Testing

How to write correct code and go back to enjoy your life

Correct code is a code without bugs or errors

bug

> there is a bug in the code when this code behave differently from what it's written in its documentation

In the documentation we include the external documentation (**the manual**) the internal one (**docstrings**) and the implicit one (**name** of the function and of its **arguments**)

Can be considered bugs also **comments** and **variable names** that do not align with what the code is doing, even if less so.

the most bugged function in history

> **REQUIREMENT**: write a function that integrate a parabola between two positions

```
def multiplication(first_name, last_name):
    """this function divides two numbers"""
    # perform the subtraction
    exponent = first_name[last_name]
    return exponent
```

the code runs fine without any errors (if I give the proper parameters), but using it in real life would be suicidal!

error

> a code has an error when behaves differently from what it was written to do (logic and behavior differ)

For example, a sorting algorithm that does not sort, or that in some corner cases sort incorrectly

good documentation

Take the documentation of numpy, it's basically as good as it gets

In [7]:
```
import numpy
print("\n".join(numpy.linalg.eig.__doc__.splitlines()[:24]))
```

```
Compute the eigenvalues and right eigenvectors of a square array.

Parameters
----------
a : (..., M, M) array
    Matrices for which the eigenvalues and right eigenvectors will
    be computed

Returns
-------
w : (..., M) array
    The eigenvalues, each repeated according to its multiplicity.
    The eigenvalues are not necessarily ordered. The resulting
    array will be of complex type, unless the imaginary part is
    zero in which case it will be cast to a real type. When `a`
    is real the resulting eigenvalues will be real (0 imaginary
    part) or occur in conjugate pairs

v : (..., M, M) array
    The normalized (unit "length") eigenvectors, such that the
    column ``v[:,i]`` is the eigenvector corresponding to the
    eigenvalue ``w[i]``.
```

In [8]:
```
print("\n".join(numpy.linalg.eig.__doc__.splitlines()[24:39]))
```

```
Raises
------
LinAlgError
    If the eigenvalue computation does not converge.

See Also
--------
eigvals : eigenvalues of a non-symmetric array.

eigh : eigenvalues and eigenvectors of a real symmetric or complex
       Hermitian (conjugate symmetric) array.

eigvalsh : eigenvalues of a real symmetric or complex Hermitian
           (conjugate symmetric) array.
```

In [9]:
```
print("\n".join(numpy.linalg.eig.__doc__.splitlines()[39:70]))
```

```
    Notes
    -----

    .. versionadded:: 1.8.0

    Broadcasting rules apply, see the `numpy.linalg` documentation for
    details.

    This is implemented using the _geev LAPACK routines which compute
    the eigenvalues and eigenvectors of general square arrays.

    The number `w` is an eigenvalue of `a` if there exists a vector
    `v` such that ``dot(a,v) = w * v``. Thus, the arrays `a`, `w`, and
    `v` satisfy the equations ``dot(a[:,:], v[:,i]) = w[i] * v[:,i]``
    for :math:`i \in \{0,...,M-1\}`.

    The array `v` of eigenvectors may not be of maximum rank, that is, som
e
    of the columns may be linearly dependent, although round-off error may
    obscure that fact. If the eigenvalues are all different, then theoreti
cally
    the eigenvectors are linearly independent. Likewise, the (complex-valu
ed)
    matrix of eigenvectors `v` is unitary if the matrix `a` is normal, i.
e.,
    if ``dot(a, a.H) = dot(a.H, a)``, where `a.H` denotes the conjugate
    transpose of `a`.

    Finally, it is emphasized that `v` consists of the *right* (as in
    right-hand side) eigenvectors of `a`.  A vector `y` satisfying
    ``dot(y.T, a) = z * y.T`` for some number `z` is called a *left*
    eigenvector of `a`, and, in general, the left and right eigenvectors
    of a matrix are not necessarily the (perhaps conjugate) transposes
    of each other.
```

In [13]:
```
print("\n".join(numpy.linalg.eig.__doc__.splitlines()[70:96]))
```

```
    References
    ----------
    G. Strang, *Linear Algebra and Its Applications*, 2nd Ed., Orlando, F
L,
    Academic Press, Inc., 1980, Various pp.

    Examples
    --------
    >>> from numpy import linalg as LA

    (Almost) trivial example with real e-values and e-vectors.

    >>> w, v = LA.eig(np.diag((1, 2, 3)))
    >>> w; v
    array([ 1.,  2.,  3.])
    array([[ 1.,  0.,  0.],
           [ 0.,  1.,  0.],
           [ 0.,  0.,  1.]])

    Real matrix possessing complex e-values and e-vectors; note that the
    e-values are complex conjugates of each other.

    >>> w, v = LA.eig(np.array([[1, -1], [1, 1]]))
    >>> w; v
    array([ 1. + 1.j,  1. - 1.j])
    array([[ 0.70710678+0.j        ,  0.70710678+0.j        ],
           [ 0.00000000-0.70710678j,  0.00000000+0.70710678j]])
```

In [15]:
```
print("\n".join(numpy.linalg.eig.__doc__.splitlines()[96:120]))
```

```
    Complex-valued matrix with real e-values (but complex-valued e-vector
s);
    note that a.conj().T = a, i.e., a is Hermitian.

    >>> a = np.array([[1, 1j], [-1j, 1]])
    >>> w, v = LA.eig(a)
    >>> w; v
    array([  2.00000000e+00+0.j,   5.98651912e-36+0.j]) # i.e., {2, 0}
    array([[ 0.00000000+0.70710678j,  0.70710678+0.j        ],
           [ 0.70710678+0.j        ,  0.00000000+0.70710678j]])

    Be careful about round-off error!

    >>> a = np.array([[1 + 1e-9, 0], [0, 1 - 1e-9]])
    >>> # Theor. e-values are 1 +/- 1e-9
    >>> w, v = LA.eig(a)
    >>> w; v
    array([ 1.,  1.])
    array([[ 1.,  0.],
           [ 0.,  1.]])
```

Purity of functions

A function is called pure if

1. with the same input, it returns the same output (it is deterministic)
2. it does not change the state of the rest of the program.

This lets you:

1. know that once you've proven that function is correct, it will always be correct
2. if you change the implementation, as long as the input-output relationship is the same, the functions that depend on it are going to be fine

pure functions are very useful, in particular because they are the easier to test and prove correct

Using global variables (or non local ones) inside a function stops it from being pure.

types of test

we want to test if a function is correct, what kind of test can be done?

- **advancement test**: edits to the function introduce the new features I deside
- **regressions test**: edits to the function do not lose functionality that other code relies on

- **positive tests**: the code does the things I expect it to do when I give the right parameters
- **negative tests**: the code fail the way I expect it to when I give the wrong parameters

Strategies of testing

to test the correctness of a single function

- informal testing
- unit testing (anecdotal testing)
- property testing

Informal tests

when we write a function, we usually test if it is working as we expect

this is a form of testing, and is necessary, but if far from enough

In [33]:
```
def inc(x):
    return x + 1

assert inc(3)==4
assert inc(5)==4
assert inc(6)==4
```

```
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
<ipython-input-33-a12389ee159e> in <module>()
      3
      4 assert inc(3)==4
----> 5 assert inc(5)==4
      6 assert inc(6)==4

AssertionError:
```

Unit testing (anecdotal tests)

What you tested with just bare asserts, you save as a separate script

any time you modify the code, run the tests to check that everything is still working according to plan.

Do not commit code that do not pass all the tests!

If you find a new bug:

1. write a test that reproduce that bug (i.e. that fails for that case)
2. adjust the function until the test passes
3. keep the test there forever to avoid regressions in the future

As a general rule, I need at least one example of a typical case use, plus one for any limit case.

Imagine to be writing a function that sort a list. You would need to test at least:

1. that an out of order list, such as `[1, 3, 2]`, get sorted `[1, 2, 3]`
2. an empty list gives back an empty list
3. an already sorted list `[1, 2, 3]` gives back the same list as output

A good library to start with **unit testing** is **pytest**.

pytest is a command line program that:

1. search all the current directory for `*.py` files
2. in each one of those, search all the functions named `test_<something>`
3. execute all of them and keep track of the results
4. prints a summary of the tests

In [2]:
```
%cd ~
```

```
/home/enrico
```

In [34]:
```
%%file test_prova.py
def inc(x):
    return x + 1

def test_answer_1():
```

```
    assert inc(3) == 5

def test_answer_2():
    assert inc(7) == 7
```

```
Overwriting test_prova.py
```

In [32]:
```
!pytest test_prova.py
```

```
============================= test session starts ==============================
platform linux -- Python 3.6.0, pytest-3.0.6, py-1.4.32, pluggy-0.4.0
rootdir: /home/enrico, inifile:
plugins: xonsh-0.5.6, hypothesis-3.6.1
collected 2 items

test_prova.py FF

=================================== FAILURES ===================================
_____________________________ test_answer_1 ______________________________

    def test_answer_1():
>       assert inc(3) == 5
E       assert 4 == 5
E        +  where 4 = inc(3)

test_prova.py:5: AssertionError
_____________________________ test_answer_2 ______________________________

    def test_answer_2():
>       assert inc(7) == 7
E       assert 8 == 7
E        +  where 8 = inc(7)

test_prova.py:8: AssertionError
=========================== 2 failed in 0.03 seconds ===========================
```

One could execute the test code by hand, but there would be several disadvantages

1. one would have to execute each function one by one (instead of having them discovered byt pytest)
2. at the first error, the whole execution would stop, giving me a partial view of the situations; pytest visualizes all the errors
3. the bare *assert* results are not very informative; pytest results are more explicit

Pytest can also check if we expect the function to raises an exception under certain circumstances.

This would be quite difficult to do with normal code

In [ ]:
```
import pytest

def test_zero_division():

```

```
    with pytest.raises(ZeroDivisionError):
        1 / 0
```

Test driven development

In the **test driven development** code and tests are written together, starting from the specifics.

There are several variants of this concept, but the main idea is that you shouldn't wait to finish your code to write your tests.

Write them alongside, even before the code itself, so that you can be sure that the function follows what you expect it to do.

When designing complicated architectures, where the design is necessarely top-down, this is a precious approach

In the top-down approach, we will develop our code as if everything works perfectly, and then we will implement it for real

1. we start from the "final" function, calling the others as if they exist
2. implement a "stub" function of each one, that does nothing aside of allowing us to execute our code
3. write tests that represents the properties that we expect from our real function
4. replace the stubs with functions that actually have these properties

Cellular Automata

Let's try to implement a simple cellular automata, based on the rules detailed by Wolfram

Our system is represented by a string of 0 and empty spaces (alive and dead cells), and the evolution is based on the status of each characted and its neighbours

We will use rule 30, inspired by this blog

In [38]:
```
rule30 = {"000": '.',
          "00.": '.',
          "0.0": '.',
          "...": '.',
          "0..": '0',
          ".00": '0',
          ".0.": '0',
          "..0": '0',
         }
```

In [22]:
```
def simulation(nsteps):
    initial_state = generate_state()
    states_seq = [initial_state]
    for i in range(nsteps):
        old_state = states_seq[-1]
```

```
        new_state = evolve(old_state)
        states_seq.append(new_state)
    return states_seq
```

note that we still haven't definide the functions **generate_state** and **evolve**

now we implement some stubs

what is the easiest way that we can use to make the code run?

In [23]:
```
def generate_state():
    return "stringa"

def evolve(stato):
    return stato
```

In [24]:
```
simulation(5)
```

Out[24]:
```
['stringa', 'stringa', 'stringa', 'stringa', 'stringa', 'stringa']
```

It might look trivial, but now we have a code that does something, and we can iteratively improve it, instead that trying to generate it perfectly all from nothing

This approach lets us divide our problem in sub-problems easier to solve, but requires a bit more abstract reasoning

Let's start from how we generate our state of the system.

which properties do we want?

As an example, we might requires that:

- the state is represented by a string
- only two states (characters) are possible, `'.'` ed `'0'`

In [30]:
```
%%file test_prova.py

def generate_state():
    return "stringa"

def evolve(stato):
    return stato

def simulation(nsteps):
    initial_state = generate_state()
    states_seq = [initial_state]
    for i in range(nsteps):
        old_state = states_seq[-1]
        new_state = evolve(old_state)
        states_seq.append(new_state)
    return states_seq

########################################################

def test_generation_valid_state():

```

```
    state = generate_state()
    assert set(state) == {'.', '0'}
```

```
Overwriting test_prova.py
```

In [31]:
```
!pytest test_prova.py
```

```
============================= test session starts ==============================
platform linux -- Python 3.6.7, pytest-4.3.0, py-1.8.0, pluggy-0.9.0
rootdir: /home/enrico/didattica/corso_programmazione_1819/programmingCourseDIFA, inifile:
plugins: xonsh-0.8.11
collected 1 item

test_prova.py F                                                          [100%]

=================================== FAILURES ===================================
_____________________________ test_generation_valid_state ______________________

    def test_generation_valid_state():
        state = generate_state()
>       assert set(state) == {'.', '0'}
E       AssertionError: assert {'a', 'g', 'i...'r', 's', ...} == {'.', '0'}
E         Extra items in the left set:
E         'i'
E         'g'
E         't'
E         'n'
E         'a'
E         's'...
E
E         ...Full output truncated (6 lines hidden), use '-vv' to show

test_prova.py:21: AssertionError
========================== 1 failed in 0.04 seconds ============================
```

In [32]:
```
%%file test_prova.py

def generate_state():
    return "....00......"

def evolve(stato):
    return stato

def simulation(nsteps):
    initial_state = generate_state()
    states_seq = [initial_state]
    for i in range(nsteps):
        old_state = states_seq[-1]
        new_state = evolve(old_state)
        states_seq.append(new_state)
    return states_seq

##########################################################
```

```
def test_generation_valid_state():
    state = generate_state()
    assert set(state) == {'.', '0'}
```

```
Overwriting test_prova.py
```

In [33]:
```
!pytest test_prova.py
```

```
============================= test session starts ==============================
platform linux -- Python 3.6.7, pytest-4.3.0, py-1.8.0, pluggy-0.9.0
rootdir: /home/enrico/didattica/corso_programmazione_1819/programmingCourseDIFA, inifile:
plugins: xonsh-0.8.11
collected 1 item

test_prova.py .                                                          [100%]

=========================== 1 passed in 0.02 seconds ===========================
```

our test is passing!

sure, our generated state is not very interesting, but at least is a valid one

TDD (**Test Driven Development**) help us avoiding over-engineering

Do not add functionality to the code before you need them!

Our next requirement might be that we have only a single `'0'`, to replicate the traditional pictures of the rule 30

In [34]:
```
%%file test_prova.py

def generate_state():
    return "....00......"

def evolve(stato):
    return stato

def simulation(nsteps):
    initial_state = generate_state()
    states_seq = [initial_state]
    for i in range(nsteps):
        old_state = states_seq[-1]
        new_state = evolve(old_state)
        states_seq.append(new_state)
    return states_seq

########################################################

def test_generation_valid_state():
    state = generate_state()
    assert set(state) == {'.', '0'}


def test_generation_single_alive():
    state = generate_state()
```

```
    num_of_0 = sum(1 for i in state if i=='0')
    assert num_of_0 == 1
```

```
Overwriting test_prova.py
```

In [35]:
```
!pytest test_prova.py
```

```
============================= test session starts ==============================
platform linux -- Python 3.6.7, pytest-4.3.0, py-1.8.0, pluggy-0.9.0
rootdir: /home/enrico/didattica/corso_programmazione_1819/programmingCourseDIFA, inifile:
plugins: xonsh-0.8.11
collected 2 items

test_prova.py .F                                                         [100%]

=================================== FAILURES ===================================
_____________________________ test_generation_single_alive _____________________

    def test_generation_single_alive():
        state = generate_state()
        num_of_0 = sum(1 for i in state if i=='0')
>       assert num_of_0 == 1
E       assert 2 == 1

test_prova.py:27: AssertionError
====================== 1 failed, 1 passed in 0.05 seconds ======================
```

In [36]:
```
%%file test_prova.py

def generate_state():
    return ".....0....."

def evolve(stato):
    return stato

def simulation(nsteps):
    initial_state = generate_state()
    states_seq = [initial_state]
    for i in range(nsteps):
        old_state = states_seq[-1]
        new_state = evolve(old_state)
        states_seq.append(new_state)
    return states_seq

############################################################

def test_generation_valid_state():
    state = generate_state()
    assert set(state) == {'.', '0'}


def test_generation_single_alive():
    state = generate_state()
```

```
    num_of_0 = sum(1 for i in state if i=='0')
    assert num_of_0 == 1
```

```
Overwriting test_prova.py
```

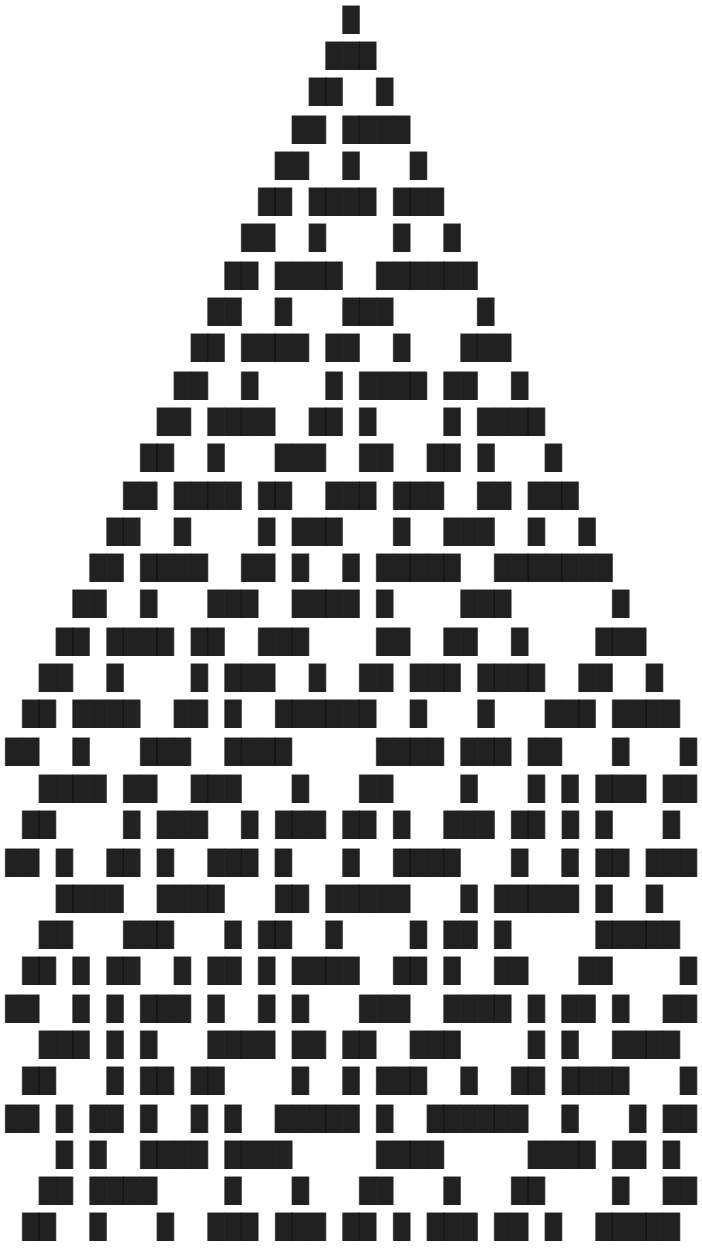
In [37]:
```
!pytest test_prova.py
```

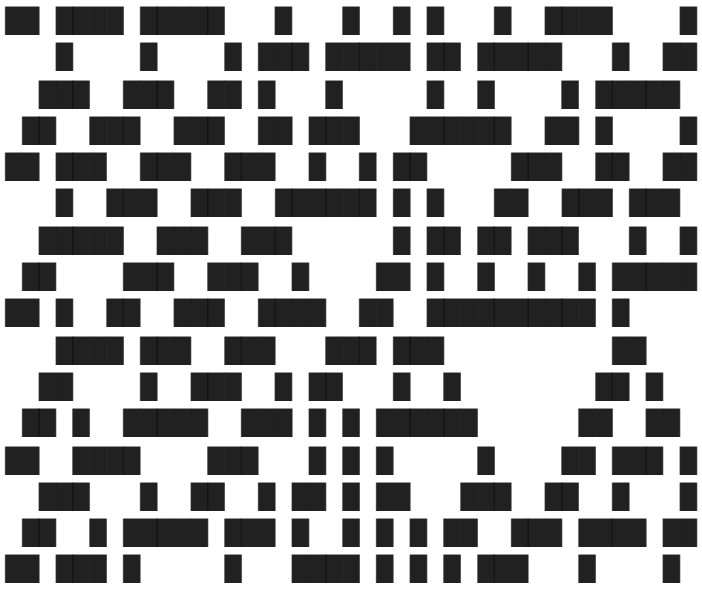
```
============================= test session starts ==============================
platform linux -- Python 3.6.7, pytest-4.3.0, py-1.8.0, pluggy-0.9.0
rootdir: /home/enrico/didattica/corso_programmazione_1819/programmingCourseDIFA, inifile:
plugins: xonsh-0.8.11
collected 2 items

test_prova.py ..
[100%]

=========================== 2 passed in 0.02 seconds ===========================
```

The final result





exercise time

try to implement the actual simulation by yourself!

now you will have to focus implementing the state evolution function:

- does it return a valid string?
- is it still of the same lenght as the one put as input

you will also have to make some choices: how do you manage the borders?

- circular border?
- reflective border?
- constant border?

Implementing other rules? 90? 110? 184?

In [39]:
```
rule30 = {"000": '.',
          "00.": '.',
          "0.0": '.',
          "...": '.',
          "0..": '0',
          ".00": '0',
          ".0.": '0',
          "..0": '0',
         }
```

In [40]:
```
RULES = {30: {"...": '0', "..0": '0', ".0.": '0', "000": '0',
              ".00": '.', "0..": '.', "0.0": '.', "00.": '.'},

         90: {"...": "0", "..0": ".", ".0.": "0", ".00": ".",
              "0..": ".", "0.0": "0", "00.": ".", "000": "0"},

         110: {"...": '0', "..0": '.', ".0.": '.', ".00": '0',
               "0..": '.', "0.0": '.', "00.": '.', "000": '0'},

         184: {"...": ".", "..0": "0", ".0.": ".", ".00": ".",
```

```
                "0..": ".", "0.0": "0", "00.": "0", "000": "0"}
        }
```

Writing good tests

until now we discussed the idea behind testing, but let's discuss some details to start mastering them.

At the beginning tests are difficult, because you're going to be more concerned with the idea of making your code work than with aking it right.

It's ok, you will learn with time, just try and you will get the hang of it

one test ~~one assert~~ one concept

you will find online a rule of thumb saying that one test should have only one assert.

it is almost right, but still wrong: one test should verify a single concept.

sometimes a single assert is enough, sometimes more than one are necessary.

but in general one should be able to explain simply **why** a test fails

In [ ]:
```
# WHY?!?
def test_my_state_generator():
    state = generate_state()
    assert set(state) == {'.', '0'}

    num_of_0 = sum(1 for i in state if i=='0')
    assert num_of_0 == 1
```

really, you don't earn money by pushing more concepts in a single tests, you're making it harder to used the tests.

once you see that a test fails, now you have to start debugging (we will talk more about that in the next lessons)... and you don't want that believe me.

In an ideal world, once a test fails, you can almost pinpoint the line of code that is the cause of it

documenting a test

test, as all the functions, should be documented. the test name is simply not enough to explain it, just keep it simple.

In this case we don't need a documentation in the style of numpy, but I suggest the BEHAVIOR DRIVEN APPROACH description.

Each test should state:

- what it wants to test

- **GIVEN**: what is the starting state of the system its testing
- **WHEN**: what happens, what operation is going to perform
- **THEN**: the expected effect of the operation after the operation

In [11]:
```
def is_valid_state(state):
    """this function tests that a state is valid

    is used mostly for testing.
    should have its own dedicated tests!
    """
    ...

def test_evolve_valid():
    """ this tests that the evolve function returns valid states when use

    GIVEN: a valid state of my simulation
    WHEN: I apply to it the evolve function
    THEN: the resulting state is still a valid one
    """
    state = generate_state() # the validity of this should be tested sepa
    new_state = evolve(state)
    assert is_valid_state(new_state)
```

tests and random numbers

random numbers do not play well with testing.

tests should be deterministic, and reliably pass or fail.

if you have a function that relies on random number (such as certain simulations) it's better to isolate the random and the deterministic code, and test the deterministic part very carefully.

one can usually isolate the random component almost totally

A different case is when you use the random number as a way of saying:

> I don't care about the specific value, and there are a lot of them and I can't be bothered of writing them all, let's use a random generator

it is legitimate, the simple solution is to fix the random seed in **every** function that use this trick

In [5]:
```
import random as rn

# passes but is conceptually wrong
def test_random_pick():
    values = rn.choose(["a", "b", "c"], k=3)
    assert len(values)==3
```

In [9]:
```
import random as rn

# passes and is completely deterministic
def test_random_pick():
    rn.seed(42)
```

```
    values = rn.choose(["a", "b", "c"], k=3)
    assert len(values)==3
```

second order functions

inside the body of the test ideally there should be only 3 lines of codes:

- defining the expected result
- executing the function one wants to test
- asserting that the results are equal

sometimes one might need more code to execute the test.

Part of this can be solved with fixtures, when the code to generate the expected result is repeated or a little more involved.

but in general, if there is non trivial code used is a test, it should:

- be encapsulated in a function
- tested itself

think back to the `is_valid_state` defined before

pytest specific - test coverage

using the `pytest-cov` package you can verify how many of the original code lines have been executed by your tests.

a coverage less than 100% (for the actual code, excluding things like plotting and GUI, at least now) means that the tests are incompleted.

A coverage of 100% is the minimal starting point, one still have to make sure that the tests are actually covering all the **logical** cases.

to use it just run you pytest program with:

```
pytest --cov=myproj tests/
```

pytest specific - parametrization

sometimes we have the same code for the test, just used on many different values to extensively test a single property.

The simple solution is just to have multiple tests, but pytest provide the option to **parametrize** the test, passing it several values to be tested using the same code.

I'm not a huge fan, but it's a possibility.

In [ ]:
```
import pytest

@pytest.mark.parametrize("a, b, remainder", [
    (3, 2, 1),
```

```
    (2, 3, 2),
    (2, 2, 0),
])
def test_divide_ok(a, b, expected):
    assert module(a, b) == expected
```

pytest specific - fixtures

a common problem in most testing framework is the **setup** and **teardown** of the test data.

If it's complicated it means a lot of repeated code, source of potential bugs.

Each framework solves this in its own way.

pytest approach are the **fixtures**.

they should the **yield** keyword, that we will see in the future in more detail.

for now we will see the simpler version with the **return**.

to create a fixture one just needs to create a function with the name of the object that it wants to create.

all the test methods can now take an argument with the same name as the fixture, and they will receive the value of that fixture as the argument

typical uses might involve reading the content of a file or setting up complex data structures.

In [ ]:
```
import pytest

@pytest.fixture
def valid_state():
    return generate_state()

def test_evolve_valid(valid_state):
    new_state = evolve(state)
    assert is_valid_state(new_state)
```

pytest specific - command line options

here are some of the pytest command line options that might be useful to shorten your testing cycle.

- showing the local variables of a failing test: `-l / --showlocals`
- exit the testing suite as soon as a test fails: `-x / --exitfirst`
- rerun only the tests that failed with the last run: `--lf / --last-failed`
- rerun all the tests, but start with those that failed first: `--ff / --failed-first`

Property based testing

Pytest automatize our testing procedure, but we still have to think and write a great number of tests, and most of them are going to be similar with small variations

The best solution would be for the computer to generate and keep track of tests for us

This is not possible in a literal sense, but we can get pretty close to it

I can generalize the anecdotal tests I wrote earlier, trying to check not for individual results, but general properties and simmetries of my code

In **unit testing**:

- for each test:
  - for each individual case
    1. I have to specify the input
    2. I have to specify the expected output (or error)

in **property based testing**:

- I need to specify the kind of output I will provide to the function
- for each test
  1. specificy the invariants of that test property

The library I am going to use will specify the input dat in a random way according to the rules I specified.

Will trow them at the function actively trying to break it

If it finds an example that breaks the expected properties of the functions, tries to find the simplest example that still breaks it

keeps track of that example and will provide it to all the future iterations of the test

property based testing does not replace unit testing.

It extends it and makes it more powerful, and reduces the amount of trivial and repeated code we have to write

Of course, to use it one needs to think harder about what they want to test, but you wouldn't be here if you were afraid of thinking

The library we are going to use is called hypothesis.

Hypothesis leverage libraries such as pytest for the basic testing, but generates test automatically using **strategies**.

A **strategy** defines how the data are randomly generated and passed to the function to be tested

In [20]:
```
%%file test_prova.py
from hypothesis import given
import hypothesis.strategies as st

def inc(x):
    if x==5:
        return 0
    return x + 1

def dec(x):
    return x - 1

@given(value=st.integers())
def test_answer_1(value):
    print(value)
    assert dec(inc(value)) == value

@given(value=st.integers())
def test_answer_2(value):
    assert dec(inc(value)) == inc(dec(value))
```

```
Overwriting test_prova.py
```

In [21]:
```
!pytest test_prova.py
```

```
============================= test session starts ==============================
platform linux -- Python 3.6.0, pytest-3.0.6, py-1.4.32, pluggy-0.4.0
rootdir: /home/enrico, inifile:
plugins: xonsh-0.5.6, hypothesis-3.6.1
collected 2 items

test_prova.py .F

=================================== FAILURES ===================================
_____________________________ test_answer_2 ______________________________

    @given(value=st.integers())
>   def test_answer_2(value):

test_prova.py:18:
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
/usr/local/lib/python3.6/site-packages/hypothesis/core.py:524: in wrapped_
test
    print_example=True, is_final=True
/usr/local/lib/python3.6/site-packages/hypothesis/executors.py:58: in defa
ult_new_style_executor
    return function(data)
/usr/local/lib/python3.6/site-packages/hypothesis/core.py:111: in run
    return test(*args, **kwargs)
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
value = 5

    @given(value=st.integers())
    def test_answer_2(value):
>       assert dec(inc(value)) == inc(dec(value))
E       assert -1 == 5
E        +  where -1 = dec(0)
E        +    where 0 = inc(5)
E        +  and   5 = inc(4)
E        +    where 4 = dec(5)

test_prova.py:19: AssertionError
---------------------------------- Hypothesis ----------------------------------
Falsifying example: test_answer_2(value=5)
====================== 1 failed, 1 passed in 0.15 seconds ======================
```

To write property based testing one doesn't need to start from scratch, but can progressively extends the classic unit tests.

for a starter, jut replace the fixed parameters with the **just** strategy.

In [44]:
```
%%file test_prova.py
def inc(x):
    return x + 1

def test_answer_1a():
```

```
    assert inc(3) == 4


from hypothesis import given
import hypothesis.strategies as st

@given(x=st.just(3))
def test_answer_1b(x):
    assert inc(x) == x+1

@given(x=st.floats())
def test_answer_1c(x):
    assert inc(x) == x+1
```

```
Overwriting test_prova.py
```

In [45]:
```
!pytest test_prova.py
```

```
============================= test session starts ==============================
platform linux -- Python 3.6.0, pytest-3.0.6, py-1.4.32, pluggy-0.4.0
rootdir: /home/enrico, inifile:
plugins: xonsh-0.5.6, hypothesis-3.6.1
collected 3 items

test_prova.py ..F

=================================== FAILURES ===================================
_____________________________________ test_answer_1c ______________________________

    @given(x=st.floats())
>   def test_answer_1c(x):

test_prova.py:16:
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
/usr/local/lib/python3.6/site-packages/hypothesis/core.py:524: in wrapped_
test
    print_example=True, is_final=True
/usr/local/lib/python3.6/site-packages/hypothesis/executors.py:58: in defa
ult_new_style_executor
    return function(data)
/usr/local/lib/python3.6/site-packages/hypothesis/core.py:111: in run
    return test(*args, **kwargs)
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _

x = nan

    @given(x=st.floats())
    def test_answer_1c(x):
>       assert inc(x) == x+1
E       assert nan == (nan + 1)
E        +  where nan = inc(nan)

test_prova.py:17: AssertionError
---------------------------------- Hypothesis ----------------------------------
Falsifying example: test_answer_1c(x=nan)
====================== 1 failed, 2 passed in 0.62 seconds ======================
```

As we were saying, math on a computer is hard...

If we want to ignore some of these cases, we could use the **assume** function, that restricts the random function generation for the testing of each individual function

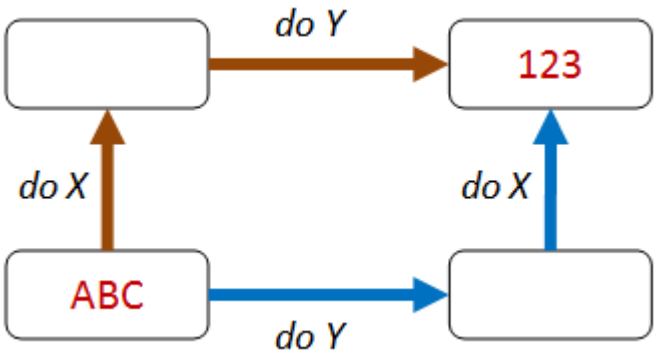
In [ ]:
```python
from math import isnan
from hypothesis import assume

@given(x=st.floats())
def test_answer_1c(x):
    assume(not isnan(x))
    assert inc(x) == x+1
```

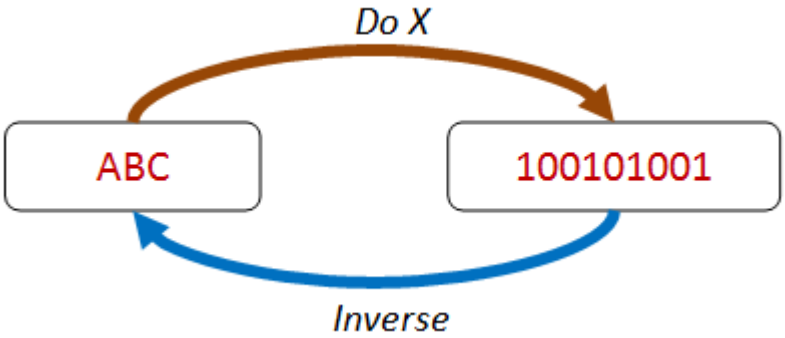
(Some) Patterns of properties to be tested

The following examples are taken from this website

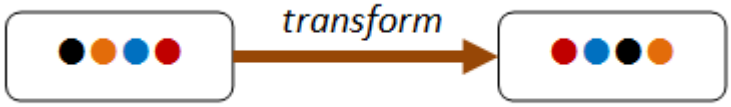
Commutative property



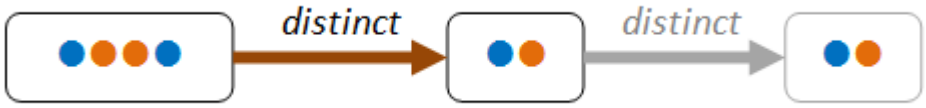
Existence of an inverse function



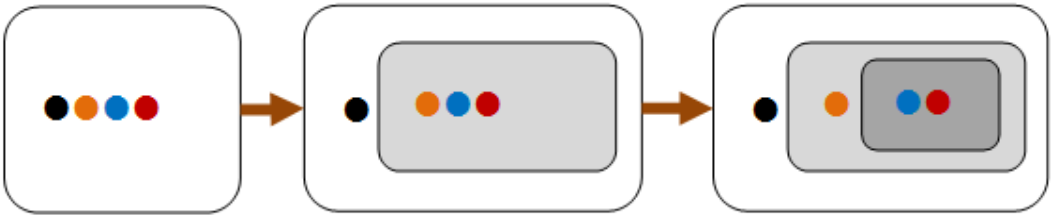
Conservation law



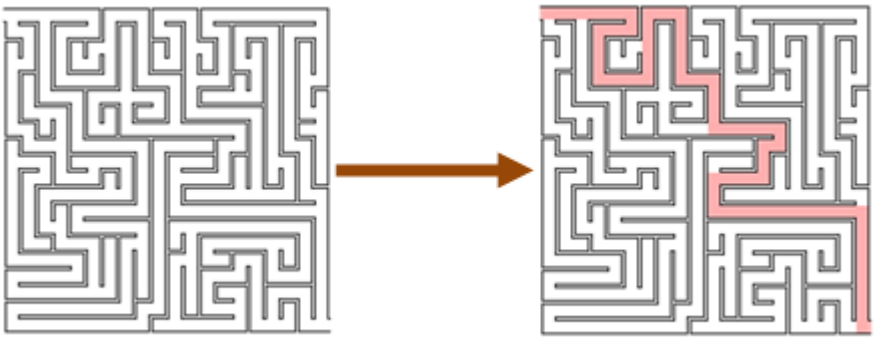
idempotence



Induction



Hard to proof, easy to verify



Oracle testing