Architektura systemów Bioinformatycznych

Automatyczne testowanie i dokumentacja

Bartek Wilczyński

11.III 2020

Czemu testować oprogramowanie

- Programy zawierają błędy
- Nie da się rozwiązać problemu „czy program jest poprawny” z powodów fundamentalnych
- W projektach bioinformatycznych nie ma zwykle nakładów na weryfikację programów (inną niż użycie) ani na testerów
- Skutki mogą czasem być dość katastrofalne

Error! – What Biomedical Computing Can Learn From Its Mistakes | Biomedical Computation Review - Chromium
Error! – What Biom ×
Bartek
biomedicalcomputationreview.org/content/error—what-biomedical-computing-can-learn-its-...

September 1st/2011

Error! – What Biomedical Computing Can Learn From Its Mistakes

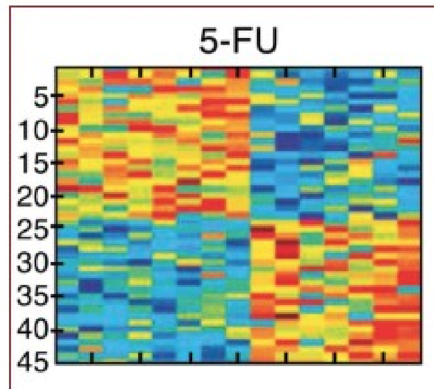
How errors in data, software, and methodology can teach us how to do better

By Kristin Sainani, PhD

pdf print leave a comment

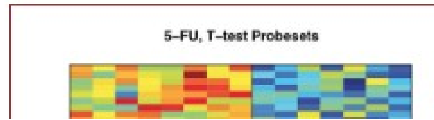
In 2006, *a paper in Nature Medicine* suggested a novel and potentially revolutionary method for predicting patient responses to cancer therapies using gene signatures. The finding piqued the interest of oncologists at The University of Texas MD Anderson Cancer Center, who sought help from two statisticians, ***Keith A. Baggerly, PhD***, and ***Kevin Coombes, PhD***, to recreate the approach.

Baggerly and Coombes, both professors of bioinformatics and computational biology, unexpectedly uncovered multiple errors with the data: off-by-one indexing errors, label reversals, inconsistencies, and duplications. The consequence: the original results were not reproducible and the approach was ultimately discredited.



The story received unusual public attention (more details follow). But it is by no means an isolated case. Errors in biomedical computing are surprisingly common. Strictly speaking, every biomedical model contains error in the sense that it is an imperfect representation of the truth. But more troubling are the errors that are avoidable—such as misadventures in Excel, glitches in the software, bad assumptions, and typos. As datasets and models become increasingly complex, errors of this type become both harder to avoid and harder to detect.

"When you've got a complicated model with a bunch of stuff in it, it's hard to tell when it's wrong," says ***James Bassingthwaighte, MD, PhD***, professor of bioengineering at the University of Washington. The point of a complex model is to predict behavior beyond the limits of intuition; but, in this realm, our intuition for spotting errors also becomes unreliable.



Plus, the current publication system wasn't designed to catch errors buried within high-dimensional data or intricate models. Reviewers and editors rarely have direct access to datasets or code; and when they do, they don't have time to check every step of the authors' analyses.

SHARE THIS
Tweet Like 105 Email

POPULAR ARTICLES

The Epigenome: A New View Into the Book of Life
There is growing recognition that epigenetics may be just as important as genetics in human health and disease.
06/01/10 by Kristin Sainani, PhD

Big Data Analytics In Biomedical Research
Can the complexities of biology be boiled down to Amazon.com-style recommendations? The examples here suggest possible pathways to an intelligent healthcare system with big data at its core.
01/02/12 by Katharine Miller

Integrating the Fragmented Mind: Bringing the Whole Elephant into View
Scientists are bringing diverse methods together to better understand schizophrenia and other mental illnesses
06/03/15 by Katharine Miller

Rodzaje testów programowania (niezbyt ścisła klasyfikacja)

- Source code checkers (static testing)
- Execution tests:
  - Unit tests
  - Integration tests
  - Regression tests
  - UX tests (GUI, Web, mobile)
  - Fuzz tests (random data, shouldn’t crash)
  - Mock tests – podmienianie modułów na „zaślepki”

PythonTestingToolsTaxonomy - Python Wiki - Chromium

PythonTestingTools ×

Secure | https://wiki.python.org/moin/PythonTestingToolsTaxonomy

Search | titles | text

» PythonTestingToolsTaxonomy

» PythonTesti...olsTaxonomy

- FrontPage
- RecentChanges
- FindPage
- HelpContents
- PythonTesti...olsTaxonomy

Page

- Immutable Page
- Info
- Attachments
- More Actions:

User

- Login

Contents

1. Unit Testing Tools
2. Mock Testing Tools
3. Fuzz Testing Tools
4. Web Testing Tools
5. Acceptance/Business Logic Testing Tools
6. GUI Testing Tools
7. Source Code Checking Tools
8. Code Coverage Tools
9. Continuous Integration Tools
10. Automatic Test Runners
11. Test Fixtures
12. Miscellaneous Python Testing Tools

Join the Testing In Python (TIP) mailing list for Python testing tools discussions!

This wiki page is originated from PyCheeseCake and it was originally created by Grig Gheorghiu

Unit Testing Tools

Tool	Category	Author	Part of	Claim to fame

Python unittest

- Tradycyjny moduł do testowania modułów
- Idea oparta na pomysłach z modułu JUnit w Javie
- Bardzo dobrze udokumentowany
- Dość dużo pisania kodu na potrzeby testów
- Wymaga dość dużej dyscypliny w kodowaniu oraz projektowania samych testów
- Wykorzystywany w dużych projektach, np.. Biopython

Jak działają unit testy

- Rozpatrujemy podjednostki (ang. Units) naszego projektu i projektujemy testy dla każdej z nich
- Testy obejmują zwykle uruchomienia funkcji i modułów naszego programu wraz z
  - przykładowymi danymi,
  - spodziewanymi wynikami,
  - Asercjami na wartości pewnych zmiennych
- Biblioteka unittest pozwala na automatyczne lub ręczne uruchamianie całych zestawów testów

Doctests

- Przykłady do testów umieszczane w komentarzach do funkcji
- Bardzo „pythonowy”, w tym sensie, że wykorzystuje dobre praktyki: jeśli w komentarzu do funkcji jest przykładowe użycie, to wykonanie powinno być zgodne z oczekiwaniem
- Wyniki oparte na zachowaniu interpretera
- Prosty, choć niestety trochę zależny od wersji interpretera i środowiska
- W biopythonie np. doctesty są oparte także na tutorialu

```python
"""
This is the "example" module.

The example module supplies one function, factorial().  For example,

>>> factorial(5)
120
"""

def factorial(n):
    """Return the factorial of n, an exact integer >= 0.

    >>> [factorial(n) for n in range(6)]
    [1, 1, 2, 6, 24, 120]
    >>> factorial(30)
    265252859812191058636308480000000
    >>> factorial(-1)
    Traceback (most recent call last):
        ...
    ValueError: n must be >= 0

    Factorials of floats are OK, but the float must be an exact integer:
    >>> factorial(30.1)
    Traceback (most recent call last):
        ...
    ValueError: n must be exact integer
    >>> factorial(30.0)
    265252859812191058636308480000000

    It must also not be ridiculously large:
    >>> factorial(1e100)
    Traceback (most recent call last):
        ...
    OverflowError: n too large
    """

    import math
    if not n >= 0:
        raise ValueError("n must be >= 0")
    if math.floor(n) != n:
        raise ValueError("n must be exact integer")
    if n+1 == n:  # catch a value like 1e300
        raise OverflowError("n too large")
    result = 1
    factor = 2
    while factor <= n:
        result *= factor
        factor += 1
    return result


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

Testowanie pokrycia kodu

- Dzięki narzędziom takim jak coverage.py, możemy zmierzyć pokrycie kodu podczas wykonania dowolnego kodu
- W szczególności można to zrobić podczas uruchomienia testów
- Założenie jest takie, że jeśli podczas testów jakaś linia kodu nie została wykonana to nie została przetestowana. Oczywiście pokrycie 100% linii kodu nie oznacza przetestowania całej funkcjonalności

Automatyzacja testów: TOX

Welcome to the tox automation project — tox 3.7.1.dev23 documentation - Chromium

Welcome to the tox auto ×

https://tox.readthedocs.io/en/latest/

Fork me on GitHub

tox

standardise testing in Python

Star 944

build passing

Search

quicklinks

home examples
install changelog
config issues[gh]
support plugins/hooks

Table of Contents

Welcome to the tox automation project

- vision: standardize testing in Python
- What is tox?
- Basic example
- System overview
- Current features
- Related projects


```
# Hiring 4 Python?
while is_open(job):
  try:
    # Hire easier!
    promote(RTD)
  finally:
```

Welcome to the tox automation project

vision: standardize testing in Python

`tox` aims to automate and standardize testing in Python. It is part of a larger vision of easing the packaging, testing and release process of Python software.

What is tox?

tox is a generic virtualenv management and test command line tool you can use for:

- checking your package installs correctly with different Python versions and interpreters
- running your tests in each of the environments, configuring your test tool of choice
- acting as a frontend to Continuous Integration servers, greatly reducing boilerplate and merging CI and shell-based testing.

Basic example

First, install `tox` with `pip install tox`. Then put basic information about your project and the test environments you want your project to run in into a `tox.ini` file residing right next to your `setup.py` file:

```
# content of: tox.ini , put in same dir as setup.py
[tox]
envlist = py27,py36

[testenv]
# install pytest in the virtualenv where commands will be executed
deps = pytest
commands =
    # NOTE: you can run any command line tool here - not just tests
    pytest
```

v: latest

You can also try generating a `tox.ini` file automatically, by running `tox-quickstart` and then answering a few simple questions.

Automatyzacja integracji

- Travis – automatyczne budowanie na linuksie, umożliwia konfigurację środowisk
- Appveyor – automatyczne budowanie na windows, pozwala testować różne wersje windows, różne wersje pythona
- Codecov – testowanie pokrycia kodu
- Wszystko to komercyjne projekty zintegrowane z github, darmowy dla projektów tj. biopython

Testowanie interfejsu, np. selenium

- Symulujemy działanie użytkownika i przeglądarki
- system „klika” w różne elementy, na podstawie „test cases”
- Stosunkowo duża rola układania test-case’ów
- Dużo zmiennych środowiska
- Trudne do zmierzenia (Kiedy strona jest „za wolna”?)

Epydoc

- Oparty na javadoc'u system dokumentowania API
- Generuje strony www i dokumenty pdf
- Wykorzystuje stosowany w pythonie schemat dokumentowania w komentarzach
- Używany w biopythonie
- W zasadzie można go użyć do dowolnego projektu w pythonie

Table of Contents

Everything

Modules

Bio
Bio.Affy
Bio.Affy.CelFile
Bio.Align
Bio.Align.AlignInfo
Bio.Align.Applications
Bio.Align.Applications._ClustalOı
Bio.Align.Applications._Clustalw

Everything

All Classes

Bio.Affy.CelFile.ParserError
Bio.Affy.CelFile.Record
Bio.Align.AlignInfo.PSSM
Bio.Align.AlignInfo.SummaryInfo
Bio.Align.Applications._ClustalOı
Bio.Align.Applications._Clustalw.
Bio.Align.Applications._Dialign.D
Bio.Align.Applications._MSAProl
Bio.Align.Applications._Mafft.Ma
Bio.Align.Applications._Muscle.M
Bio.Align.Applications._Prank.Pra
Bio.Align.Applications._Probcons
Bio.Align.Applications._TCoffee.
Bio.Align.MultipleSeqAlignment
Bio.AlignIO.ClustalIO.ClustalItera
Bio.AlignIO.ClustalIO.ClustalWri
Bio.AlignIO.EmbossIO.EmbossIte
Bio.AlignIO.EmbossIO.EmbossW
Bio.AlignIO.Interfaces.Alignment
Bio.AlignIO.Interfaces.Alignment
Bio.AlignIO.Interfaces.Sequential.
Bio.AlignIO.MafIO.MafIndex
Bio.AlignIO.MafIO.MafWriter
Bio.AlignIO.MauveIO.MauveItera
Bio.AlignIO.MauveIO.MauveWrit
Bio.AlignIO.NexusIO.NexusWrite
Bio.AlignIO.PhylipIO.PhylipIterat

Module Hierarchy

- **Bio**: *Collection of modules for dealing with biological data in Python.*
  - **Bio.Affy**: *Deal with Affymetrix related data such as cel files.*
    - **Bio.Affy.CelFile**: *Reading information from Affymetrix CEL files version 3 and 4.*
  - **Bio.Align**: *Code for dealing with sequence alignments.*
    - **Bio.Align.AlignInfo**: *Extract information from alignment objects.*
    - **Bio.Align.Applications**: *Alignment command line tool wrappers.*
      - **Bio.Align.Applications._ClustalOmega**: *Command line wrapper for the multiple alignment program Clustal Omega.*
      - **Bio.Align.Applications._Clustalw**: *Command line wrapper for the multiple alignment program Clustal W.*
      - **Bio.Align.Applications._Dialign**: *Command line wrapper for the multiple alignment program DIALIGN2-2.*
      - **Bio.Align.Applications._MSAProbs**: *Command line wrapper for the multiple sequence alignment program MSAProbs.*
      - **Bio.Align.Applications._Mafft**: *Command line wrapper for the multiple alignment programme MAFFT.*
      - **Bio.Align.Applications._Muscle**: *Command line wrapper for the multiple alignment program MUSCLE.*
      - **Bio.Align.Applications._Prank**: *Command line wrapper for the multiple alignment program PRANK.*
      - **Bio.Align.Applications._Probcons**: *Command line wrapper for the multiple alignment program PROBCONS.*
      - **Bio.Align.Applications._TCoffee**: *Command line wrapper for the multiple alignment program TCOFFEE.*
  - **Bio.AlignIO**: *Multiple sequence alignment input/output as alignment objects.*
    - **Bio.AlignIO.ClustalIO**: *Bio.AlignIO support for "clustal" output from CLUSTAL W and other tools.*
    - **Bio.AlignIO.EmbossIO**: *Bio.AlignIO support for "emboss" alignment output from EMBOSS tools.*
    - **Bio.AlignIO.FastaIO**: *Bio.AlignIO support for "fasta-m10" output from Bill Pearson's FASTA tools.*
    - **Bio.AlignIO.Interfaces**: *AlignIO support module (not for general use).*
    - **Bio.AlignIO.MafIO**: *Bio.AlignIO support for the "maf" multiple alignment format.*
    - **Bio.AlignIO.MauveIO**: *Bio.AlignIO support for "xmfa" output from Mauve/ProgressiveMauve.*
    - **Bio.AlignIO.NexusIO**: *Bio.AlignIO support for the "nexus" file format.*
    - **Bio.AlignIO.PhylipIO**: *AlignIO support for "phylip" format from Joe Felsenstein's PHYLIP tools.*
    - **Bio.AlignIO.StockholmIO**: *Bio.AlignIO support for "stockholm" format (used in the PFAM database).*
  - **Bio.Alphabet**: *Alphabets used in Seq objects etc to declare sequence type and letters.*
    - **Bio.Alphabet.IUPAC**: *Standard nucleotide and protein alphabets defined by IUPAC.*
    - **Bio.Alphabet.Reduced**: *Reduced alphabets which lump together several amino-acids into one letter.*
  - **Bio.Application**: *General mechanisms to access applications in Biopython.*
  - **Bio.Blast**: *Code for dealing with BLAST programs and output.*
    - **Bio.Blast.Applications**: *Definitions for interacting with BLAST related applications.*
    - **Bio.Blast.NCBIStandalone**: *Code for calling standalone BLAST and parsing plain text output (DEPRECATED).*
    - **Bio.Blast.NCBIWWW**: *Code to invoke the NCBI BLAST server over the internet.*
    - **Bio.Blast.NCBIXML**: *Code to work with the BLAST XML output.*
    - **Bio.Blast.ParseBlastTable**: *A parser for the NCBI blastpgp version 2.2.5 output format. Currently only supports the '-m 9' option, (table w/ annotations). Returns a BlastTableRec instance*
    - **Bio.Blast.Record**: *Record classes to hold BLAST output.*
  - **Bio.CAPS**: *Cleaved amplified polymorphic sequence (CAPS) markers.*
  - **Bio.Cluster**: *Cluster Analysis*

Sphinx

- Kombajn do tworzenia dokumentacji
- Używa markdownu w komentarzach do generowania wyjściowego HTML
- Stosowany do wielu projektów, np. docs.python.org
- Ogromne możliwości, oczywiście wymaga nauki, żeby je w pełni wykorzystać
- Stosunkowo prosty w użyciu, jeśli mamy prosty projekt

Previous topic
10. Full Grammar specification
Next topic
1. Introduction
This Page
Report a Bug
Show Source

The Python Standard Library

While The Python Language Reference describes the exact syntax and semantics of the Python language, this library reference manual describes the standard library that is distributed with Python. It also describes some of the optional components that are commonly included in Python distributions.

Python's standard library is very extensive, offering a wide range of facilities as indicated by the long table of contents listed below. The library contains built-in modules (written in C) that provide access to system functionality such as file I/O that would otherwise be inaccessible to Python programmers, as well as modules written in Python that provide standardized solutions for many problems that occur in everyday programming. Some of these modules are explicitly designed to encourage and enhance the portability of Python programs by abstracting away platform-specifics into platform-neutral APIs.

The Python installers for the Windows platform usually include the entire standard library and often also include many additional components. For Unix-like operating systems Python is normally provided as a collection of packages, so it may be necessary to use the packaging tools provided with the operating system to obtain some or all of the optional components.

In addition to the standard library, there is a growing collection of several thousand components (from individual programs and modules to packages and entire application development frameworks), available from the Python Package Index.

- 1. Introduction
- 2. Built-in Functions
- 3. Built-in Constants
  - 3.1. Constants added by the site module

Table Of Contents

- 16.4. **argparse** — Parser for command-line options, arguments and sub-commands
  - 16.4.1. Example
    - 16.4.1.1. Creating a parser
    - 16.4.1.2. Adding arguments
    - 16.4.1.3. Parsing arguments
  - 16.4.2. ArgumentParser objects
    - 16.4.2.1. prog
    - 16.4.2.2. usage
    - 16.4.2.3. description
    - 16.4.2.4. epilog
    - 16.4.2.5. parents
    - 16.4.2.6. formatter_class
    - 16.4.2.7. prefix_chars
    - 16.4.2.8. fromfile_prefix_chars
    - 16.4.2.9. argument_default
    - 16.4.2.10. allow_abbrev
    - 16.4.2.11. conflict_handler
    - 16.4.2.12. add_help
  - 16.4.3. The add_argument() method

16.4. `argparse` — Parser for command-line options, arguments and sub-commands

New in version 3.2.

Source code: Lib/argparse.py

The `argparse` module makes it easy to write user-friendly command-line interfaces. The program defines what arguments it requires, and `argparse` will figure out how to parse those out of `sys.argv`. The `argparse` module also automatically generates help and usage messages and issues errors when users give the program invalid arguments.

Tutorial

This page contains the API reference information. For a more gentle introduction to Python command-line parsing, have a look at the argparse tutorial.

16.4.1. Example¶

The following code is a Python program that takes a list of integers and produces either the sum or the max:

```
import argparse

parser = argparse.ArgumentParser(description='Process some integers.')
parser.add_argument('integers', metavar='N', type=int, nargs='+',
                    help='an integer for the accumulator')
parser.add_argument('--sum', dest='accumulate', action='store_const',
                    const=sum, default=max,
                    help='sum the integers (default: find the max)')

args = parser.parse_args()
```

LaTeX

- Bardzo ogólny system do tworzenia plików do wydruku (pdf, dvi)
- Może też generować html (np. przez konwerter HEVEA)
- Popularny w środowisku naukowym
- Niekoniecznie najłatwiejszy do nauki
- Pozwala na eleganckie teksty zawierające notację matematyczną
- Wykorzystywany w biopythonie