Using unit testing to teach data science

self

Assistant professor, Graduate Program in Linguistics, City University of New York

Software engineer, Speech & language algorithms, Google (just part-time these days)

Developer: Pynini, DetectorMorse, Perceptronix

Contributor: OpenFst, OpenGrm, ldamatch, etc.

@wellformedness on Twitter

Outline

- Why test?
- Types of testing
- The `unittest` module
- A worked example: *Unicode-normalization-aware* string comparison

NB: all examples are Python 3.7.

Why test?

Why test?

- Detecting *regressions* while modifying code, so you don’t accidentally make things worse in the course of trying to make them better
- Doing *test-driven development*: when the all the tests pass, you’re done

Exercises

- *Test-writing* exercises: the code is written, you just have to write the tests
- *Test-driven* exercises: the tests are written, you just have to write the code

Testing in the research world

Some researchers see testing as an practice specific to industry. But:

- It is *recommended* for complex personal research software
- It is *essential* for multi-person or multi-site development projects
- It is *expected* for free software libraries you share (i.e., on GitHub, etc.)
- It is *required* in industry (and some students will ultimately go down that path)

Types of testing

Levels of testing

- *Unit tests* test the functionality of a small segment of code
  - In functional programming, the "unit" is usually a function
  - In object-oriented programming, the "unit" usually constructs an instance of a class then verifies its properties/functionality by calling instance methods
- *Integration tests* test the interactions of between multiple components (functions, classes, modules, etc.)
- *System tests* test the end-to-end the behavior of a system

Many other levels, fuzzy distinctions, etc.

Some types of non-tests (1/)

- Sprinkling `assert` statements throughout your code:

```
while mylist:
    head = mylist.pop()
    ...

...
assert not mylist, "List should be empty"
```

Some types of non-tests (2/)

- Argument validation code:

```
def heartbreakingly_brilliant_function(mylist):
    if not mylist:
        raise ValueError("Cannot create heartbreakingly
brilliant results from an empty list")
    ...
```

- Static type checking (e.g., `mypy`, `pytype`)

The `unittest` module

The `unittest` module (1/)

Unit tests consist of a set of test cases. These are expressed as class definitions inheriting from `unittest.TestCase`. Here's a "no-op" one:

```
import unittest

class WorthlessTestCase(unittest.TestCase):

    pass
```

The `unittest` module (2/)

The actual tests are instance methods of this test case class. These must have a identifier starts with `test` and take no arguments (except for `self`).

Within these methods, we perform some kind of computation and assert something about it using an inherited assertion method:

- `self.assertEqual(a, b)`: checks that `a == b` (cf. `assertNotEqual`, `assertAlmostEqual`, etc.)
- `self.assertLess(a, b)`: checks that `a < b` (cf. `assertLessEqual`, `assertGreater`, etc.)
- `self.assertTrue(a)`: checks that `bool(a)` is `True` (cf. `assertFalse`)
- ...

Example

```
class ExampleTestCase(unittest.TestCase):

    def test_three_plus_four_is_less_than_eight(self):
        self.assertLess(3 + 4, 8)

    def test_set_lookup(self):
        determiners = {"a", "an", "the"}
        self.assertIn("the", determiners)
```

Test running

- In a script:

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

- In an Jupyter notebook:

```
_ = unittest.main(argv=[""], exit=False)
```

Test execution

From the command line:

```
$ python nfc_eq_test.py

----------------------------------------------------------------------

Ran 3 tests in 0.001s

OK
```

Unicode normalization-aware string comparison

How do computers encode text?

Text was something of an afterthought to the physicists who invented modern digital computing. They were interested in one thing—numbers—and in some sense, *numbers are the only thing that computers know about*.

Computers thus encode text usings sequences of numbers.

Errors are catastrophic: who wants to send out a ***resumÃ©***?

Glossary

Character: the smallest atomic unit of writing

Character set: a finite, ordered catalog of characters

Encoding: an algorithm for mapping elements of a character set to binary

Decoding: the inverse algorithm, which maps binary to elements of a character set

History

1837: Samuel Morse creates what becomes the International Morse Code.

1963: The American Standards Association (ASA) creates the 7-bit American Standard Code for Information Interchange (ASCII).

1987: The International Standards Organization and the International Electrotechnical Commission publish the first of the 8-bit ISO/IEC 8859 encodings

1991: The Unicode Consortium publishes the first edition of the Unicode Standard.

1993: Ken Thompson and Rob Pike publish UTF-8, a variable-width code for the Unicode character set.

ASCII (1963)

ASCII Code Chart

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2		!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

[Source: Wikipedia Foundation.]

English extensions

But this is not even sufficient for English text because of words like *coöperation* (as it is spelled in the *New Yorker*) or *Motörhead*, an English heavy-metal band.

There are two encoding strategies to handle more than just the ASCII character set:

- We can either use the 8th bit to get another 128 characters,
- or, we can use more than one byte per character.

History

1837: Samuel Morse creates what becomes the International Morse Code.

1963: The American Standards Association (ASA) creates the 7-bit American Standard Code for Information Interchange (ASCII).

1987: The International Standards Organization and the International Electrotechnical Commission publish the first of the 8-bit ISO/IEC 8859 encodings

1991: The Unicode Consortium publishes the first edition of the Unicode Standard.

1993: Ken Thompson and Rob Pike publish UTF-8, a variable-width code for the Unicode character set.

ISO/IEC 8859 (1987 onwards)

Part 1 (“Latin-1”, “Western European”): Danish*, Faroese, Finnish*, French*, German, Icelandic, Irish, Italian, Norwegian, Portuguese, Rhaeto-Romance, Scottish Gaelic, Spanish, Catalan, Swedish

Part 2 (“Latin-2”, “Central European”): Bosnian, Polish, Croatian, Czech, Slovak, Slovene, Serbian, Hungarian

…

Part 5 (“Latin/Cyrillic”): Belarussian, Bulgarian, Macedonian, Russian, Serbian, Ukrainian*

*: Partial support.

ISO/IEC 8859 (1987 onwards)

Part 6 (“Latin/Arabic”)

Part 7 (“Latin/Greek”)

Part 8 (“Latin/Hebrew”)

Part 9 (“Latin/Turkish”)

...

Part 15 (“Latin 9”): Like Part 1, but with a euro sign (€) and letters needed for complete Finnish and French support

Limitation of ISO/IEC 8859

You still can’t write a Ukrainian textbook in Finnish, or write Arabic words in a French cookbook.

History

1837: Samuel Morse creates what becomes the International Morse Code.

1963: The American Standards Association (ASA) creates the 7-bit American Standard Code for Information Interchange (ASCII).

1987: The International Standards Organization and the International Electrotechnical Commission publish the first of the 8-bit ISO/IEC 8859 encodings

1991: The Unicode Consortium publishes the first edition of the Unicode Standard.

1993: Ken Thompson and Rob Pike publish UTF-8, a variable-width code for the Unicode character set.

Unicode (1991)

A massive multilingual character set (over one million characters), grouped by writing system, and associated metadata.

Letter: e
Code point: U+0065
Name: Latin Small Letter e
Script: Latin
Category: Lowercase Letter

Letter: ج
Code point: U+062C
Name: Arabic Letter jeem
Script: Arabic
Category: Other Letter

Letter: ツ
Code point: U+30C4
Name: Katakana Letter tu
Script: Katakana
Category: Letter, Other

Letter: 🤢
Code point: U+1F922
Name: Nauseated Face
Script: Supplemental Symbols And Pictographs
Category: Symbol, Other

Letter: ´
Code point: U+00B4
Name: Acute Accent
Script: Common
Category: Modifier Symbol

Letter: é
Code point: U+00E9
Name: Latin Small Letter e with Acute
Script: Latin
Category: Lowercase Letter

Other Unicode metadata

- Case-folding equivalences (e.g., *A* vs. *a*),
- text direction (left-to-right vs. right-to-left),
- line-breaking and hyphenation rules,
- ligaturing rules,
- etc.

Writing systems

Writing systems are *linguistic technologies*…

….in fact they are the **first** linguistic technologies…

and as such they instantiate a (possibly naïve) linguistic analysis…

and in fact, ancient writing systems are the **first** linguistic analyses.

And the analyses are **not** trivial.

The character

It is not always obvious how to split up text into characters:

- Is *é* one character (“lowercase e with an acute accent”) or two (“lowercase e, followed by an acute accent”)? How about *œ*?
- What about Korean *hangul* characters like 비 <bi>, which can be decomposed into the *jamo* ㅂ <b> and ㅣ <i>?
- In some languages, *digraph* sequences alphabetize as if they were a single character:
  - Dutch: *IJ/ij*
  - Spanish: *Ll/ll* and *Ch/ch* (but not other digraphs like *rr* or *ue*)
  - Croatian: *Dž/dž*, *Lj/lj*, and *Nj/nj*

The problem

[U+00E9]: é

[U+0065 e, U+00B4 ´]: é

Unicode allows us to use “precomposed” or “decomposed” form.

But: `assert "café" != "café"`

The solution: normalization forms

Unicode defines four *normalization forms* which create equivalence classes of visually and/or linguistically similar code sequences.

Two types of equivalence classes—“canonical” and “compatibility”—and “composed” and “decomposed” versions of both:

NFD: Normalization Form Canonical Decomposition
NFC: Normalization Form Canonical Composition
NFKD: Normalization Form Compatibility Decomposition
NFKC: Normalization Form Compatibility Composition

café^2

NFD	U+0063	U+0061	U+0066	U+0065	U+0301	U+00B2
NFC	U+0063	U+0061	U+0066	U+00E9	U+00B2	
NFKD	U+0063	U+0061	U+0066	U+0065	U+0301	U+0032
NFKC	U+0063	U+0061	U+0066	U+00E9	U+0032	

(h/t: Steven Bedrick.)

Suggestions

When taking in arbitrary user input, apply normalization form NFC before performing string comparison.

Your assignment

Create a function `nfc_eq` which performs string comparison *after* applying NFC normalization:

```
def nfc_eq(s1: str, s2: str) -> bool:
    pass
```

Hint: to perform NFC normalization on a string `s`, use:

```
import unicodedata

s: str = unicodedata.normalize(s, "NFC")
```

Test-driven exercise (1/)

```
class NfcEqTest(unittest.TestCase):

    def testTrivialEqualityIsTrue(self):
        s = "foo"
        self.assertTrue(nfc_eq(s, s))

    def testTrivalInequalityIsFalse(self):
        s1 = "foo"
        s2 = "bar"
        self.assertFalse(nfc_eq(s1, s2))
```

Test-driven exercise (2/)

```
...

    def testNfcNfdBibimbapEquality(self):
        s = "비빔밥"
        s1 = unicodedata.normalize(s, "NFC")
        s2 = unicodedata.normalize(s, "NFD")
        self.assertTrue(nfc_eq(s1, s2))
```

My solution

```
import unicodedata

def nfc_eq(s1: str, s2: str) -> bool:
    s1 = unicodedata.normalize(s1, "NFC")
    s2 = unicodedata.normalize(s2, "NFC")
    return s1 == s2
```

Testing advice

Start writing tests when:

- You’re planning on releasing, sharing, or co-developing your code with others
- You know exactly what you want your code to do *but not how to do it yet*
- Your code experiences frequent regressions
- Your code is particularly complex
- Your code is "mission-critical"

Also, consider other testing frameworks, including `doctest` (inline unit tests), `nose` ("lightweight" unit tests), and `pytest` ("modern" unit testing).

Not just for Python anymore...

In R one excellent option is [`testthat`].

In C++ I prefer [`googletest`].

Thanks!