Human Readable

staticdev

Aug 20, 2023

CONTENTS

CHAPTER

ONE

FEATURES

- File size humanization.
- List humanization.
- Numbers humanization.
- Time and dates humanization.
- Internacionalization (i18n) to 20+ locales:
  - Abbreviated English (en_ABBR)
  - Brazilian Portuguese (pt_BR)
  - Dutch (nl_NL)
  - Finnish (fi_FI)
  - French (fr_FR)
  - German (de_DE)
  - Indonesian (id_ID)
  - Italian (it_IT)
  - Japanese (ja_JP)
  - Korean (ko_KR)
  - Persian (fa_IR)
  - Polish (pl_PL)
  - Portugal Portuguese (pt_PT)
  - Russian (ru_RU)
  - Simplified Chinese (zh_CN)
  - Slovak (sk_SK)
  - Spanish (es_ES)
  - Taiwan Chinese (zh_TW)
  - Turkish (tr_TR)
  - Ukrainian (uk_UA)
  - Vietnamese (vi_VI)

CHAPTER

TWO

REQUIREMENTS

- It works in Python 3.8+.

CHAPTER

THREE

INSTALLATION

You can install *Human Readable* via pip from PyPI:

```
$ pip install human-readable
```

CHAPTER

FOUR

BASIC USAGE

Import the lib with:

```
import human_readable
```

Date and time humanization examples:

```
human_readable.time_of_day(17)
"afternoon"

import datetime as dt
human_readable.timing(dt.time(6, 59, 0))
"one minute to seven hours"

human_readable.timing(dt.time(21, 0, 40), formal=False)
"nine in the evening"

human_readable.time_delta(dt.timedelta(days=65))
"2 months"

human_readable.date_time(dt.datetime.now() - dt.timedelta(minutes=2))
"2 minutes ago"

human_readable.day(dt.date.today() - dt.timedelta(days=1))
"yesterday"

human_readable.date(dt.date(2019, 7, 2))
"Jul 02 2019"

human_readable.year(dt.date.today() + dt.timedelta(days=365))
"next year"
```

Precise time delta examples:

```
import datetime as dt
delta = dt.timedelta(seconds=3633, days=2, microseconds=123000)
human_readable.precise_delta(delta)
"2 days, 1 hour and 33.12 seconds"

human_readable.precise_delta(delta, minimum_unit="microseconds")
"2 days, 1 hour, 33 seconds and 123 milliseconds"
```

(continues on next page)

(continued from previous page)

```
human_readable.precise_delta(delta, suppress=["days"], format="0.4f")
"49 hours and 33.1230 seconds"
```

File size humanization examples:

```
human_readable.file_size(1000000)
"1.0 MB"

human_readable.file_size(1000000, binary=True)
"976.6 KiB"

human_readable.file_size(1000000, gnu=True)
"976.6K"
```

Lists humanization examples:

```
human_readable.listing(["Alpha", "Bravo"], ",")
"Alpha, Bravo"

human_readable.listing(["Alpha", "Bravo", "Charlie"], ";", "or")
"Alpha; Bravo or Charlie"
```

Numbers humanization examples:

```
human_readable.int_comma(12345)
"12,345"

human_readable.int_word(123455913)
"123.5 million"

human_readable.int_word(12345591313)
"12.3 billion"

human_readable.ap_number(4)
"four"

human_readable.ap_number(41)
"41"
```

Floating point number humanization examples:

```
human_readable.fractional(1.5)
"1 1/2"

human_readable.fractional(0.3)
"3/10"
```

Scientific notation examples:

```
human_readable.scientific_notation(1000)
"1.00 x 10³"
```

(continues on next page)

(continued from previous page)

```
human_readable.scientific_notation(5781651000, precision=4)
"5.7817 x 10^9"
```

Complete instructions can be found at [human-readable.readthedocs.io].

CHAPTER

FIVE

LOCALIZATION

How to change locale at runtime:

```
import datetime as dt
human_readable.date_time(dt.timedelta(seconds=3))
'3 seconds ago'

_t = human_readable.i18n.activate("ru_RU")
human_readable.date_time(dt.timedelta(seconds=3))
'3  '

human_readable.i18n.deactivate()
human_readable.date_time(dt.timedelta(seconds=3))
'3 seconds ago'
```

You can pass additional parameter `path` to `activate` to specify a path to search locales in.

```
human_readable.i18n.activate("xx_XX")
...
FileNotFoundError: [Errno 2] No translation file found for domain: 'human_readable'
human_readable.i18n.activate("pt_BR", path="path/to/my/portuguese/translation/")
<gettext.GNUTranslations instance ...>
```

You can see how to add a new locale on the *Contributor Guide*.

A special locale, `en_ABBR`, renderes abbreviated versions of output:

```
human_readable.date_time(datetime.timedelta(seconds=3))
3 seconds ago

human_readable.int_word(12345591313)
12.3 billion

human_readable.date_time(datetime.timedelta(seconds=86400*476))
1 year, 3 months ago

human_readable.i18n.activate('en_ABBR')
human_readable.date_time(datetime.timedelta(seconds=3))
3s

human_readable.int_word(12345591313)
12.3 B
```

(continues on next page)

(continued from previous page)

```
human_readable.date_time(datetime.timedelta(seconds=86400*476))
1y 3M
```

CHAPTER

SIX

CONTRIBUTING

Contributions are very welcome. To learn more, see the *Contributor Guide*.

CHAPTER

SEVEN

LICENSE

Distributed under the terms of the *MIT license*, *Human Readable* is free and open source software.

CHAPTER

EIGHT

ISSUES

If you encounter any problems, please file an issue along with a detailed description.

CHAPTER

NINE

CREDITS

This lib is based on original [humanize] with some added features such as listing, improved naming, documentation, functional tests, type-annotations, bug fixes and better localization.

This project was generated from @cjolowicz's Hypermodern Python Cookiecutter template.

9.1 Usage

9.1.1 Basic usage

Import the lib with:

```
import human_readable
```

Date and time humanization examples:

```
human_readable.time_of_day(17)
"afternoon"

import datetime as dt
human_readable.timing(dt.time(6, 59, 0))
"one minute to seven hours"

human_readable.timing(dt.time(21, 0, 40), formal=False)
"nine in the evening"

human_readable.time_delta(dt.timedelta(days=65))
"2 months"

human_readable.date_time(dt.datetime.now() - dt.timedelta(minutes=2))
"2 minutes ago"

human_readable.day(dt.date.today() - dt.timedelta(days=1))
"yesterday"

human_readable.date(dt.date(2019, 7, 2))
"Jul 02 2019"

human_readable.year(dt.date.today() + dt.timedelta(days=365))
"next year"
```

Precise time delta examples:

```
import datetime as dt
delta = dt.timedelta(seconds=3633, days=2, microseconds=123000)
human_readable.precise_delta(delta)
"2 days, 1 hour and 33.12 seconds"

human_readable.precise_delta(delta, minimum_unit="microseconds")
"2 days, 1 hour, 33 seconds and 123 milliseconds"

human_readable.precise_delta(delta, suppress=["days"], format="0.4f")
"49 hours and 33.1230 seconds"
```

File size humanization examples:

```
human_readable.file_size(1000000)
"1.0 MB"

human_readable.file_size(1000000, binary=True)
"976.6 KiB"

human_readable.file_size(1000000, gnu=True)
"976.6K"
```

Lists humanization examples:

```
human_readable.listing(["Alpha", "Bravo"], ",")
"Alpha, Bravo"

human_readable.listing(["Alpha", "Bravo", "Charlie"], ";", "or")
"Alpha; Bravo or Charlie"
```

Numbers humanization examples:

```
human_readable.int_comma(12345)
"12,345"

human_readable.int_word(123455913)
"123.5 million"

human_readable.int_word(12345591313)
"12.3 billion"

human_readable.ap_number(4)
"four"

human_readable.ap_number(41)
"41"
```

Floating point number humanization examples:

```
human_readable.fractional(1.5)
"1 1/2"
```

(continues on next page)

(continued from previous page)

```
human_readable.fractional(0.3)
"3/10"
```

Scientific notation examples:

```
human_readable.scientific_notation(1000)
"1.00 x 10^3"

human_readable.scientific_notation(5781651000, precision=4)
"5.7817 x 10^9"
```

9.1.2 Complete usage

Import the lib with:

```
import human_readable
```

Date and time humanization

time_of_day(hour: int) -> str

Given current hour, returns time of the day.

```
human_readable.time_of_day(17)
"afternoon"
```

timing(time: dt.time, formal: bool = True) -> str

Return human-readable time. Compares time values to present time returns representing readable of time with the given day period.

```
human_readable.timing(dt.time(6, 59, 0))
"one minute to seven hours"
```

You can also specify formal=False, then it won't consider 24h time and instead tell the period of the day. Eg.:

```
human_readable.timing(dt.time(21, 0, 40), formal=False)
"nine in the evening"
```

time_delta(value: dt.timedelta | int | dt.datetime, use_months: bool = True, minimum_unit: str = "seconds", when: dt.datetime | None = None) -> str

Return human-readable time difference. Given a timedelta or a number of seconds, return a natural representation of the amount of time elapsed. This is similar to `date_time`, but does not add tense to the result. If `use_months` is True, then a number of months (based on 30.5 days) will be used for fuzziness between years. Eg.:

```
human_readable.time_delta(dt.timedelta(days=65))
"2 months"
```

Args:

: value: A timedelta or a number of seconds. use_months: If `True`, then a number of months (based on 30.5 days) will be used for fuzziness between years. minimum_unit: The lowest unit that can be used. Options: "years", "months",

"days", "hours", "minutes", "seconds", "milliseconds" or "microseconds". when: Point in time relative to which _value_ is interpreted. Defaults to the current time in the local timezone.

Egs.:

```
human_readable.time_delta(dt.timedelta(hours=4, seconds=3, microseconds=2), minimum_unit=
↪"microseconds")
"2 months"

human_readable.time_delta(dt.timedelta(hours=4, seconds=30, microseconds=200), minimum_
↪unit="microseconds", suppress=["hours"])
"240 minutes, 30 seconds and 200 microseconds"

human_readable.time_delta(dt.datetime.now(), when=dt.datetime.now())
"a moment"
```

date_time(value: dt.timedelta | int | dt.datetime, future: bool = False, use_months: bool = True, minimum_unit: str = "seconds", when: dt.datetime | None = None) -> str

Return human-readable time. Given a datetime or a number of seconds, return a natural representation of that time in a resolution that makes sense. This is more or less compatible with Django's `natural_time` filter. `future` is ignored for datetimes, where the tense is always figured out based on the current time. If an integer is passed, the return value will be past tense by default, unless `future` is set to True.

Eg.:

```
human_readable.date_time(dt.datetime.now() - dt.timedelta(minutes=2))
"2 minutes ago"
```

Args:

: value: time value. future: if false uses past tense. Defaults to False. use_months: if true return number of months. Defaults to True. minimum_unit: The lowest unit that can be used. when: Point in time relative to which _value_ is interpreted. Defaults to the current time in the local timezone.

Specifing `use_months=False` you can supress that unit. Eg.:

```
human_readable.date_time(dt.datetime.now() - dt.timedelta(days=365 + 35), use_
↪months=False)
"1 year, 35 days ago"
```

For usage of `minimum_unit` and `when` just refer to `time_delta()` documentation above.

day(date: dt.date, formatting: str = "%b %d") -> str

Return human-readable day. For date values that are tomorrow, today or yesterday compared to present day returns representing string.

Eg.:

```
human_readable.day(dt.date.today() - dt.timedelta(days=1))
"yesterday"
```

Otherwise, returns a string formatted according to `formatting`.

Eg.:

```
human_readable.day(dt.date(1982, 6, 27), "%Y.%m.%d")
"1982.06.27"
```

date(date: dt.date) -> str

Return human-readable date. Like `day()`, but will append a year for dates that are a year ago or more.

Eg.:

```
human_readable.date(dt.date(2019, 7, 2))
"Jul 02 2019"
```

year(date: dt.date) -> str

Return human-readable year. For date values that are last year, this year or next year compared to present year returns representing string. Otherwise, returns a string formatted according to the year.

Eg.:

```
human_readable.year(dt.date.today() + dt.timedelta(days=365))
"next year"

human_readable.year(dt.date(1988, 11, 12))
"1988"
```

Precise delta humanization

precise_delta(value: dt.timedelta | int, minimum_unit: str = "seconds", suppress: list[str] | None = None, formatting: str = ".2f") -> str

Return a precise representation of a timedelta.

Args:

: value: a time delta. minimum_unit: minimum unit. suppress: list of units to be suppressed. formatting: standard Python format.

Eg.:

```
delta = dt.timedelta(seconds=3633, days=2, microseconds=123000)
human_readable.precise_delta(delta)
"2 days, 1 hour and 33.12 seconds"
```

A custom `formatting` can be specified to control how the fractional part is represented:

Eg.:

```
human_readable.precise_delta(delta, formatting=".4f")
"2 days, 1 hour and 33.1230 seconds"
```

Instead, the `minimum_unit` can be changed to have a better resolution; the function will still readjust the unit to use the greatest of the units that does not lose precision. For example setting microseconds but still representing the date with milliseconds:

Eg.:

```
human_readable.precise_delta(delta, minimum_unit="microseconds")
"2 days, 1 hour, 33 seconds and 123 milliseconds"
```

If desired, some units can be suppressed: you will not see them represented and the time of the other units will be adjusted to keep representing the same timedelta:

Eg.:

```
human_readable.precise_delta(delta, suppress=['days'])
"49 hours and 33.12 seconds"
```

Note that microseconds precision is lost if the seconds and all the units below are suppressed:

Eg.:

```
delta = dt.timedelta(seconds=90, microseconds=100)
human_readable.precise_delta(delta, suppress=['seconds', 'milliseconds', 'microseconds'])
"1.50 minutes"
```

If the delta is too small to be represented with the minimum unit, a value of zero will be returned:

Egs.:

```
delta = dt.timedelta(seconds=1)
human_readable.precise_delta(delta, minimum_unit="minutes")
"0.02 minutes"

delta = dt.timedelta(seconds=0.1)
human_readable.precise_delta(delta, minimum_unit="minutes")
"0 minutes"
```

File size humanization

file_size(value: int, binary: bool = False, gnu: bool = False, formatting: str = ".1f") -> str

Return human-readable file size.

By default, decimal suffixes (kB, MB) are used. Passing binary=true will use binary suffixes (KiB, MiB) are used and the base will be `2**10` instead of `10**3`. If `gnu` is True, the binary argument is ignored and GNU-style (`ls -sh` style) prefixes are used (K, M) with the `2**10` definition. Non-gnu modes are compatible with jinja2's `filesizeformat` filter.

```
human_readable.file_size(1000000)
"1.0 MB"

human_readable.file_size(1000000, binary=True)
"976.6 KiB"

human_readable.file_size(1000000, gnu=True)
"976.6K"
```

You can also specify formatting in standard Python such as ".3f" (default: ".1f"):

```
human_readable.file_size(2900000, gnu=True, formatting=".3f")
"2.766M"
```

List humanization

listing(items: list[str], separator: str, conjunction: str = “”) -> str

Return human readable list separated by separator.

You can use listing with just a list of strings and a separator:

```
human_readable.listing(["Alpha", "Bravo"], ",")
"Alpha, Bravo"
```

You can also specify an optional conjunction, so it is used between last and second last elements.

```
human_readable.listing(["Alpha", "Bravo"], ",", "and")
"Alpha and Bravo"

human_readable.listing(["Alpha", "Bravo", "Charlie"], ";", "or")
"Alpha; Bravo or Charlie"
```

Numbers humanization

ordinal(value: Union[int, str]) -> str

Convert an integer to its ordinal as a string.

```
human_readable.ordinal("1")
"1st"

human_readable.ordinal(111)
"111th"
```

int_comma(value: Union[str, float]) -> str

Convert an integer to a string containing commas every three digits.

For example, 3000 becomes ‘3,000’ and 45000 becomes ‘45,000’. To maintain some compatability with Django’s int_comma, this function also accepts floats.

```
human_readable.int_comma(12345)
"12,345"
```

int_word(value: float, formatting: str = “.1f”) -> str

Convert a large integer to a friendly text representation.

```
human_readable.int_word(123455913)
"123.5 million"

human_readable.int_word(12345591313)
"12.3 billion"
```

You can also specify formatting in standard Python such as “.3f” (default: “.1f”):

```
human_readable.int_word(1230000, "0.2f")
"1.23 million"
```

ap_number(value: Union[float, str]) -> Union[str, float]

For numbers 1-9, returns the number spelled out. Otherwise, returns the number.

This follows Associated Press style numbering:

```
human_readable.ap_number(4)
"four"

human_readable.ap_number(41)
"41"
```

Floating point number humanization

fractional(value: Union[str, float]) -> str

Return a human readable fractional number.

```
human_readable.fractional(1.5)
"1 1/2"

human_readable.fractional(0.3)
"3/10"
```

Scientific notation

scientific_notation(value: Union[float, str], precision: int = 2) -> str

Return number in string scientific notation z.wq x 10.

```
human_readable.scientific_notation(1000)
"1.00 x 10^3"
```

You can also specify precision to it (default: 2):

```
human_readable.scientific_notation(5781651000, precision=4)
"5.7817 x 10^9"
```

9.2 Contributor Guide

Thank you for your interest in improving this project. This project is open-source under the MIT license and welcomes contributions in the form of bug reports, feature requests, and pull requests.

Here is a list of important resources for contributors:

- Source Code
- Documentation
- Issue Tracker
- *Code of Conduct*

9.2.1 How to report a bug

Report bugs on the Issue Tracker.

When filing an issue, make sure to answer these questions:

- Which operating system and Python version are you using?
- Which version of this project are you using?
- What did you do?
- What did you expect to see?
- What did you see instead?

The best way to get your bug fixed is to provide a test case, and/or steps to reproduce the issue.

9.2.2 How to request a feature

Request features on the Issue Tracker.

9.2.3 How to set up your development environment

You need Python 3.8+ and the following tools:

- Hatch

Install the package with development requirements:

```
$ hatch env create
```

You can now run an interactive Python session:

```
$ hatch shell
import human_readable
```

9.2.4 How to test the project

Run the full test suite:

```
$ hatch run all
```

List the available Hatch env scripts:

```
$ hatch env show
```

You can run a specific Hatch env script. For example, invoke the unit test suite like this:

```
$ hatch run tests:run
```

Unit tests are located in the `tests` directory, and are written using the pytest testing framework.

9.2.5 How to add a new locale

Make sure you have installed a PO Editor, you can easily install that on a Debian-based system with:

```
apt install poedit
```

To add a new locale you need to execute:

```
xgettext --from-code=UTF-8 -o human_readable.pot -k'_' -k'N_' -k'P_:1c,2' -l python src/
↪human_readable/*.py  # extract new phrases
msginit -i human_readable.pot -o human_readable/locale/<locale name>/LC_MESSAGES/human_
↪readable.po --locale <locale name>
```

Then edit your .po file in the locale folder that got created.

If possible, add tests to `tests/functional/new_locale`. Run them with:

```
nox -s tests
```

9.2.6 How to submit changes

Open a pull request to submit changes to this project.

Your pull request needs to meet the following guidelines for acceptance:

- The Nox test suite must pass without errors and warnings.
- Include unit tests. This project maintains 100% code coverage.
- If your changes add functionality, update the documentation accordingly.

Feel free to submit early, though—we can always iterate on this.

To run linting and code formatting checks before commiting your change, you can install pre-commit as a Git hook by running the following command:

```
$ nox --session=pre-commit -- install
```

It is recommended to open an issue before starting work on anything. This will allow a chance to talk it over with the owners and validate your approach.

9.3 Contributor Covenant Code of Conduct

9.3.1 Our Pledge

We as members, contributors, and leaders pledge to make participation in our community a harassment-free experience for everyone, regardless of age, body size, visible or invisible disability, ethnicity, sex characteristics, gender identity and expression, level of experience, education, socio-economic status, nationality, personal appearance, race, religion, or sexual identity and orientation.

We pledge to act and interact in ways that contribute to an open, welcoming, diverse, inclusive, and healthy community.

9.3.2 Our Standards

Examples of behavior that contributes to a positive environment for our community include:

- Demonstrating empathy and kindness toward other people
- Being respectful of differing opinions, viewpoints, and experiences
- Giving and gracefully accepting constructive feedback
- Accepting responsibility and apologizing to those affected by our mistakes, and learning from the experience
- Focusing on what is best not just for us as individuals, but for the overall community

Examples of unacceptable behavior include:

- The use of sexualized language or imagery, and sexual attention or advances of any kind
- Trolling, insulting or derogatory comments, and personal or political attacks
- Public or private harassment
- Publishing others' private information, such as a physical or email address, without their explicit permission
- Other conduct which could reasonably be considered inappropriate in a professional setting

9.3.3 Enforcement Responsibilities

Community leaders are responsible for clarifying and enforcing our standards of acceptable behavior and will take appropriate and fair corrective action in response to any behavior that they deem inappropriate, threatening, offensive, or harmful.

Community leaders have the right and responsibility to remove, edit, or reject comments, commits, code, wiki edits, issues, and other contributions that are not aligned to this Code of Conduct, and will communicate reasons for moderation decisions when appropriate.

9.3.4 Scope

This Code of Conduct applies within all community spaces, and also applies when an individual is officially representing the community in public spaces. Examples of representing our community include using an official e-mail address, posting via an official social media account, or acting as an appointed representative at an online or offline event.

9.3.5 Enforcement

Instances of abusive, harassing, or otherwise unacceptable behavior may be reported to the community leaders responsible for enforcement at mailto:staticdev-support@protonmail.com. All complaints will be reviewed and investigated promptly and fairly.

All community leaders are obligated to respect the privacy and security of the reporter of any incident.

9.3.6 Enforcement Guidelines

Community leaders will follow these Community Impact Guidelines in determining the consequences for any action they deem in violation of this Code of Conduct:

1. Correction

Community Impact: Use of inappropriate language or other behavior deemed unprofessional or unwelcome in the community.

Consequence: A private, written warning from community leaders, providing clarity around the nature of the violation and an explanation of why the behavior was inappropriate. A public apology may be requested.

2. Warning

Community Impact: A violation through a single incident or series of actions.

Consequence: A warning with consequences for continued behavior. No interaction with the people involved, including unsolicited interaction with those enforcing the Code of Conduct, for a specified period of time. This includes avoiding interactions in community spaces as well as external channels like social media. Violating these terms may lead to a temporary or permanent ban.

3. Temporary Ban

Community Impact: A serious violation of community standards, including sustained inappropriate behavior.

Consequence: A temporary ban from any sort of interaction or public communication with the community for a specified period of time. No public or private interaction with the people involved, including unsolicited interaction with those enforcing the Code of Conduct, is allowed during this period. Violating these terms may lead to a permanent ban.

4. Permanent Ban

Community Impact: Demonstrating a pattern of violation of community standards, including sustained inappropriate behavior, harassment of an individual, or aggression toward or disparagement of classes of individuals.

Consequence: A permanent ban from any sort of public interaction within the community.

9.3.7 Attribution

This Code of Conduct is adapted from the Contributor Covenant, version 2.0, available at https://www.contributor-covenant.org/version/2/0/code_of_conduct.html.

Community Impact Guidelines were inspired by Mozilla's code of conduct enforcement ladder.

For answers to common questions about this code of conduct, see the FAQ at https://www.contributor-covenant.org/faq. Translations are available at https://www.contributor-covenant.org/translations.

9.4 License

Copyright (C) 2022 by staticdev

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in
all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN
THE SOFTWARE.