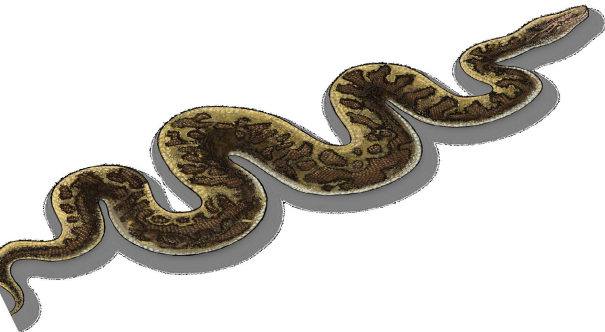


Writing Python in a Nutshell, 4th Edition

Anna Martelli Ravenscroft and Alex Martelli



A short history of the book

2003: 1st edition. (up to) v2.2; martellibot; 600 pages

2006: 2nd edition. (up to) v2.5; Alex (with Anna as tech reviewer); 738 pages (eep, what growth!)

...long time lapse as Alex was not yet using Python 3...

2017: 3rd edition. v2.7 & 3.5; Alex & Anna, with Steve Holden as co-author; 772 pages (way too big!)

2023: 4th edition. V3.7 to 3.11; Alex, Anna, Steve, and Paul McGuire; 735 pages (a little better!)



The process of writing/editing/finalizing

- Location (authors): 2 x Sunnyvale, CA; 1 x Austin, TX; 1 near London, UK
- Tools: Google Spreadsheets, weekly Google Meet meetings, Google Docs... (and more meetings, spreadsheets, docs...)→ ...meeting *all* chapter-by-chapter deadlines (to the editor's surprise!-)
- Writing stylistic issues:
 - Oxford commas and active voice (avoid passives!)
 - Typography (*italics*? **bold**? which font should commas inside code be?)
 - Indicating version numbers, e.g.: **3.9+**
- Reviewers and copyeditor (Rachel Head rocked!) and indexer, oh my!



Some hard decisions

- Not covering
 - `asyncio` & friends (too important: deserves whole books of its own!)
 - key 3rd-party frameworks (`django`, `flask`, `panda`, `PyTorch`, `TensorFlow/Keras`, ..., any GUI, ...)
- Two online-only chapters:
 - <https://github.com/pynutshell/pynut4/tree/main/chapters>
 - Packaging (chapter 24)
 - Extending and Embedding (chapter 25)



Some not-so-hard decisions

- Github repo for all sample code from the book:
<https://github.com/pynutshell/pynut4/tree/main>
- Search PEP abstracts by keywords:
https://ptmcg.pythonanywhere.com/pep_search
- Search for language changes by version:
https://ptmcg.pythonanywhere.com/python_nutshell_app_a_search



A few of our favorite things



Anna's favorite new thing

The `zoneinfo` Module **3.9+**

- Concrete implementation of timezones for use with `tzinfo`
- Uses system timezone data, or `tzdata` as a fallback
 - On Windows, you *may* need to `pip install tzdata`
 - Once `tzdata` installed, you don't need to `import tzdata`
- Get a list of timezone names: `zoneinfo.available_timezones()`
- `zoneinfo` has one class `ZoneInfo`

```
>>> from zoneinfo import ZoneInfo
```



Updating a Timezone

```
>>> d
datetime.datetime(2023, 5, 4, 18, 30,
tzinfo=zoneinfo.ZoneInfo(key='America/New_York'))
>>> d=d.replace(tzinfo=ZoneInfo("America/Los_Angeles"))
>>> d
datetime.datetime(2023, 5, 4, 18, 30,
tzinfo=zoneinfo.ZoneInfo(key='America/Los_Angeles'))
>>> dk=d.astimezone(tz=ZoneInfo("Asia/Katmandu"))
>>> dk
datetime.datetime(2023, 5, 5, 7, 15,
tzinfo=zoneinfo.ZoneInfo(key='Asia/Katmandu'))
# Katmandu offset is +5 hours 45 minutes from UTC (+12:45 ahead of PST)
```



replace() VS astimezone()

replace

- returns new `datetime` with updated timezone without changing other attributes
- pass keyword arg `tzinfo=` to change timezone
- can also be used to replace other attributes, e.g.:

```
>>> d=d.replace(hour=19, minute=0)
```

astimezone

- returns new `datetime` with updated time, date (as needed), & timezone
- won't work with `tzinfo=`, requires keyword arg `tz=`



Alex's favorite (and... *not-so-much*) things

- Since 3.0...
 - text strings default to Unicode (good!)
 - identifiers *may* contain unicode (hmmm...)
 - **They get normalized** (NFKC, *not* NFC as rec'd in UAX#31!)
 - see [PEP 672](https://peps.python.org/pep-0672) (<https://peps.python.org/pep-0672>)
 - and devote your time (I estimate: several months!) to study <https://www.unicode.org/versions/Unicode15.0.0/UnicodeStandard-15.0.pdf> and <https://www.unicode.org/reports/> (all of them!)



Middle Dot “.” is a Valid Identifier Character

```
>>> phone·number = "(512)555-1212"  
>>> # on Mac, Shift-Option-9  
... # on Windows, Alt-0183  
>>>  
>>> print(phone·number)  
(512)555-1212
```

Not valid as a leading character

May cause difficulties due to visual similarity between · and .





Avoid Normalizable Unicode Characters in Identifiers

Python may create unintended aliases between variables when names contain certain Unicode characters, by internally converting the name as shown in the Python script to one using normalized characters. For example, the letters `ª` and `º` normalize to the ASCII lowercase letters `a` and `o`, so variables using these letters could clash with other variables:

```
>>> a, o = 100, 101
>>> ª, º = 200, 201
>>> print(a, o, ª, º)
200 201 200 201 # not "100 101 200 201"
```

It is best to avoid using normalizable Unicode characters in your Python identifiers.



Unicode Identifiers Can Often “normalize to ASCII”

```
def hello():  
    try:  
        hello_ = "Hello"  
        world_ = "World"  
        print(f"{hello_}, {world_}!")  
    except TypeError as exc:  
        print("failed: {}".format(exc))  
  
if __name__ == "__main__":  
    hello()
```

```
def he1lo():  
    try:  
        he1lo_ = "Hello"  
        wo1rld_ = "World"  
        print(f"{he1lo_}, {wo1rld_}!")  
    except TypeError as exc:  
        print("failed: {}".format(exc))  
  
if _|name|_| == "__main__":  
    he1lo()
```

Python Font Mixer

http://ptmcg.pythonanywhere.com/font_mixer



18 Unicode Characters Normalize To 'A'...

A	65	LATIN CAPITAL LETTER A
À	7468	MODIFIER LETTER CAPITAL A
Ⓐ	9398	CIRCLED LATIN CAPITAL LETTER A
Ａ	65313	FULLWIDTH LATIN CAPITAL LETTER A
A	119808	MATHEMATICAL BOLD CAPITAL A
<i>A</i>	119860	MATHEMATICAL ITALIC CAPITAL A
<i>A</i>	119912	MATHEMATICAL BOLD ITALIC CAPITAL A
<i>ℳ</i>	119964	MATHEMATICAL SCRIPT CAPITAL A
<i>ℳ</i>	120016	MATHEMATICAL BOLD SCRIPT CAPITAL A
𝔐	120068	MATHEMATICAL FRAKTUR CAPITAL A
𝔐	120120	MATHEMATICAL DOUBLE-STRUCK CAPITAL A
𝔐	120172	MATHEMATICAL BOLD FRAKTUR CAPITAL A
A	120224	MATHEMATICAL SANS-SERIF CAPITAL A
A	120276	MATHEMATICAL SANS-SERIF BOLD CAPITAL A
<i>A</i>	120328	MATHEMATICAL SANS-SERIF ITALIC CAPITAL A
<i>A</i>	120380	MATHEMATICAL SANS-SERIF BOLD ITALIC CAPITAL A
A	120432	MATHEMATICAL MONOSPACE CAPITAL A
Ⓐ	127280	SQUARED LATIN CAPITAL LETTER A

But **NOT** Α (Greek capital letter α)!



Some tidbits from Paul McGuire

Watch out for `str.lstrip()` and `str.rstrip()`;
they might remove more than you want!

```
>>> print("https://shell.com".lstrip("https://"))  
ell.com
```

`removeprefix()` **3.9+** does what you probably wanted:

```
>>> print("https://shell.com".removeprefix("https://"))  
shell.com
```

There's also a new `removesuffix()` **3.9+**



COUNT HIGHER WITH INTS THAN WITH FLOATS

```
>>> base = 2**53
>>> for i in range(-2, 7):
...     print(f"2**53{i:+} {base+i} {base+float(i)}")
...
2**53-2 9007199254740990 9007199254740990.0
2**53-1 9007199254740991 9007199254740991.0
2**53+0 9007199254740992 9007199254740992.0
2**53+1 9007199254740993 9007199254740992.0
2**53+2 9007199254740994 9007199254740994.0
2**53+3 9007199254740995 9007199254740996.0
2**53+4 9007199254740996 9007199254740996.0
2**53+5 9007199254740997 9007199254740996.0
2**53+6 9007199254740998 9007199254740998.0
```

Floats lose integer precision at 2^{53}

Ints can count up to $2^{(2^{63})}$, or roughly $10^{(2.8e18)}$



<https://www.build-python-from-source.com/>

- Choose Python version
- Choose a few other options
- Generates text for a bash script (good for Linux or WSL)



Python's `sort()` Cited in Supreme Court Case

Python's `sort()` algorithm (known as *TimSort*, named for its author Tim Peters) was part of the source code cited in the case of *Oracle America v. Google, 2012*

The case was generally decided for Google, except for the infringement on TimSort – the judgment against Google for that was \$0

United States District Court
For the Northern District of California

3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28

IN THE UNITED STATES DISTRICT COURT
FOR THE NORTHERN DISTRICT OF CALIFORNIA

ORACLE AMERICA, INC.,

Plaintiff,

No. C 10-03561 WHA

v.

GOOGLE INC.,

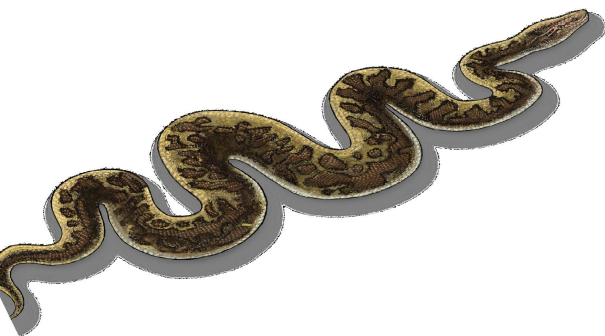
Defendant.

FINAL JUDGMENT

The pleadings in this action asserted the following: Oracle asserted infringement of seven patents, U.S. Patent Nos. 6,125,447; 6,192,476; 5,966,702; 7,426,720; RE38,104; 6,910,205; and 6,061,520. Oracle further asserted infringement of its copyrights in the code, documentation, specifications, libraries, and other materials that comprise the Java platform. Oracle alleged that the infringed elements included Java method and class names, definitions, organization, and parameters; the structure, organization and content of Java class libraries; and the content and organization of Java's documentation. In turn, Google asserted declaratory judgments of non-infringement and invalidity, and equitable defenses. Before trial, Oracle dismissed with prejudice all claims for relief based on the '447, '476, '702, '720, and '205 patents. During trial, Google abandoned claims for relief for invalidity declarations as to the '104 and '520 patents.



A portion of our royalties for *Python in a Nutshell*
are donated to the PSF



Discounts and book signings and Q&A

Discount code for 40% off, good thru May 31, 2023:

<https://play.google.com/redeem?code=61P98Z8G5R20U>

Upcoming book signings (*tentative*):

- North Bay Python <https://2023.northbaypython.org/>
- PyBay <https://www.sfpythonmeetup.com/>

Q&A

