Scientific Programming Practical 13

Introduction

Luca Bianco - Academic Year 2018-19
luca.bianco@fmach.it

Testing

Testing the code is quite an important step to make sure that the code is **predictable** and although some bugs can always slip through, **testing is the process of making the code as predictable as possible**.

Testing: white box

Testing the code is quite an important step to make sure that the code is **predictable** and although some bugs can always slip through, **testing is the process of making the code as predictable as possible**.

Program as white box

Check the code throughly getting into the details of how things are calculated

Testing: black box

Testing the code is quite an important step to make sure that the code is predictable and although some bugs can always slip through, **testing is the process of making the code as predictable as possible**.

Program as black box

testing does not look at the details of how the code is implemented, but it just focus on the correctness of the output produced by the code.

Testing: test

Program as white box

Program as black box

A **test** is a piece of code written with the sole purpose of checking the correctness of another piece of code.

Testing: test

A **test** is a piece of code written with the sole purpose of checking the correctness of another piece of code.

Program as white box

Program as black box

Testing requires three successive moments:

1. **setup** the test connecting methods to test data;

2. **execute** the code under test using the connections setup at the previous step

3. **verification** of the results to make sure they look as they are expected to

Doctest

A very simple way to specifying tests for the code is by using an embedded module called **doctest**. It will basically search for pieces of code in your python file that look like **interactive python sessions** (that are lines starting with `>>>`) and will execute them to check if they run giving the result specified in the next line.

```
import doctest

def func(data):
     """
     This is a function that
     returns three values in a
     list...
     >>> fun(mylist)
     [x, y, z]
     """
...

doctest.testmod()
```

tests if fun(mylist) returns [x,y,z]

Doctest

Example: Let's define some doctest tests for the simple function computing the first N prime numbers.

```
def getFirstNprimes(N):
    '''
    This function should output the first N prime numbers.
    >>> getFirstNprimes(1)
    [2]
    >>> getFirstNprimes(2)
    [2, 3]
    >>> getFirstNprimes(10)
    [2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
    '''
    if N == 0:
        return []
    res = [2]
    current = 3
    while len(res) < N:
        if len([x for x in res if current % x == 0]) == 0:
            res.append(current)
        current += 1
    #uncomment next line to introduce a bug
    #res.append(1)
    return res

if __name__ == "__main__":
    import doctest
    doctest.testmod()
    print(getFirstNprimes(20))
```

space

The line `if __name__ == "__main__":` is used to specify if the code is executed as a script (i.e. it is not invoked as an imported module somewhere else in another piece of code).

Code as it is, passes tests!

Doctest

Example: Let's define some doctest tests for the simple function computing the first N prime numbers.

```python
def getFirstNprimes(N):
    '''
    This function should output the first N prime numbers.
    >>> getFirstNprimes(1)
    [2]
    >>> getFirstNprimes(2)
    [2, 3]
    >>> getFirstNprimes(10)
    [2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
    '''
    if N == 0:
        return []
    res = [2]
    current = 3
    while len(res) < N:
        if len([x for x in res if current % x == 0]) == 0:
            res.append(current)
        current += 1
    #uncomment next line to introduce a bug
    res.append(1)
    return res

if __name__ == "__main__":
    import doctest
    doctest.testmod()
    print(getFirstNprimes(20))
```

```
**********************************************************************
File "__main__", line 6, in __main__.getFirstNprimes
Failed example:
    getFirstNprimes(1)
Expected:
    [2]
Got:
    [2, 1]
**********************************************************************
File "__main__", line 8, in __main__.getFirstNprimes
Failed example:
    getFirstNprimes(2)
Expected:
    [2, 3]
Got:
    [2, 3, 1]
**********************************************************************
File "__main__", line 10, in __main__.getFirstNprimes
Failed example:
    getFirstNprimes(10)
Expected:
    [2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
Got:
    [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 1]
**********************************************************************
1 items had failures:
   3 of   3 in __main__.getFirstNprimes
***Test Failed*** 3 failures.
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 1]
```

Raising/catching exceptions

Exceptions are a very good way to **inform the program that some unexpected thing has happened** (e.g. division by zero, tentative to access a position in a list that does not exist, summing a string and an integer...).

```
class MyIntPair:
    def __init__(self, x,y):
        if not type(x) == int:
            raise Exception("x {} is not integer".format(x))
        if not type(y) == int:
            raise Exception("y {} is not integer".format(y))
        self.x = x
        self.y = y

    def __add__(self, other):
        return (self.x + other.x, self.y + other.y)

A = MyIntPair(5,10)
B = MyIntPair(3,6)
print(A + B)
C = MyIntPair(1, "two")
```

Raising/catching exceptions

Exceptions are a very good way to **inform the program that some unexpected thing has happened** (e.g. division by zero, tentative to access a position in a list that does not exist, summing a string and an integer...).

```
class MyIntPair:
    def __init__(self, x,y):
        if not type(x) == int:
            raise Exception("x {} is not integer".format(x))
        if not type(y) == int:
            raise Exception("y {} is not integer".format(y))
        self.x = x
        self.y = y

    def __add__(self, other):
        return (self.x + other.x, self.y + other.y)

A = MyIntPair(5,10)
B = MyIntPair(3,6)
print(A + B)
C = MyIntPair(1, "two")
```

```
(8, 16)

---------------------------------------------------------------------------
Exception                                 Traceback (most recent call last)
<ipython-input-27-c7df16e96a19> in <module>()
     14 B = MyIntPair(3,6)
     15 print(A + B)
---> 16 C = MyIntPair(1, "two")

<ipython-input-27-c7df16e96a19> in __init__(self, x, y)
      4             raise Exception("x {} is not integer".format(x))
      5         if not type(y) == int:
----> 6             raise Exception("y {} is not integer".format(y))
      7         self.x = x
      8         self.y = y

Exception: y two is not integer
```

Raising/catching exceptions

Try - except:

try to perform some operations, but be ready as some exceptions might occur.
If that is the case the **except portion is executed** and the exception can be dealt with

```
class MyIntPair:
    def __init__(self, x,y):
        if not type(x) == int:
            raise Exception("x: {} is not integer".format(x))
        if not type(y) == int:
            raise Exception("y: {} is not integer".format(y))
        self.x = x
        self.y = y

    def __add__(self, other):
        return (self.x + other.x, self.y + other.y)

try:
    A = MyIntPair(5,10)
    B = MyIntPair(3,6)
    #Uncomment to see a different error
    #print(A/0)
    print(A + B)
    C = MyIntPair(1, "two")
    print(A + C)

except Exception as e:
    print("Whoops something went wrong. Ignore the rest.")
    print(str(e))
```

```
(8, 16)
Whoops something went wrong. Ignore the rest.
y: two is not integer
```

Unittests

Unittests are another way to perform testing of the code.

The module `unittest` must be imported first with `import unittest` and then the Test class must be implemented to perform the tests.

To create some unit tests:

1. define a `Testing` class (we are free to call it as we like) which is a subclass of the class `unittest.TestCase`.
2. Specify the tests we want to run. Every test is a method and its name **must start** with `test_` (e.g. `test_length`).

Tests can use assertions:

`assertEqual(value1, value2)`, `assertTrue(condition)` or `assertFalse(condition)`

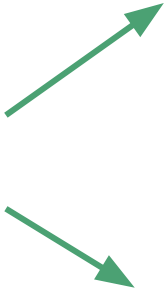
that allow to check the equality of two values (i.e. the known result and the output of the method to be tested) and the truth value of a condition (typically computed on the output of the method under test).

Run the tests with: `unittest.main()`

or: `python3 -m unittest my_testing_function.py`

Unittests

```
python3 -m unittest
my_testing_function.py
```



```
python3 -m unittest file_samples/my_testing_function.py

.....
----------------------------------------------------------------------
Ran 5 tests in 0.387s

OK
```

```
======================================================================
FAIL: test_ten (file_samples.my_testing_function.Testing)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/home/biancol/Google Drive/work/courses/QCBsciprolab/file_samples/my_testing_function.py", line 37,
 in test_ten
    [2, 3, 5, 7, 11, 13, 17, 19, 23, 29])
AssertionError: Lists differ: [2, 3, 5, 7, 11, 13, 17, 10, 23, 29] != [2, 3, 5, 7, 11, 13, 17, 19, 23, 29]

First differing element 7:
10
19

- [2, 3, 5, 7, 11, 13, 17, 10, 23, 29]
?                           ^

+ [2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
?                           ^


----------------------------------------------------------------------
Ran 5 tests in 0.770s

FAILED (failures=2)
```

Example: Let's define some doctest tests for the simple function computing the first N prime numbers.

```
import unittest
import random

def getFirstNprimes(N):
    '''
    This function should output the first N prime numbers.
     '''
    if N <= 0:
        return []
    res = [2]
    current = 3
    while len(res) < N:
        if len([x for x in res if current % x == 0]) == 0:
            res.append(current)
        current += 1
    #uncomment next line to introduce a bug
    #res.append(1)
    #or a more subtle error:
    #ind = random.randint(len(res))
    #res[ind] = 10
    return res
```

```
class Testing(unittest.TestCase):
    def test_empty(self):
        self.assertEqual(getFirstNprimes(0),[])

    def test_one(self):
        self.assertEqual(getFirstNprimes(1),[2])

    def test_ten(self):
        self.assertEqual(getFirstNprimes(10),
                         [2, 3, 5, 7, 11, 13, 17, 19, 23, 29])

    def test_len(self):
        for i in range(0,10):
            n = random.randint(1,1000)
            self.assertFalse(len(getFirstNprimes(n)) != n)


    def test_negative(self):
        self.assertTrue(len(getFirstNprimes(-1)) == 0)

if __name__ == "__main__":
    #uncomment to run the tests in the main (without -m)
    #unittest.main()
    print(getFirstNprimes(20))
```

Example: Let's define some doctest tests for the simple function computing the first N prime numbers.

```python
import unittest
import random

def getFirstNprimes(N):
    '''
    This function should output the first N prime numbers.
     '''
    if N <= 0:
        return []
    res = [2]
    current = 3
    while len(res) < N:
        if len([x for x in res if current % x == 0]) == 0:
            res.append(current)
        current += 1
    #uncomment next line to introduce a bug
    #res.append(1)
    #or a more subtle error:
    #ind = random.randint(len(res))
    #res[ind] = 10
    return res
```

extends unittest.Testcase

```python
class Testing(unittest.TestCase):
    def test_empty(self):
        self.assertEqual(getFirstNprimes(0),[])

    def test_one(self):
        self.assertEqual(getFirstNprimes(1),[2])

    def test_ten(self):
        self.assertEqual(getFirstNprimes(10),
                         [2, 3, 5, 7, 11, 13, 17, 19, 23, 29])

    def test_len(self):
        for i in range(0,10):
            n = random.randint(1,1000)
            self.assertFalse(len(getFirstNprimes(n)) != n)


    def test_negative(self):
        self.assertTrue(len(getFirstNprimes(-1)) == 0)

if __name__ == "__main__":
    #uncomment to run the tests in the main (without -m)
    #unittest.main()
    print(getFirstNprimes(20))
```

Beware: random errors are quite difficult to spot/corrct

Regular expressions

A **regular expression** (**regex**) is a string of characters defining a **search pattern** with which we can carry out operations such as pattern and string matching, find/replace etc.

There are two types of characters: **normal characters** (which have to match amongst themselves) and **special characters** (which are used to specify repetitions (`*, ?, +, {x,y}`), a set of elements (`[]`), negation (`[^]`), beginning (`^`) of a string, end of a string (`$`), etc.

Character	Meaning
`text`	Matches itself
`(regex)`	Matches the regex regex (i.e. parentheses don't count)
`^`	Matches the start of the string
`$`	Matches the end of the string or just before the newline
`.`	Matches any character except a newline
`regex?`	Matches 0 or 1 repetitions of regex (longest possible)
`regex*`	Matches 0 or more repetitions of regex (longest possible)
`regex+`	Matches 1 or more repetitions of regex (longest possible)
`regex{m,n}`	Matches from m to n repetitions of regex (longest possible)
`[...]`	Matches a set of characters
`[c1-c2]`	Matches the characters "in between" `c1` and `c2`
`[^...]`	Matches the complement of a set of characters
`r1|r2`	Matches both `r1` and `r2`

Regular expressions

What does the following regex match?

```
regex = "[A-Z]__[0-9]{1,4}__[a-z:-]*__[A-Z]"
```

Regular expressions

What does the following regex match (assuming only IUPAC ambiguous nucleotide alphabet)?

```
regex = "[ATCG]*([^ATCG]+[ATCG]*)*"
```

Match, search, finditer and findall

```
import re
```

`re.match(regex, str)` where `regex` is a string (the regular expression), `str` is the input string. Tries to match the regex on the string starting **from the beginning** (i.e. left-to-right). Returns an `MatchObject` or None if no match;

`re.search(regex,str)` searches `regex` in the whole string and returns a `MatchObject` with **the first occurrence** of the pattern or None if the regex could not match anything;

`re.finditer(regex,str)`: returns an iterator to `MatchObject` instances over all **non-overlapping** matches for the regular expression pattern in string. The string is scanned left-to-right, and matches are returned in the order found;

Given a `MatchObject`, if not None, provides the following information:

- `MatchObject.group()` : the matched string;
- `MatchObject.start()` : the starting point of the matched string in the tested string (str);
- `MatchObject.end()` : the ending point of the matched string in the tested string (str);
- `MatchObject.groups()` : when defining a regular expression, we can define subgroups by using “()”. This method returns a tuple containing all the subgroups.

```python
import re


myStr = "hi there, i am using Python and learning hOw to UsE Regular Expressions AKA REGEX"

m = re.search("123", myStr)
print(myStr)
if(m):
    print("123 is in myStr")
else:
    print("123 is NOT in myStr")
    print("m is {}".format(m))

a = myStr.split()
print("\nCapitalized words:")
for word in a:
    match = re.match("[A-Z]+[a-zA-Z]*",word)
    if(match):
        print(word)

print("\nOn the whole string with SEARCH:")
result = re.search(" [A-Z]+[a-zA-Z]*|^[A-Z]+[a-zA-Z]*| [A-Z]+[a-zA-Z]*$", myStr)

print("{}: starts:{} ends: {} that is: \"{}\"".format(result.group(),
                                    result.start(),
                                    result.end(),
                                    myStr[result.start():result.end()]))

print("\nIteratively with finditer")
#get all the words starting with a capital letter
for m in re.finditer(" [A-Z]+[a-zA-Z]*|^[A-Z]+[a-zA-Z]*| [A-Z]+[a-zA-Z]*$", myStr):
    print(type(m))
    print("{}: starts:{} ends: {} that is: \"{}\"".format(m.group(),
                                    m.start(),
                                    m.end(),
                                    myStr[m.start():m.end()]))
```

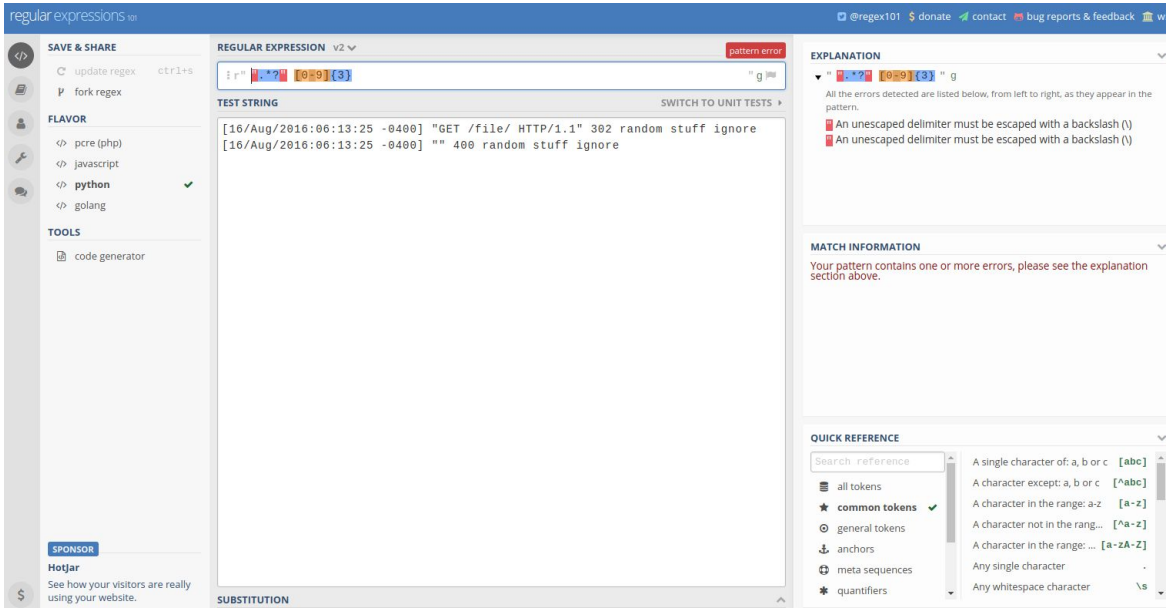
Output

```
hi there, i am using Python and learning hOw to UsE Regular Expressions AKA REGEX
123 is NOT in myStr
m is None

Capitalized words:
Python
UsE
Regular
Expressions
AKA
REGEX

On the whole string with SEARCH:
 Python: starts:20 ends: 27 that is: " Python"

Iteratively with finditer
<class '_sre.SRE_Match'>
 Python: starts:20 ends: 27 that is: " Python"
<class '_sre.SRE_Match'>
 UsE: starts:47 ends: 51 that is: " UsE"
<class '_sre.SRE_Match'>
 Regular: starts:51 ends: 59 that is: " Regular"
<class '_sre.SRE_Match'>
 Expressions: starts:59 ends: 71 that is: " Expressions"
<class '_sre.SRE_Match'>
 AKA: starts:71 ends: 75 that is: " AKA"
<class '_sre.SRE_Match'>
 REGEX: starts:75 ends: 81 that is: " REGEX"
```

Regular expressions

If you want to test them:

https://regex101.com/



http://qcbprolab.readthedocs.io/en/latest/practical13.html

Exercises

1. The following function is supposed to get two lists of integers (let's call then X and Y) and return the list of elements that are contained in both (let's call it B). Is it correct? Devise a unit test to check if it is correct or not. In the latter case propose a correct version of the function.

```
def myListIntersection(X,Y):
    tmp = X + Y
    vals = [x for x in tmp if tmp.count(x) == 2]
    return list(set(vals))

A = [1, 2, 3, 4, 7, 12]
B = [4, 1, 7, 120]
C = [120, 6]
D = []

print("A, B: {}".format(myListIntersection(A,B)))
print("A, C: {}".format(myListIntersection(A,C)))
print("B, C: {}".format(myListIntersection(B,C)))
print("A, D: {}".format(myListIntersection(A,D)))
```

Show/Hide Solution

2. CRISPR-Cas9 is quite a neat system to perform genome editing. Guide RNAs (gRNAs) can transport Cas9 to anywhere in the genome for gene editing, but no editing can occur at any site other than one at which Cas9 recognizes the protospacer adjacent motif (PAM). The PAM site is a 2-6 base pair DNA sequence immediately following the DNA sequence targeted by the Cas9 nuclease in the CRISPR bacterial adaptive immune system. Some used PAMs are the following:

```
NGG (where N is any base)
NGA
YG (where Y is a Pyrimidine, i.e. C or T)
TTTN
YTN
```

write a function that loads the fasta sequences contig82.fasta and for each sequence reports