985 – January 1987: six accidents with massive overdoses

Radiation machine, built in 1982

Dual mode, for milder and stronger dose

Designers had in mind a danger of overdose, and implemented some, but not sufficient tests…

If a technician missed a keystroke, the device could run in both modes, resulting in a concentrated beam of radiation…

# Why do we need to test our code?

- When things can go wrong, they go wrong

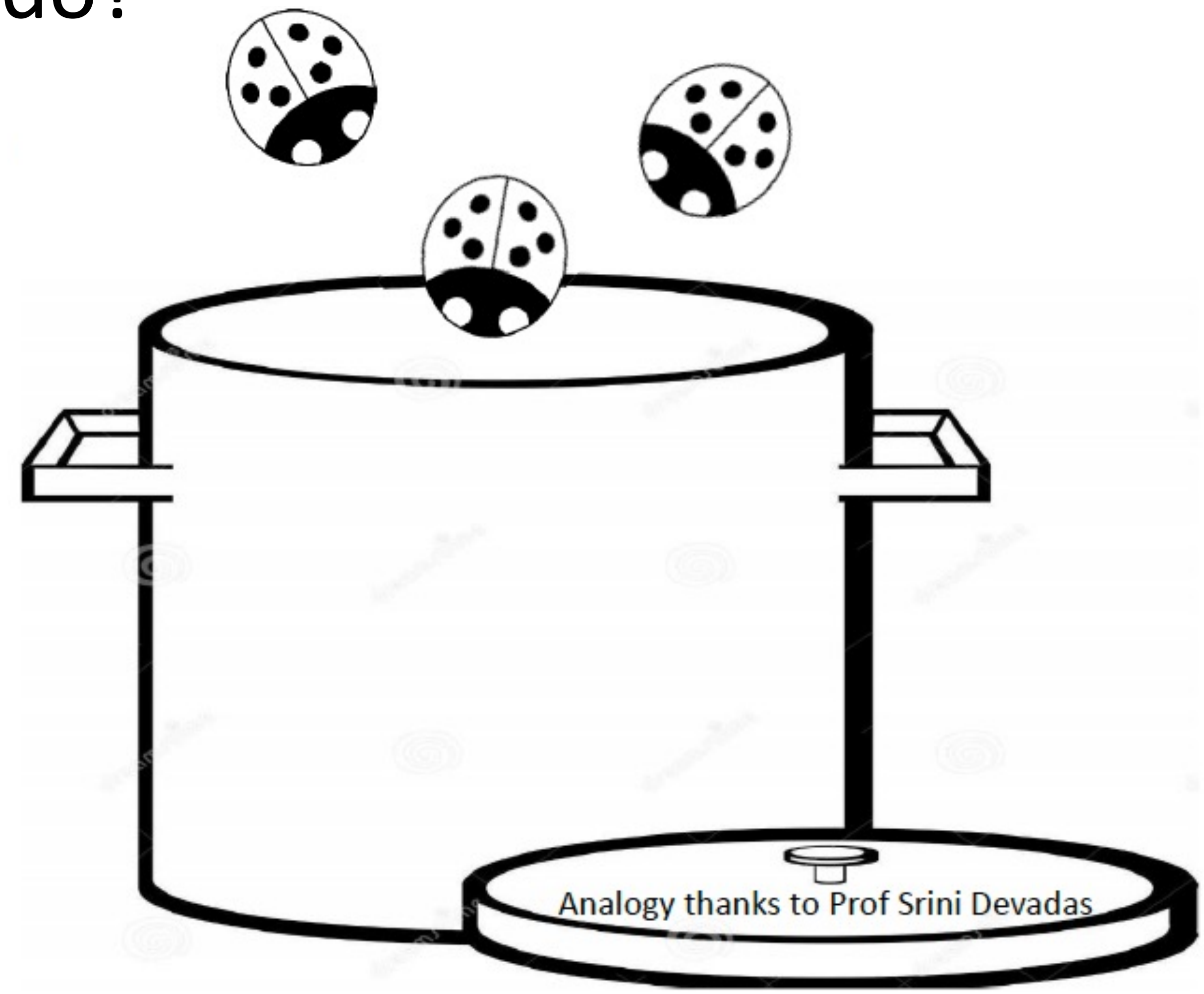
- Sometimes software bugs can be fatal

- Bugs and undesired output can be hard to identify!

- Testing a seamless part of coding

# High quality code – an analogy with soup

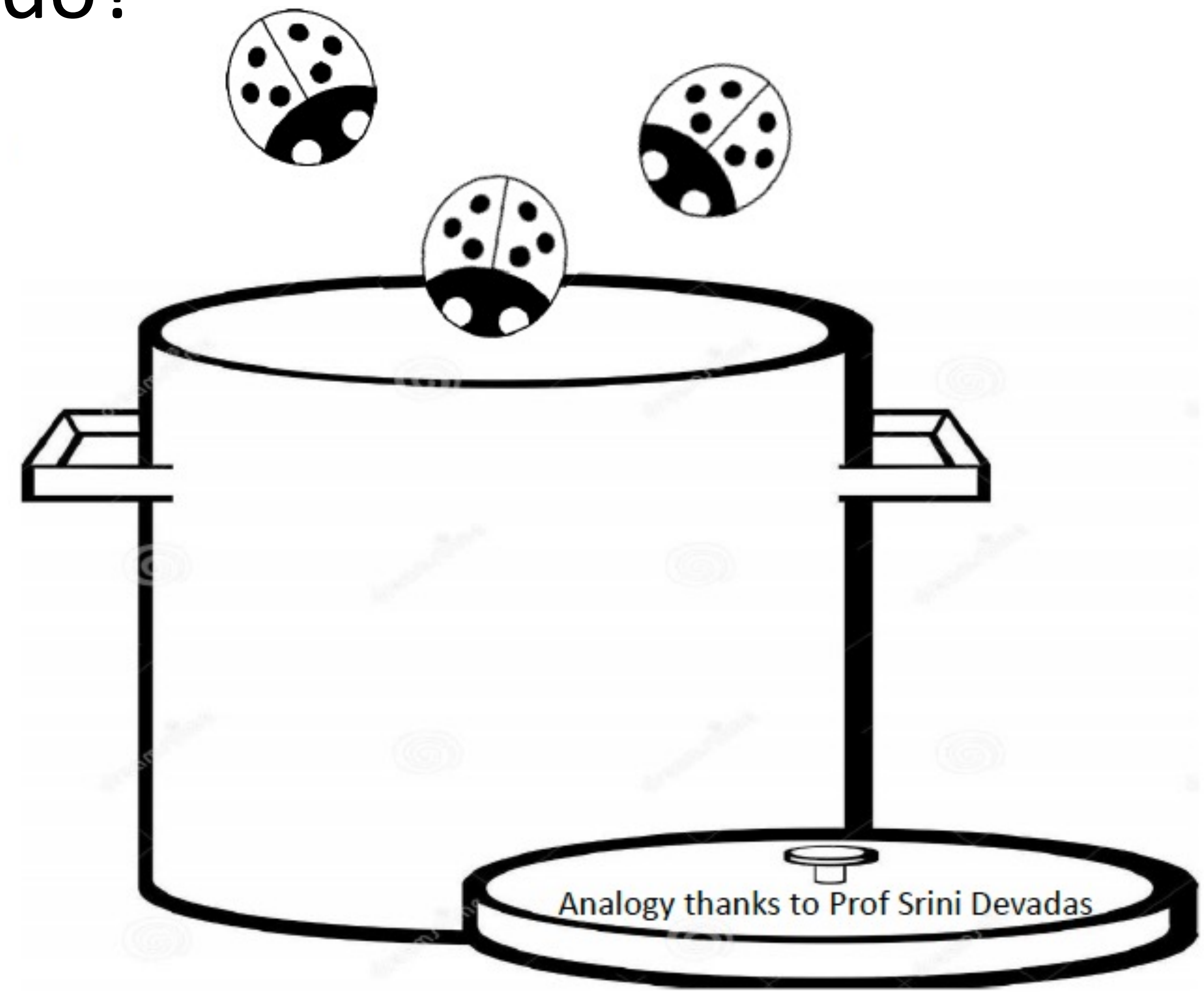
You are preparing a soup but bugs keep falling in from the ceiling. What do you do?



# High quality code – an analogy with soup

You are preparing a soup but bugs keep falling in from the ceiling. What do you do?

- Check the soup for bugs
    - *Testing*
- Keep the lid closed
    - *Defensive programming*
- Clean the kitchen
    - *Eliminating the source of bugs*



# Defensive programming

1. Write **specifications** for functions

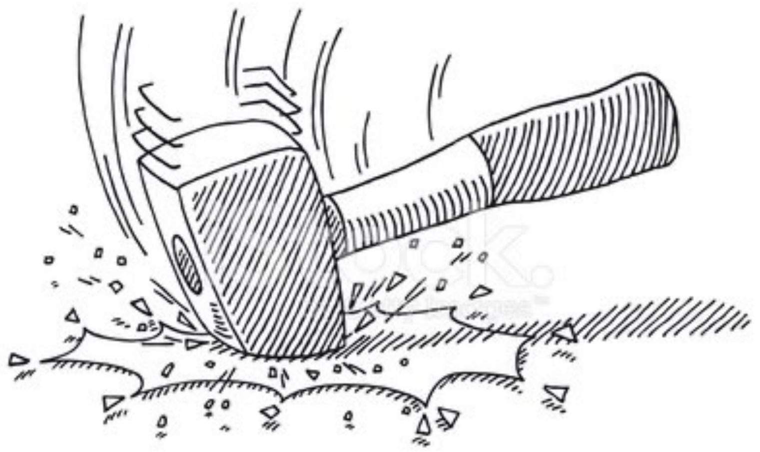
2. **Modularize** programs

3. Check conditions on inputs / outputs (**assertions**)

# Defensive programming

**A. Testing / Validation**

- Compare input / output pairs to what we expect
- Find out when things are not working
- Essentially, thinking " How can I break my program?"

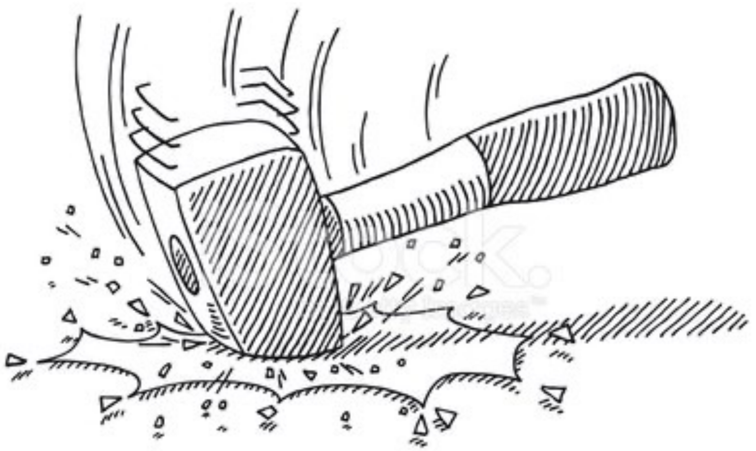


**B. Debugging**

- Study events leading up to an error
- Ask the question: "Why something is not working?"
- Essentially, thinking: "How can I fix my program?"

# How to make our code easy to test and debug?

- Code has to be easy to test by design
- Break a program up into modules that can be tested and debugged individually
- Document constraints on modules:
- - What do you expect the input to be?
- - What do you expect the output to be?
- Document assumptions behind the code design



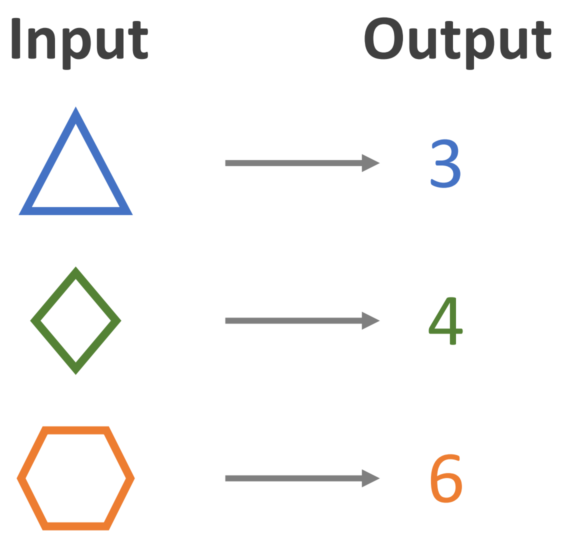
## When are we ready to test?

- Ensure first that code runs!
  - Remove **syntax** errors
  - Remove static **semantic** errors
  - Python **interpreters** can find those usually
- Have a set of expected results
  - An **input** set
  - For each input, the expected **output**

# When are we ready to test?

- Ensure first that code runs!
  - Remove **syntax** errors
  - Remove static **semantic** errors
  - Python **interpreters** can find those usually
- Have a set of expected results
  - An **input** set
  - For each input, the expected **output**

**Example.** *Test set for an algorithm that detects the number of edges of a geometrical object*



# What classes of tests do we have?

- **Unit testing**
  - Validates each piece of a program
  - Test each function separately
- **Regression testing**
  - Add test for bugs as you find them
  - Catch reintroduced errors that were previously fixed
- **Integration testing**
  - Does the overall program work?
  - Test to rush to do this

# Testing approaches

- **Intuition about natural boundaries to the problem**

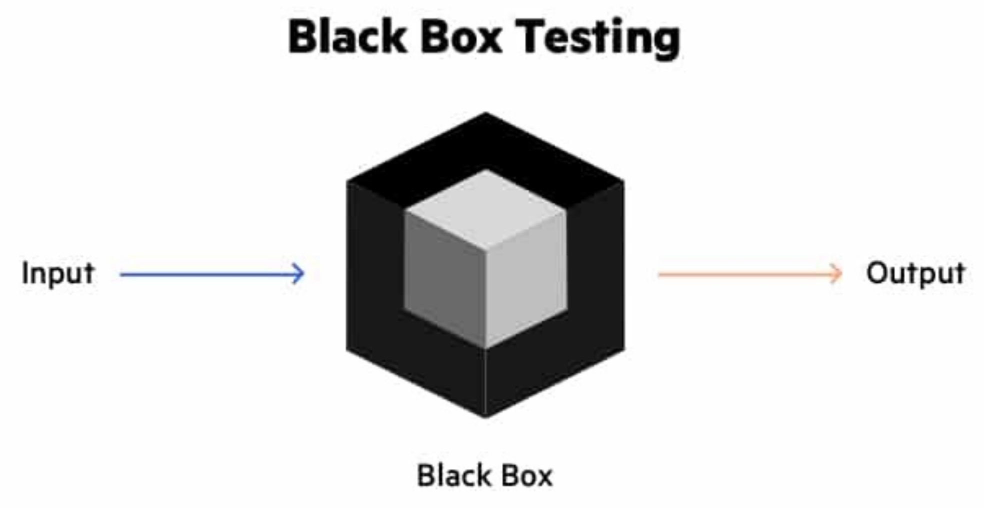
  - **def is_bigger(x,y):**
    **“assumes that x and y are ints**
    **returns TRUE if y is less than x, else FALSE”**

- **If no natural partitions exist: Random testing**
  - **Probability that code is correct increases with more tests**

# Testing approaches

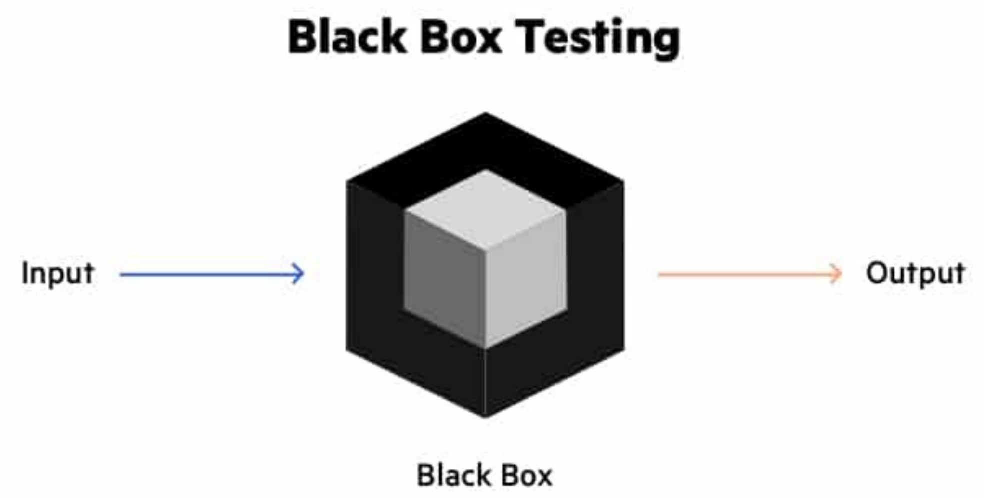
- **Intuition about natural boundaries to the problem**
  - **def is_bigger(x,y):**
    **“assumes that x and y are ints**
    **returns TRUE if y is less than x, else FALSE”**
- **If no natural partitions exist: Random testing**
  - **Probability that code is correct increases with more tests**
- **Black box testing**
  - Explore paths through specifications
- **Glass box testing**
  - Explore paths through code

# Black box testing




- Tests designed without looking at the code
- Tests can be done by someone other than the implementer (avoiding bias)
- Tests are reusable even if the implementation changes
- Partitioning solution space and building test cases
- Considering boundary conditions

# Black box testing

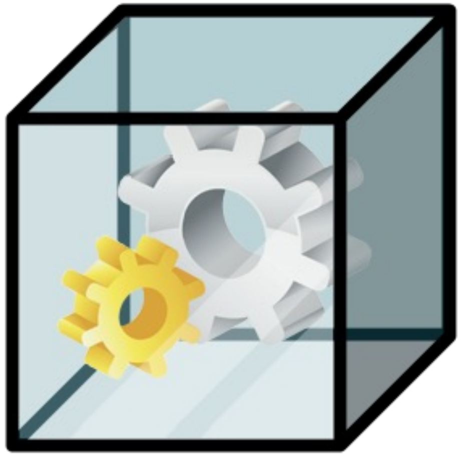



- Tests designed without looking at the code
- Tests can be done by someone other than the implementer (avoiding bias)
- Tests are reusable even if the implementation changes
- Partitioning solution space and building test cases
- Considering boundary conditions

```
def list_div(list1, list2):
“”” List division, assumes list1, list2 are numerical lists,
Returns list3  such that list3[i] = list1[i]/list2[i]”””
```

***What test cases can you think of?***

# Glass box testing

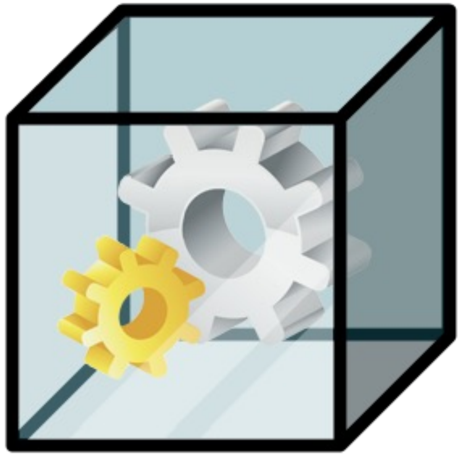


- Using code to guide and design our test cases
- "**Path-complete**" test suite will find test cases that go through every possible path in our code

- Example:
- it needs to execute all branches of a conditional
- For loops: test behavior when the loop is entered more than once; exactly once; not at all
- While loops: test cases that cover every possible way to exit the loop

# Glass box testing: Example

```python
def my_abs(x):
    """ Assumes x is an int
    Returns x if x>=0 and -x otherwise """
if x < -1:
    return -x
else:
    return x
```



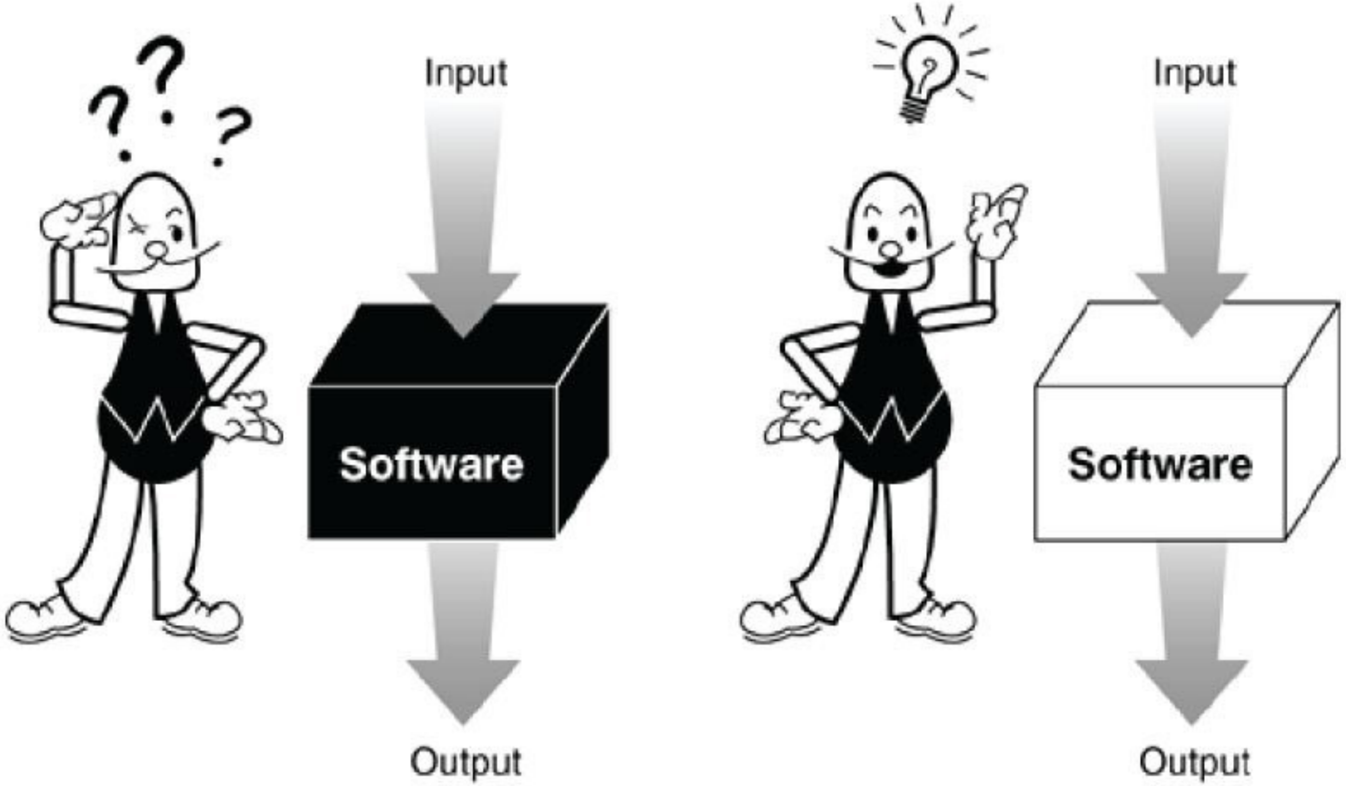
A “**path complete**” test suite : -3 and 3 <-- it can test both branches of IF-ELSE

However, a path complete test can still miss a bug:

`my_abs(-1)` would erroneously return `-1` !

We need to also test boundary cases

# Glass box vs. Black box testing




Source: https://blackboxlabz.com/how-to-define-and-compare-black-box-testing-vs-white-box-testing/

# Debugging


Source: https://flinthillsgroup.com/bug-free-custom-software-development-company-in-kansas/

- Steep learning curve
- **Goal: bug-free program!**

- Tools:
  - `print()`
  - Use logic
  - Be systematic!

# Debugging: print()


Source: https://flinthillsgroup.com/bug-free-custom-software-development-company-in-kansas/

- Good way to test our hypotheses

- Example: if we suspect that our code may have a mistake in a given line, we use a `print()` statement to clarify

- Example, we can print the dimensions of an array; the contents of a variable and so on

# When to print


Source: https://flinthillsgroup.com/bug-free-custom-software-development-company-in-kansas/

- When we enter a function
- Print parameters
- Print the results of a function

- We can put print halfway in our code
- Decide where bugs may be depending on values of the print

# Debugging steps


Source: https://flinthillsgroup.com/bug-free-custom-software-development-company-in-kansas/

- Study program code
  - Don’t ask what went wrong
  - Ask instead HOW did you receive unexpected result

- Parallel to scientific method
  - Study available data
  - Form hypothesis
  - Repeatable experiments
  - Pick simplest input to test with

# Today

1. How do we know if our code works?

2. **Exceptions and Assertions**

3. Unit testing

# Finding errors via error messages

**Typical error messages**

1. Trying to access beyond the limits of a list / array

```
test_list = [1,2,3]

test_list[3]
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-1-ba9cb747fb95> in <module>()
      1 test_list = [1,2,3]
      2
----> 3 test_list[3]

IndexError: list index out of range
```

# Finding errors via error messages

**Typical error messages**

1. Trying to access beyond the limits of a list / array
2. **Trying to convert an inappropriate type**

```
int(test_list)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-3-06c68ac376d6> in <module>()
----> 1 int(test_list)

TypeError: int() argument must be a string, a bytes-like object or a number, not 'list'
```

# Finding errors via error messages

**Typical error messages**

1. Trying to access beyond the limits of a list / array
2. Trying to convert an inappropriate type
3. **Referencing a non-existent variable**

```
x
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-4-6fcf9dfbd479> in <module>()
----> 1 x

NameError: name 'x' is not defined
```

# Finding errors via error messages

**Typical error messages**

1. Trying to access beyond the limits of a list / array
2. Trying to convert an inappropriate type
3. Referencing a non-existent variable
4. **Mixing data types**

```
'3'/4
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-5-9d0caf584989> in <module>()
----> 1 '3'/4

TypeError: unsupported operand type(s) for /: 'str' and 'int'
```

# Finding errors via error messages

**Typical error messages**

1. Trying to access beyond the limits of a list / array
2. Trying to convert an inappropriate type
3. Referencing a non-existent variable
4. Mixing data types
5. **Forgetting to close parentheses or quotations:**

```
x = len([4, 5, 2]
print(x)
```

```
  File "<ipython-input-7-a324e7704639>", line 2
    print(x)
        ^
SyntaxError: invalid syntax
```

# Finding errors via error messages: types of errors

**Typical error messages**

1. Trying to access beyond the limits of a list / array `IndexError`
2. Trying to convert an inappropriate type `TypeError`
3. Referencing a non-existent variable `NameError`
4. Mixing data types `TypeError`
5. Forgetting to close parentheses or quotations `SyntaxError`

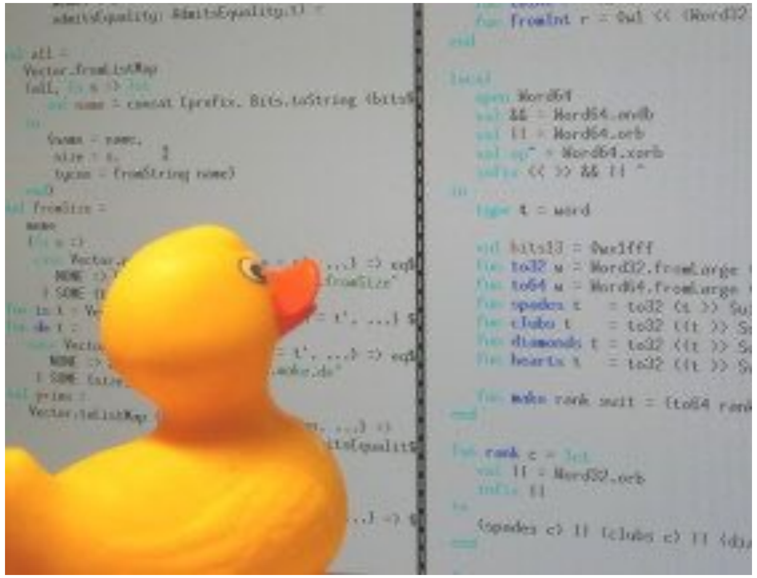
Error messages are easy to find

Relatively easy to fix

# Finding errors in logic

**Logic errors are hard to identify**

1. Think through before writing new code
2. Design your logic: draw pictures, draw your algorithm
3. Explain the code to someone else



Rubber duck debugging

# How to proceed in practice?

**DON'T:**

- Write entire program
- Test entire program at once
- Debug entire program

**Do:**

- Write one function
- Test the function
- Debug the function
- Write one function
- Test the function
- Debug the function
- …
- Repeat…

- ***Do integration testing***

# How to proceed in practice?

**DON'T:**

- Change code
- Remember where a bug was
- Test code
- Forget where the bug was / what change you made

**Do:**

- Backup code (Version control)
- Change code
- Write suspected bug in a comment
- Test code
- Compare new with old versions

# Exceptions and assertions

**What happens when an unexpected condition is met while we execute code?**

**We get an exception** to expected conditions

```
IndexError
TypeError
NameError
SyntaxError
```

Python can provide handlers to deal with exceptions

# Dealing with exceptions: handlers

Python can provide handlers to deal with exceptions

```
try:
    num1 = int(input("Enter first number: "))
    num2 = int(input("Enter second number: "))

    print("num1/num2 = ", num1/num2)
except:
    print("Something went wrong...")
```

Any exception that is raised in the “try” part will be handled by the except statement.

Code execution does not stop upon error, but continues with the body of the except statement

# Dealing with exceptions: handlers

Python can provide handlers to deal with exceptions

```python
try:
    num1 = int(input("Enter first number: "))
    num2 = int(input("Enter second number: "))

    print("num1/num2 = ", num1/num2)
except:
    print("Something went wrong...")
```

Input "l" cannot be converted to `int()`

```
staff-214-233:UnitTesting athina$ python exceptions_example.py
Enter first number: 10
Enter second number: 2
num1/num2 =  5.0
```

# Dealing with exceptions: handlers

Python can provide handlers to deal with exceptions

```
try:
    num1 = int(input("Enter first number: "))
    num2 = int(input("Enter second number: "))

    print("num1/num2 = ", num1/num2)
except:
    print("Something went wrong...")
```

Input “I” cannot be converted to `int()`

```
staff-214-233:UnitTesting athina$ python exceptions_example.py
Enter first number: 10
Enter second number: 2
num1/num2 =  5.0
staff-214-233:UnitTesting athina$ python exceptions_example.py
Enter first number: I
Something went wrong...
```

# Dealing with exceptions: handlers

We can have separate “except” clauses to deal with different exception types

```python
try:
    num1 = int(input("Enter first number: "))
    num2 = int(input("Enter second number: "))

    print("num1/num2 = ", num1/num2)
except ValueError:
    print("Input cannot be converted to a number.")
except ZeroDivisionError:
    print("Cannot divide by zero.")
except:
    print("Something went wrong...")
```

**Execute if these error types appear**

**Executes for all other error types**

# Dealing with exceptions: handlers

We can have separate “except” clauses to deal with different exception types

```
[staff-214-233:UnitTesting athina$ python exceptions_example.py
Enter first number: 10
Enter second number: 2
num1/num2 =  5.0
[staff-214-233:UnitTesting athina$ python exceptions_example.py
Enter first number: 10
Enter second number: p
Input cannot be converted to a number.
[staff-214-233:UnitTesting athina$ python exceptions_example.py
Enter first number: 10
Enter second number: 0
Cannot divide by zero.
```

ValueError

ZeroDivisionError

# Dealing with exceptions: else / finally

**else:**

body of else will be executed when execution of associated “try” body completes with no exceptions

**finally:**

body will be always executed after ”try” “else” and “except” clauses, even if they raised another error or executed a ”break” “continue” “return”

useful for cleaning-up and doing operations that must be done no matter what e.g. save and/or close a file

# What do we do with exceptions?

How do we want to deal with errors?

1. Fail in silence:

    we can substitute default values or just continue

    however, it gives no warning to the user

    if the ”except” message or body is not informative enough it can be hard to debug a program as user

2. return an ‘error’ value:

    what values do we choose?

    complicates code by having to search for one special value

3. stop execution and signal error condition

    raise an exception:

    **raise Exception(“Descriptive exception”)**

# Assertions

Simplest type of tests

”Bound” acceptable behavior during runtime

```
>>> assert True == False
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AssertionError
>>> assert True == True
```

*Code execution will halt if the comparison is false*

*Nothing happens if comparison is true*

# When are Assertions useful?

"Guarding" our code against unintended input:

```python
def mean_list(num_list):
    """ Computing the mean of a list"""

    # Assert that the list is not empty:
    assert len(num_list) != 0

    # Return the mean:
    return sum(num_list)/len(num_list)


print(mean_list([0,1,3,1]))

print(mean_list([]))
```

*Asserting that a list is not empty*

```
staff-214-233:UnitTesting athina$ python assertions_example.py
1.25
Traceback (most recent call last):
  File "assertions_example.py", line 13, in <module>
    print(mean_list([]))
  File "assertions_example.py", line 5, in mean_list
    assert len(num_list) != 0
AssertionError
```

*AssertionError is raised*

# Today

1. How do we know if our code works?

2. Exceptions and Assertions

3. **Unit testing**

# UNIT TESTING

***Testing individual pieces of code (units) to determine if they are fit for use***

# Unit testing: the recipe

**Design:**

- The developer defines criteria in test scripts; these can be based on previous knowledge of expected test cases

**Implementation & Execution:**

- Tests that fail any of the criteria will be logged

**Output:**

- a report that describes which tests failed any criterion

# Unit testing: what do we need

Automated test scripts; these ensure that all code sections **-each unit-** meet their objectives and show expected behavior

In python: built-in framework: `unittest`

Goal: render our code future proof, by anticipating cases where it can fail or result in a bug!

# What is a unit?

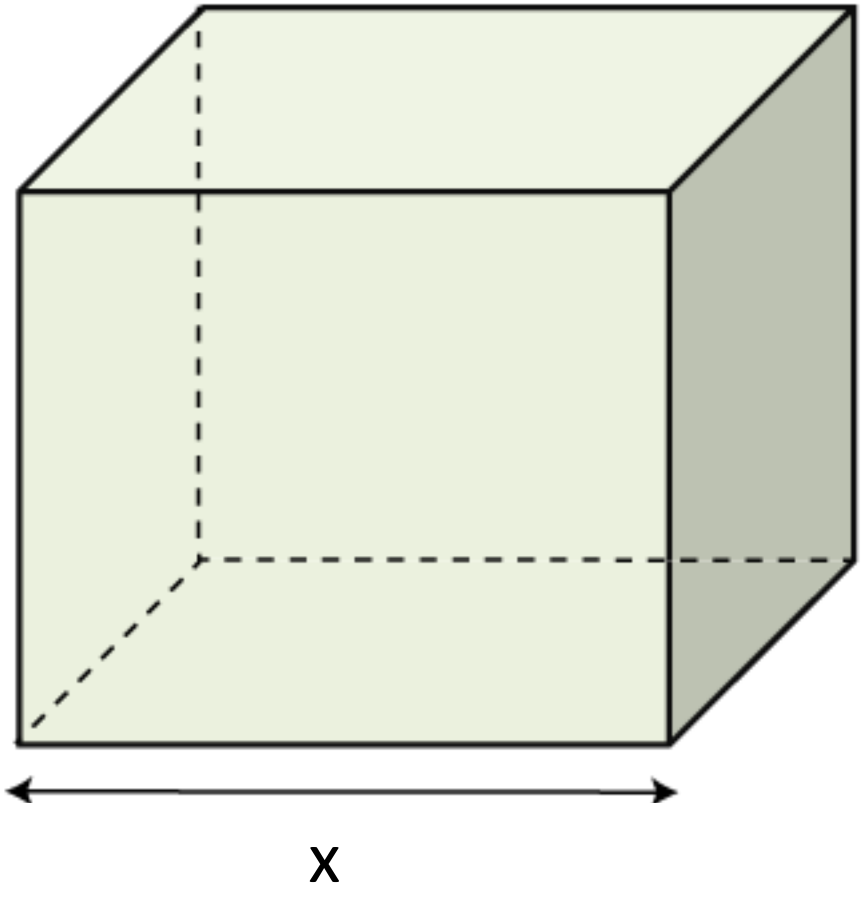
- An entire module
- An individual function
- A complete interface: class or method

# How do we start?

- start with the smallest testable unit of your code
- move on to other units
- see how that smallest unit interacts with other units

# Example

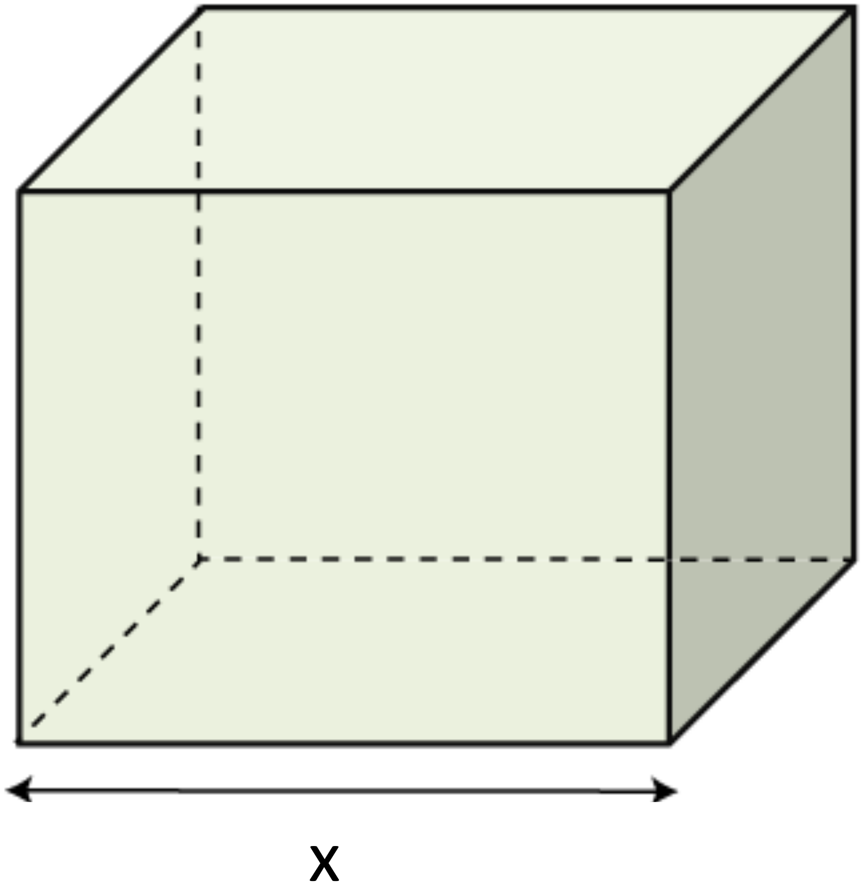
- We want to compute the volume of a cube, given the edge, “x”:




Volume = $x^3$

# Example

- We want to compute the volume of a cube, given the edge, “x”
- We write a function ”volume_cube(x)” that does the following calculation:




$$\text{Volume} = x^3$$

compute_volume.py

```python
def volume_cube(x):

    return(x*x*x)

```

# Example: trying out likely cases

- We want to compute the volume of a cube, given the edge, “x”
- We write a function ”volume_cube(x)” that does the following calculation:

$$\text{Volume} = x^3$$

- We call the function with different examples:

```
from compute_volume import volume_cube

# We try a few examples:
x = 1
print("The volume of a cube with edge ", str(x), " is ", volume_cube(x))
```

```
The volume of a cube with edge  1  is  1
```

```
x = 3
print("The volume of a cube with edge ", str(x), " is ", volume_cube(x))
```

```
The volume of a cube with edge  3  is  27
```

```
x = 0
print("The volume of a cube with edge ", str(x), " is ", volume_cube(x))
```

```
The volume of a cube with edge  0  is  0
```

**All these seem to work just fine!**

**Do we need more tests?**

# Trying out unlikely cases

- Next, we call our function with some unlikely cases:

```
x = -5
print("The volume of a cube with edge ", str(x), " is ", volume_cube(x))
```

```
The volume of a cube with edge  -5  is  -125
```

```
x = 2+9j
print("The volume of a cube with edge ", str(x), " is ", volume_cube(x))
```

```
The volume of a cube with edge  (2+9j)  is  (-478-621j)
```

What do you observe here?

# Trying out unlikely cases

- Next, we call our function with some unlikely cases:

```
x = -5
print("The volume of a cube with edge ", str(x), " is ", volume_cube(x))
```

```
The volume of a cube with edge  -5  is  -125
```

```
x = 2+9j
print("The volume of a cube with edge ", str(x), " is ", volume_cube(x))
```

```
The volume of a cube with edge  (2+9j)  is  (-478-621j)
```

- There is no error
- Our function gives some output
- The output is numerically correct, but it is a physical impossibility

# Trying out unlikely cases

- Last, we can call our function with an input of wrong type:

```
x = "two"
print("The volume of a cube with edge ", str(x), " is ", volume_cube(x))
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call la
st)
<ipython-input-16-53d6816e5376> in <module>()
      1 x = "two"
----> 2 print("The volume of a cube with edge ", str(x), " is ", volume_
cube(x))

~/Documents/Teaching/Python_BioInf2021/Examples/compute_volume.py in vol
ume_cube(x)
      1 def volume_cube(x):
      2 
----> 3     return(x*x*x)

TypeError: can't multiply sequence by non-int of type 'str'
```

# The importance of having a unit test: diverse examples of 'failure'

Incorrect cases:

1. The volume cannot be negative

2. The volume cannot be a complex number

**These two "succeeded" even though the output is not physically possible!**

3. We cannot multiply strings, so we are getting a **TypeError**

**This problem resulted in an error**

# Unit tests to the rescue

Unit tests can be written in a separate code in a different file

Naming conventions to follow for our example:

- **compute_volume_test.py**
- **test_compute_volume.py**

***test + name of unit***
***name of unit + test***

# Example of a unit test

**The function we want to test:**

compute_volume.py

```python
def volume_cube(x):

    return(x*x*x)

```

**Our first unit test:**

compute_volume_test.py

```python
from compute_volume import *
import unittest

class TestCubes(unittest.TestCase):
    def test_volume_cube(self):
        self.assertAlmostEqual(volume_cube(1), 1)
        self.assertAlmostEqual(volume_cube(0), 0)
        self.assertAlmostEqual(volume_cube(3), 27)
        self.assertAlmostEqual(volume_cube(3.5), 42.875)
```

# Our first unit test

The .py file where we saved the function we are testing

The function we are testing

The unit test python library

compute_volume_test.py

```
from compute_volume import *
import unittest

class TestCubes(unittest.TestCase):
    def test_volume_cube(self):
        self.assertAlmostEqual(volume_cube(1), 1)
        self.assertAlmostEqual(volume_cube(0), 0)
        self.assertAlmostEqual(volume_cube(3), 27)
        self.assertAlmostEqual(volume_cube(3.5), 42.875)
```

A set of correct input / outputs

# Running the first unit test: `assertAlmostEqual`

```
!python -m unittest compute_volume_test.py
.
----------------------------------------------------------------------
Ran 1 test in 0.000s

OK
```

**Success!**

compute_volume_test.py

```python
from compute_volume import *
import unittest

class TestCubes(unittest.TestCase):
    def test_volume_cube(self):
        self.assertAlmostEqual(volume_cube(1), 1)
        self.assertAlmostEqual(volume_cube(0), 0)
        self.assertAlmostEqual(volume_cube(3), 27)
        self.assertAlmostEqual(volume_cube(3.5), 42.875)
```

# When a unit test fails

Consider the following case: we re-write the main function that computes the volume of a cube, but there is a mistake.

What will the unit test do?

compute_volume.py

```
def volume_cube(x):

    return(x*x)
```

**Mistake in the formula that calculates volume!**

compute_volume_test.py

```
from compute_volume import *
import unittest

class TestCubes(unittest.TestCase):
    def test_volume_cube(self):
        self.assertAlmostEqual(volume_cube(1), 1)
        self.assertAlmostEqual(volume_cube(0), 0)
        self.assertAlmostEqual(volume_cube(3), 27)
        self.assertAlmostEqual(volume_cube(3.5), 42.875)
```

# When a unit test fails

Consider the following case: we re-write the main function that computes the volume of a cube, but there is a mistake.

What will the unit test do?

```
!python -m unittest compute_volume_test.py
```

```
F
======================================================================
FAIL: test_volume_cube (compute_volume_test.TestCubes)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/Users/athina/Documents/Teaching/Python_BioInf2021/Examples/Unit Testing/compute_volume_test.py", line 8, in test_volume_cube
    self.assertAlmostEqual(volume_cube(3), 27)
AssertionError: 9 != 27 within 7 places

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

FAILED (failures=1)
```

compute_volume_test.py

```python
from compute_volume import *
import unittest

class TestCubes(unittest.TestCase):
    def test_volume_cube(self):
        self.assertAlmostEqual(volume_cube(1), 1)
        self.assertAlmostEqual(volume_cube(0), 0)
        self.assertAlmostEqual(volume_cube(3), 27)
        self.assertAlmostEqual(volume_cube(3.5), 42.875)
```

**The first 2 tests pass;**
**The third test fails!**

We see the reason for the failure and the number of failures that our code has

# Another assert test: `assertRaises`

assertRaises: it can find out if our function handles input values correctly

compute_volume_test.py

```python
from compute_volume import *
import unittest

class TestCubes(unittest.TestCase):
    def test_volume_cube(self):
        self.assertAlmostEqual(volume_cube(1), 1)
        self.assertAlmostEqual(volume_cube(0), 0)
        self.assertAlmostEqual(volume_cube(3), 27)
        self.assertAlmostEqual(volume_cube(3.5), 42.875)

    def test_input_value(self):
        self.assertRaises(TypeError, volume_cube, True)
```

**We add a new test**

**assertRaises:** we test whether our function handles the class or type of input e.g. if we enter as input a string, will it be handled as an exception? With an if condition? Will it raise an error?

# Another assert test: `assertRaises`

assertRaises: it can find out if our function handles input values correctly

```
!python -m unittest compute_volume_test.py
```

```
F.
======================================================================
FAIL: test_input_value (compute_volume_test.TestCubes)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/Users/athina/Documents/Teaching/Python_BioInf2021/Examples/Unit
Testing/compute_volume_test.py", line 12, in test_input_value
    self.assertRaises(TypeError, volume_cube, True)
AssertionError: TypeError not raised by volume_cube

----------------------------------------------------------------------
Ran 2 tests in 0.000s

FAILED (failures=1)
```

compute_volume_test.py

```python
from compute_volume import *
import unittest

class TestCubes(unittest.TestCase):
    def test_volume_cube(self):
        self.assertAlmostEqual(volume_cube(1), 1)
        self.assertAlmostEqual(volume_cube(0), 0)
        self.assertAlmostEqual(volume_cube(3), 27)
        self.assertAlmostEqual(volume_cube(3.5), 42.875)

    def test_input_value(self):
        self.assertRaises(TypeError, volume_cube, True)
```

**The test fails!**

Our code volume_cube does not check whether the input is passed to it properly

**How do we fix that?**

# How to avoid `assertRaises` failures

`assertRaises`: it can find out if our function handles input values correctly
We need to control if the input to our functions is passed on properly!

compute_volume.py

```
def volume_cube(x):
    if type(x) not in [int, float]:
        raise TypeError("The edge of the cube can only be an integer or float.")

    return(x*x*x)
```

Revisiting our initial function "volume_cube"

It now checks the type of input data
If that is not acceptable (int / float in this case), it raises a TypeError and exits

# How to avoid `assertRaises` failures

`assertRaises`: it can find out if our function handles input values correctly
We need to control if the input to our functions is passed on properly!

compute_volume.py

```
def volume_cube(x):
    if type(x) not in [int, float]:
        raise TypeError("The edge of the cube can only be an integer or float.")

    return(x*x*x)
```

**Revised function**

```
!python -m unittest compute_volume_test.py
```

```
..
----------------------------------------------------------------------
Ran 2 tests in 0.000s

OK
```

**Unit Test:**
**Successful now!**

# How many unit tests do we run?

Test methods should start with they keyword “test”
If this keyword is not present, the test method will not run. Example:

**No keyword “test”**

**Keyword “test”**

```
from compute_volume import *
import unittest

class TestCubes(unittest.TestCase):
    def volume_cube(self):
        self.assertAlmostEqual(volume_cube(1), 1)
        self.assertAlmostEqual(volume_cube(0), 0)
        self.assertAlmostEqual(volume_cube(3), 27)
        self.assertAlmostEqual(volume_cube(3.5), 42.875)

    def test_input_value(self):
        self.assertRaises(TypeError, volume_cube, True)
```

# How many unit tests do we run?

Test methods should start with they keyword “test”
If this keyword is not present, the test method will not run. Example:

```
!python -m unittest compute_volume_test.py
```

```
.
----------------------------------------------------------------------
Ran 1 test in 0.000s

OK
```

```python
from compute_volume import *
import unittest

class TestCubes(unittest.TestCase):
    def volume_cube(self):
        self.assertAlmostEqual(volume_cube(1), 1)
        self.assertAlmostEqual(volume_cube(0), 0)
        self.assertAlmostEqual(volume_cube(3), 27)
        self.assertAlmostEqual(volume_cube(3.5), 42.875)

    def test_input_value(self):
        self.assertRaises(TypeError, volume_cube, True)
```

**Only 1 test ran!**

Only the test that starts with keyword “test” runs. The other test is ignored

# How do we make the output of unit tests more humanly understandable?

In .assertEqual : we can add a string that explains the AssertionError and desired output

compute_volume.py

```python
def volume_cube(x):
    if type(x) not in [int, float]:
        raise TypeError("The edge of the cube can only be an integer or float.")

    return(x*x)
```

**(back to the) Mistake in the formula that calculates volume!**

compute_volume_test.py

```python
from compute_volume import *
import unittest

class TestCubes(unittest.TestCase):
    def test_volume_cube(self):
        self.assertEqual(volume_cube(1), 1, "Should be 1")
        self.assertEqual(volume_cube(0), 0, "Should be 0")
        self.assertEqual(volume_cube(3), 27, "Should be 27")
        self.assertEqual(volume_cube(3.5), 42.875, "Should be 42.875")

    def test_input_value(self):
        self.assertRaises(TypeError, volume_cube, True)
```

**We specify the desired and correct output**

# How do we make the output of unit tests more humanly understandable?

In .assertEqual : we can add a string that explains the AssertionError and desired output

```
!python -m unittest compute_volume_test.py
```

```
.F
======================================================================
FAIL: test_volume_cube (compute_volume_test.TestCubes)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/Users/athina/Documents/Teaching/Python_BioInf2021/Examples/Unit
Testing/compute_volume_test.py", line 8, in test_volume_cube
    self.assertEqual(volume_cube(3), 27, "Should be 27")
AssertionError: 9 != 27 : Should be 27

----------------------------------------------------------------------
Ran 2 tests in 0.000s

FAILED (failures=1)
```

**Our specified output is displayed when a test fails**

compute_volume_test.py

```
1  from compute_volume import *
2  import unittest
3
4  class TestCubes(unittest.TestCase):
5      def test_volume_cube(self):
6          self.assertEqual(volume_cube(1), 1, "Should be 1")
7          self.assertEqual(volume_cube(0), 0, "Should be 0")
8          self.assertEqual(volume_cube(3), 27, "Should be 27")
9          self.assertEqual(volume_cube(3.5), 42.875, "Should be 42.875")
10
11     def test_input_value(self):
12         self.assertRaises(TypeError, volume_cube, True)
```

**We specify the desired and correct output**

# Summary: Unit tests in python

Testing individual units against likely and unlikely cases

They can help identify errors and bugs in the code that would not necessarily lead to an actual error while running the code

First step, then, Integration testing → combining individual modules and testing them as a group

# Further reading and notes

**Continuous integration on github:**
https://docs.github.com/en/actions/automating-builds-and-tests/about-continuous-integration

**unittest:**
https://docs.python.org/3/library/unittest.html

# Today

1. How do we know if our code works?

2. Exceptions and Assertions

3. Unit testing