inflect 0.3.1

Latest version

```
pip install inflect
```

Last released: May 8, 2018

Correctly generate plurals, singular nouns, ordinals, indefinite articles; convert numbers to words

Navigation

Project description

Release history

Download files

Project links

Homepage

Project description

This is a Jazzband project. By contributing you agree to abide by the Contributor Code of Conduct and follow the guidelines.

NAME

inflect.py - Correctly generate plurals, singular nouns, ordinals, indefinite articles; convert numbers to words.

Statistics

GitHub statistics:

- **Stars:** 190
- **Forks:** 29
- **Open issues/PRs:** 21

View statistics for this project via Libraries.io, or by using Google BigQuery

Meta

License: GNU Affero General Public License v3

Author: Alex Gronholm

plural

Maintainers


agronholm


jaraco


pwdyson

INSTALLATION

```
pip install inflect
```

SYNOPSIS

```
import inflect
p = inflect.engine()

# METHODS:

# plural plural_noun plural_verb plural_adj sin
# compare compare_nouns compare_nouns compare_a
# a an
# present_participle
# ordinal number_to_words
# join
# inflect classical gender
# defnoun defverb defadj defa defan


# UNCONDITIONALLY FORM THE PLURAL

     print("The plural of ", word, " is ", p.pl


# CONDITIONALLY FORM THE PLURAL

     print("I saw", cat_count, p.plural("cat",c


# FORM PLURALS FOR SPECIFIC PARTS OF SPEECH

     print(p.plural_noun("I",N1), p.plural_verb
           p.plural_noun("saw",N2)


# FORM THE SINGULAR OF PLURAL NOUNS
```

Classifiers

Development Status
- 4 - Beta

Intended Audience
- Developers

License
- OSI Approved :: GNU Affero General Public License v3

Natural Language
- English

Operating System
- OS Independent

Programming Language
- Python
- Python :: 2.7
- Python :: 3
- Python :: 3.3
- Python :: 3.4
- Python :: 3.5
- Python :: 3.6

Topic
- Software Development :: Libraries :: Python Modules
- Text Processing :: Linguistic

```
    print("The singular of ", word, " is ", p.

# SELECT THE GENDER OF SINGULAR PRONOUNS

    print(p.singular_noun('they')  # 'it')
    p.gender('f')
    print(p.singular_noun('they')  # 'she')


# DEAL WITH "0/1/N" -> "no/1/N" TRANSLATION:

    print("There ", p.plural_verb("was",errors


# USE DEFAULT COUNTS:

    print(p.num(N1,""), p.plural("I"), p.plura
    print("There ", p.num(errors,''), p.plural


# COMPARE TWO WORDS "NUMBER-INSENSITIVELY":

    print("same\n"      if p.compare(word1, wo
    print("same noun\n" if p.compare_nouns(wor
    print("same verb\n" if p.compare_verbs(wor
    print("same adj.\n" if p.compare_adjs(word


# ADD CORRECT "a" OR "an" FOR A GIVEN WORD:

    print("Did you want ", p.a(thing), " or ",


# CONVERT NUMERALS INTO ORDINALS (i.e. 1->1st,

    print("It was", p.ordinal(position), " fro

# CONVERT NUMERALS TO WORDS (i.e. 1->"one", 101
# RETURNS A SINGLE STRING...

   words = p.number_to_words(1234)      # "one
   words = p.number_to_words(p.ordinal(1234)) #
```

```
# GET BACK A LIST OF STRINGS, ONE FOR EACH "CHU

   words = p.number_to_words(1234, getlist=True


# OPTIONAL PARAMETERS CHANGE TRANSLATION:

   words = p.number_to_words(12345, group=1)
   # "one, two, three, four, five"

   words = p.number_to_words(12345, group=2)
   # "twelve, thirty-four, five"

   words = p.number_to_words(12345, group=3)
   # "one twenty-three, forty-five"

   words = p.number_to_words(1234, andword='')
   # "one thousand, two hundred thirty-four"

   words = p.number_to_words(1234, andword=', p
   # "one thousand, two hundred, plus thirty-fo

   words = p.number_to_words(555_1202, group=1,
   # "five, five, five, one, two, oh, two"

   words = p.number_to_words(555_1202, group=1,
   # "five, five, five, unity, two, oh, two"

   words = p.number_to_words(123.456, group=1,
   # "one two three mark four five six"  #TODO:

# LITERAL STYLE ONLY NAMES NUMBERS LESS THAN A

   words = p.number_to_words(   9, threshold=10
   words = p.number_to_words(  10, threshold=10
   words = p.number_to_words(  11, threshold=10
   words = p.number_to_words(1000, threshold=10

# JOIN WORDS INTO A LIST:
```

```
   mylist = join(("apple", "banana", "carrot"))
       # "apple, banana, and carrot"

   mylist = join(("apple", "banana"))
       # "apple and banana"

   mylist = join(("apple", "banana", "carrot"),
       # "apple, banana and carrot"


# REQUIRE "CLASSICAL" PLURALS (EG: "focus"->"fo

     p.classical()               # USE ALL CLAS

     p.classical(all=True)       # USE ALL CLAS
     p.classical(all=False)      # SWITCH OFF C

     p.classical(zero=True)      #  "no error"
     p.classical(zero=False)     #  "no errors"

     p.classical(herd=True)      #  "2 buffalo"
     p.classical(herd=False)     #  "2 buffalos

     p.classical(persons=True)   # "2 chairpers
     p.classical(persons=False)  # "2 chairpeop

     p.classical(ancient=True)   # "2 formulae"
     p.classical(ancient=False)  # "2 formulas"



# INTERPOLATE "plural()", "plural_noun()", "plu
# a()", "an()", "num()" AND "ordinal()" WITHIN

     print(p.inflect("The plural of {0} is plur
     print(p.inflect("The singular of {0} is si
     print(p.inflect("I saw {0} plural("cat",{0
     print(p.inflect("plural(I,{0}) plural_verb
     print(p.inflect("num({0},)plural(I) plural
     print(p.inflect("I saw num({0}) plural("ca
     print(p.inflect("There plural_verb(was,{0}
     print(p.inflect("There num({0},) plural_ve
```

```
     print(p.inflect("Did you want a({0}) or an
     print(p.inflect("It was ordinal({0}) from


# ADD USER-DEFINED INFLECTIONS (OVERRIDING INBU

     p.defnoun( "VAX", "VAXen" )  # SINGULAR =>

     p.defverb( "will" , "shall",  # 1ST PERSON
                "will" , "will",   # 2ND PERSON
                "will" , "will")   # 3RD PERSON

     p.defadj(  "hir"  , "their")  # SINGULAR =

     p.defa("h")        # "AY HALWAYS SEZ 'HAIT

     p.defan(   "horrendous.*" )    # "AN HORRE
```

DESCRIPTION

The methods of the class `engine` in module `inflect.py` provide plural inflections, singular noun inflections, “a”/”an” selection for English words, and manipulation of numbers as words.

Plural forms of all nouns, most verbs, and some adjectives are provided. Where appropriate, “classical” variants (for example: “brother” -> “brethren”, “dogma” -> “dogmata”, etc.) are also provided.

Single forms of nouns are also provided. The gender of singular pronouns can be chosen (for example “they” -> “it” or “she” or “he” or “they”).

Pronunciation-based “a”/”an” selection is provided for all English words, and most initialisms.

It is also possible to inflect numerals (1,2,3) to ordinals (1st, 2nd, 3rd) and to English words ("one", "two", "three").

In generating these inflections, `inflect.py` follows the Oxford English Dictionary and the guidelines in Fowler's Modern English Usage, preferring the former where the two disagree.

The module is built around standard British spelling, but is designed to cope with common American variants as well. Slang, jargon, and other English dialects are *not* explicitly catered for.

Where two or more inflected forms exist for a single word (typically a "classical" form and a "modern" form), `inflect.py` prefers the more common form (typically the "modern" one), unless "classical" processing has been specified (see *MODERN VS CLASSICAL INFLECTIONS*).

FORMING PLURALS AND SINGULARS

Inflecting Plurals and Singulars

All of the `plural...` plural inflection methods take the word to be inflected as their first argument and return the corresponding inflection. Note that all such methods expect the *singular* form of the word. The results of passing a plural form are undefined (and unlikely to be correct). Similarly, the `si...` singular inflection method expects the *plural* form of the word.

The `plural...` methods also take an optional second argument, which indicates the grammatical "number" of the word (or of another word with which the word being inflected must agree). If the "number" argument is supplied and is not `1` (or `"one"` or `"a"`, or some other adjective

that implies the singular), the plural form of the word is returned. If the “number” argument *does* indicate singularity, the (uninflected) word itself is returned. If the number argument is omitted, the plural form is returned unconditionally.

The `si...` method takes a second argument in a similar fashion. If it is some form of the number `1`, or is omitted, the singular form is returned. Otherwise the plural is returned unaltered.

The various methods of `inflect.engine` are:

`plural_noun(word, count=None)`

> *The method `plural_noun()` takes a singular English noun or pronoun and returns its plural. Pronouns in the nominative (“I” -> “we”) and accusative (“me” -> “us”) cases are handled, as are possessive pronouns (“mine” -> “ours”).*

`plural_verb(word, count=None)`

> *The method `plural_verb()` takes the singular form of a conjugated verb (that is, one which is already in the correct “person” and “mood”) and returns the corresponding plural conjugation.*

`plural_adj(word, count=None)`

> *The method `plural_adj()` takes the singular form of certain types of adjectives and returns the corresponding plural form. Adjectives that are correctly handled include: “numerical” adjectives (“a” -> “some”), demonstrative adjectives (“this” -> “these”, “that” -> “those”), and possessives (“my” -> “our”, “cat’s” -> “cats’”, “child’s” -> “childrens’”, etc.)*

`plural(word, count=None)`

> *The method `plural()` takes a singular English noun, pronoun, verb, or adjective and returns its plural form. Where a word has more than one inflection depending on its part of speech (for example, the noun "thought" inflects to "thoughts", the verb "thought" to "thought"), the (singular) noun sense is preferred to the (singular) verb sense.*
>
> *Hence `plural("knife")` will return "knives" ("knife" having been treated as a singular noun), whereas `plural("knifes")` will return "knife" ("knifes" having been treated as a 3rd person singular verb).*
>
> *The inherent ambiguity of such cases suggests that, where the part of speech is known, `plural_noun`, `plural_verb`, and `plural_adj` should be used in preference to `plural`.*

`singular_noun(word, count=None)`

> *The method `singular_noun()` takes a plural English noun or pronoun and returns its singular. Pronouns in the nominative ("we" -> "I") and accusative ("us" -> "me") cases are handled, as are possessive pronouns ("ours" -> "mine"). When third person singular pronouns are returned they take the neuter gender by default ("they" -> "it"), not ("they"-> "she") nor ("they" -> "he"). This can be changed with `gender()`.*

Note that all these methods ignore any whitespace surrounding the word being inflected, but preserve that whitespace when the result is returned. For example, `plural(" cat  ")` returns " cats ".

`gender(genderletter)`

> *The third person plural pronoun takes the same form for the female, male and neuter (e.g. “they”). The singular however, depends upon gender (e.g. “she”, “he”, “it” and “they” – “they” being the gender neutral form.) By default `singular_noun` returns the neuter form, however, the gender can be selected with the `gender` method. Pass the first letter of the gender to `gender` to return the f(eminine), m(asculine), n(euter) or t(hey) form of the singular. e.g. gender(‘f’) followed by singular_noun(‘themselves’) returns ‘herself’.*

Numbered plurals

The `plural...` methods return only the inflected word, not the count that was used to inflect it. Thus, in order to produce “I saw 3 ducks”, it is necessary to use:

```
print("I saw", N, p.plural_noun(animal,N))
```

Since the usual purpose of producing a plural is to make it agree with a preceding count, inflect.py provides a method (`no(word, count)`) which, given a word and a(n optional) count, returns the count followed by the correctly inflected word. Hence the previous example can be rewritten:

```
print("I saw ", p.no(animal,N))
```

In addition, if the count is zero (or some other term which implies zero, such as `"zero"`, `"nil"`, etc.) the count is replaced by the word “no”. Hence, if `N` had the value zero, the previous example would print(the somewhat more elegant::)

> *I saw no animals*

rather than:

```
I saw 0 animals
```

Note that the name of the method is a pun: the method returns either a number (a *No.*) or a `"no"`, in front of the inflected word.

Reducing the number of counts required

In some contexts, the need to supply an explicit count to the various `plural...` methods makes for tiresome repetition. For example:

```
print(plural_adj("This",errors), plural_noun("
      plural_verb(" was",errors), " fatal."
```

inflect.py therefore provides a method (`num(count=None, show=None)`) which may be used to set a persistent "default number" value. If such a value is set, it is subsequently used whenever an optional second "number" argument is omitted. The default value thus set can subsequently be removed by calling `num()` with no arguments. Hence we could rewrite the previous example:

```
p.num(errors)
print(p.plural_adj("This"), p.plural_noun(" err
p.num()
```

Normally, `num()` returns its first argument, so that it may also be “inlined” in contexts like:

```
print(p.num(errors), p.plural_noun(" error"), p
if severity > 1:
    print(p.plural_adj("This"), p.plural_noun("
```

However, in certain contexts (see *INTERPOLATING INFLECTIONS IN STRINGS*) it is preferable that `num()` return an empty string. Hence `num()` provides an optional second argument. If that argument is supplied (that is, if it is defined) and evaluates to false, `num` returns an empty string instead of its first argument. For example:

```
print(p.num(errors,0), p.no("error"), p.plural_
if severity > 1:
    print(p.plural_adj("This"), p.plural_noun("
```

Number-insensitive equality

inflect.py also provides a solution to the problem of comparing words of differing plurality through the methods `compare(word1, word2)`, `compare_nouns(word1, word2)`, `compare_verbs(word1, word2)`, and `compare_adjs(word1, word2)`. Each of these methods takes two strings, and compares them using the corresponding plural-inflection method (`plural()`, `plural_noun()`, `plural_verb()`, and `plural_adj()` respectively).

The comparison returns true if:

- the strings are equal, or

- one string is equal to a plural form of the other, or
- the strings are two different plural forms of the one word.

Hence all of the following return true:

```
p.compare("index","index")      # RETURNS "eq"
p.compare("index","indexes")    # RETURNS "s:p"
p.compare("index","indices")    # RETURNS "s:p"
p.compare("indexes","index")    # RETURNS "p:s"
p.compare("indices","index")    # RETURNS "p:s"
p.compare("indices","indexes")  # RETURNS "p:p"
p.compare("indexes","indices")  # RETURNS "p:p"
p.compare("indices","indices")  # RETURNS "eq"
```

As indicated by the comments in the previous example, the actual value returned by the various `compare` methods encodes which of the three equality rules succeeded: “eq” is returned if the strings were identical, “s:p” if the strings were singular and plural respectively, “p:s” for plural and singular, and “p:p” for two distinct plurals. Inequality is indicated by returning an empty string.

It should be noted that two distinct singular words which happen to take the same plural form are *not* considered equal, nor are cases where one (singular) word’s plural is the other (plural) word’s singular. Hence all of the following return false:

```
p.compare("base","basis")       # ALTHOUGH BOTH
p.compare("syrinx","syringe")   # ALTHOUGH BOTH
p.compare("she","he")           # ALTHOUGH BOTH

p.compare("opus","operas")      # ALTHOUGH "opu
p.compare("taxi","taxes")       # ALTHOUGH "tax
```

Note too that, although the comparison is “number-insensitive” it is *not* case-insensitive (that is, `plural("time","Times")` returns false. To obtain both number and case insensitivity, use the `lower()` method on both strings (that is, `plural("time".lower(), "Times".lower())` returns true).

OTHER VERB FORMS

Present participles

`inflect.py` also provides the `present_participle` method, which can take a 3rd person singular verb and correctly inflect it to its present participle:

```
p.present_participle("runs")   # "running"
p.present_participle("loves")  # "loving"
p.present_participle("eats")   # "eating"
p.present_participle("bats")   # "batting"
p.present_participle("spies")  # "spying"
```

PROVIDING INDEFINITE ARTICLES

Selecting indefinite articles

inflect.py provides two methods (`a(word, count=None)` and `an(word, count=None)`) which will correctly prepend the appropriate indefinite article to a word, depending on its pronunciation. For example:

```
p.a("cat")        # -> "a cat"
p.an("cat")       # -> "a cat"
```

```
p.a("euphemism")      # -> "a euphemism"
p.a("Euler number")   # -> "an Euler number"
p.a("hour")       # -> "an hour"
p.a("houri")      # -> "a houri"
```

The two methods are *identical* in function and may be used interchangeably. The only reason that two versions are provided is to enhance the readability of code such as:

```
print("That is ", an(errortype), " error)
print("That is ", a(fataltype), " fatal error)
```

Note that in both cases the actual article provided depends *only* on the pronunciation of the first argument, *not* on the name of the method.

`a()` and `an()` will ignore any indefinite article that already exists at the start of the string. Thus:

```
half_arked = [
    "a elephant",
    "a giraffe",
    "an ewe",
    "a orangutan",
]

for txt in half_arked:
    print(p.a(txt))

# prints:
#     an elephant
#     a giraffe
#     a ewe
#     an orangutan
```

`a()` and `an()` both take an optional second argument. As with the `plural...` methods, this second argument is a

"number" specifier. If its value is `1` (or some other value implying singularity), `a()` and `an()` insert "a" or "an" as appropriate. If the number specifier implies plurality, (`a()` and `an()` insert the actual second argument instead. For example:

```
p.a("cat",1)      # -> "a cat"
p.a("cat",2)      # -> "2 cat"
p.a("cat","one")      # -> "one cat"
p.a("cat","no")       # -> "no cat"
```

Note that, as implied by the previous examples, `a()` and `an()` both assume that their job is merely to provide the correct qualifier for a word (that is: "a", "an", or the specified count). In other words, they assume that the word they are given has already been correctly inflected for plurality. Hence, if `N` has the value 2, then:

```
print(p.a("cat",N))
```

prints "2 cat", instead of "2 cats". The correct approach is to use:

```
print(p.a(p.plural("cat",N),N))
```

or, better still:

```
print(p.no("cat",N))
```

Note too that, like the various `plural...` methods, whenever `a()` and `an()` are called with only one argument

they are subject to the effects of any preceding call to `num()`. Hence, another possible solution is:

```
p.num(N)
print(p.a(p.plural("cat")))
```

Indefinite articles and initialisms

"Initialisms" (sometimes inaccurately called "acronyms") are terms which have been formed from the initial letters of words in a phrase (for example, "NATO", "NBL", "S.O.S.", "SCUBA", etc.)

Such terms present a particular challenge when selecting between "a" and "an", since they are sometimes pronounced as if they were a single word ("nay-tow", "sku-ba") and sometimes as a series of letter names ("en-eff-ell", "ess-oh-ess").

`a()` and `an()` cope with this dichotomy using a series of inbuilt rules, which may be summarized as:

> *If the word starts with a single letter, followed by a period or dash (for example, "R.I.P.", "C.O.D.", "e-mail", "X-ray", "T-square"), then choose the appropriate article for the sound of the first letter ("an R.I.P.", "a C.O.D.", "an e-mail", "an X-ray", "a T-square").*
>
> *If the first two letters of the word are capitals, consonants, and do not appear at the start of any known English word, (for example, "LCD", "XML", "YWCA"), then once again choose "a" or "an" depending on the sound of the first letter ("an LCD", "an XML", "a YWCA").*
>
> *Otherwise, assume the string is a capitalized word or a pronounceable initialism (for example, "LED", "OPEC",*

> *“FAQ”, “UNESCO”), and therefore takes “a” or “an” according to the (apparent) pronunciation of the entire word (“a LED”, “an OPEC”, “a FAQ”, “a UNESCO”).*

Note that rules 1 and 3 together imply that the presence or absence of punctuation may change the selection of indefinite article for a particular initialism (for example, “a FAQ” but “an F.A.Q.”).

Indefinite articles and “soft H’s”

Words beginning in the letter ‘H’ present another type of difficulty when selecting a suitable indefinite article. In a few such words (for example, “hour”, “honour”, “heir”) the ‘H’ is not voiced at all, and so such words inflect with “an”. The remaining cases (“voiced H’s”) may be divided into two categories: “hard H’s” (such as “hangman”, “holograph”, “hat”, etc.) and “soft H’s” (such as “hysterical”, “horrendous”, “holy”, etc.)

Hard H’s always take “a” as their indefinite article, and soft H’s normally do so as well. But *some* English speakers prefer “an” for soft H’s (although the practice is now generally considered an affectation, rather than a legitimate grammatical alternative).

At present, the `a()` and `an()` methods ignore soft H’s and use “a” for any voiced ‘H’. The author would, however, welcome feedback on this decision (envisaging a possible future “soft H” mode).

INFLECTING ORDINALS

Occasionally it is useful to present an integer value as an ordinal rather than as a numeral. For example:

```
Enter password (1st attempt): ********
Enter password (2nd attempt): *********
Enter password (3rd attempt): *********
No 4th attempt. Access denied.
```

To this end, inflect.py provides the `ordinal()` method. `ordinal()` takes a single argument and forms its ordinal equivalent. If the argument isn't a numerical integer, it just adds "-th".

CONVERTING NUMBERS TO WORDS

The method `number_to_words` takes a number (cardinal or ordinal) and returns an English representation of that number.

```
word = p.number_to_words(1234567)
```

puts the string:

```
"one million, two hundred and thirty-four thous
```

into `words`.

A list can be return where each comma-separated chunk is returned as a separate element. Hence:

```
words = p.number_to_words(1234567, wantlist=Tru
```

puts the list:

```
["one million",
 "two hundred and thirty-four thousand",
 "five hundred and sixty-seven"]
```

into `words`.

Non-digits (apart from an optional leading plus or minus sign, any decimal points, and ordinal suffixes – see below) are silently ignored, so the following all produce identical results:

```
p.number_to_words(5551202)
p.number_to_words(5_551_202)
p.number_to_words("5,551,202")
p.number_to_words("555-1202")
```

That last case is a little awkward since it's almost certainly a phone number, and "five million, five hundred and fifty-one thousand, two hundred and two" probably isn't what's wanted.

To overcome this, `number_to_words()` takes an optional argument, 'group', which changes how numbers are translated. The argument must be a positive integer less than four, which indicated how the digits of the number are to be grouped. If the argument is `1`, then each digit is translated separately. If the argument is `2`, pairs of digits (starting from the *left*) are grouped together. If the argument is `3`, triples of numbers (again, from the *left*) are grouped. Hence:

```
p.number_to_words("555-1202", group=1)
```

returns `"five, five, five, one, two, zero, two"`, whilst:

```
p.number_to_words("555-1202", group=2)
```

returns `"fifty-five, fifty-one, twenty, two"`, and:

```
p.number_to_words("555-1202", group=3)
```

returns `"five fifty-five, one twenty, two"`.

Phone numbers are often written in words as `"five..five..five..one..two..zero..two"`, which is also easy to achieve:

```
join '..', p.number_to_words("555-1202", group=
```

`number_to_words` also handles decimal fractions. Hence:

```
p.number_to_words("1.2345")
```

returns `"one point two three four five"` in a scalar context and `("one","point","two","three","four","five")` in an array context. Exponent form (`"1.234e56"`) is not yet handled.

Multiple decimal points are only translated in one of the “grouping” modes. Hence:

```
p.number_to_words(101.202.303)
```

returns `"one hundred and one point two zero two three zero three"`, whereas:

```
p.number_to_words(101.202.303, group=1)
```

returns `"one zero one point two zero two point three zero three"`.

The digit `'0'` is unusual in that in may be translated to English as “zero”, “oh”, or “nought”. To cater for this diversity, `number_to_words` may be passed a named argument, ‘zero’, which may be set to the desired translation of `'0'`. For example:

```
print(join "..", p.number_to_words("555-1202",
```

prints `"five..five..five..one..two..oh..two"`. By default, zero is rendered as “zero”.

Likewise, the digit `'1'` may be rendered as “one” or “a/an” (or very occasionally other variants), depending on the context. So there is a `'one'` argument as well:

```
for num in [3,2,1,0]:
      print(p.number_to_words(num, one='a solit
      p.plural(" bottle of beer on the wall", n

# prints:
#     three bottles of beer on the wall
#     two bottles of beer on the wall
#     a solitary bottle of beer on the wall
#     no more bottles of beer on the wall
```

Care is needed if the word “a/an” is to be used as a `'one'` value. Unless the next word is known in advance, it’s almost always necessary to use the `A` function as well:

```
    for word in ["cat aardvark ewe hour".split(
        print(p.a("{0} {1}".format(p.number_to_

# prints:
#     a cat
#     an aardvark
#     a ewe
#     an hour
```

Another major regional variation in number translation is the use of “and” in certain contexts. The named argument ‘and’ allows the programmer to specify how “and” should be handled. Hence:

```
print(scalar p.number_to_words("765", andword='
```

prints “seven hundred sixty-five”, instead of “seven hundred and sixty-five”. By default, the “and” is included.

The translation of the decimal point is also subject to variation (with “point”, “dot”, and “decimal” being the favorites). The named argument ‘decimal’ allows the programmer to how the decimal point should be rendered. Hence:

```
print(scalar p.number_to_words("666.124.64.101"
```

prints “six sixty-six, dot, one twenty-four, dot, sixty-four, dot, one zero one” By default, the decimal point is rendered as “point”.

`number_to_words` also handles the ordinal forms of numbers. So:

```
print(p.number_to_words('1st'))
print(p.number_to_words('3rd'))
print(p.number_to_words('202nd'))
print(p.number_to_words('1000000th'))
```

prints:

```
first
third
two hundred and twenty-second
one millionth
```

Two common idioms in this regard are:

```
print(p.number_to_words(ordinal(number)))
```

and:

```
print(p.ordinal(p.number_to_words(number)))
```

These are identical in effect, except when `number` contains a decimal:

```
number = 99.09
print(p.number_to_words(p.ordinal(number));
print(p.ordinal(p.number_to_words(number));
```

Use whichever you feel is most appropriate.

CONVERTING LISTS OF WORDS TO PHRASES

When creating a list of words, commas are used between adjacent items, except if the items contain commas, in which case semicolons are used. But if there are less than two items, the commas/semicolons are omitted entirely. The final item also has a conjunction (usually “and” or “or”) before it. And although it’s technically incorrect (and sometimes misleading), some people prefer to omit the comma before that final conjunction, even when there are more than two items.

That’s complicated enough to warrant its own method: `join()`. This method expects a tuple of words, possibly with one or more options. It returns a string that joins the list together in the normal English usage. For example:

```
print("You chose ", p.join(selected_items))
# You chose barley soup, roast beef, and Yorksh

print("You chose ", p.join(selected_items, fina
# You chose barley soup, roast beef and Yorkshi

print("Please chose ", p.join(side_orders, conj
# Please chose salad, vegetables, or ice-cream
```

The available options are:

```
Option named    Specifies                Defaul

conj            Final conjunction        "and"
```

```
sep             Inter-item separator     ","
last_sep        Final separator          value
sep_spaced      Space follows sep        True
conj_spaced     Spaces around conj       True
```

INTERPOLATING INFLECTIONS IN STRINGS

By far the commonest use of the inflection methods is to produce message strings for various purposes. For example:

```
print(p.num(errors), p.plural_noun(" error"), p
if severity > 1:
    print(p.plural_adj("This"), p.plural_noun("
```

Unfortunately the need to separate each method call detracts significantly from the readability of the resulting code. To ameliorate this problem, inflect.py provides a string-interpolating method (`inflect(txt)`), which recognizes calls to the various inflection methods within a string and interpolates them appropriately.

Using `inflect` the previous example could be rewritten:

```
print(p.inflect("num({0}) plural_noun(error) pl
if severity > 1:
    print(p.inflect("plural_adj(This) plural_no
```

Note that `inflect` also correctly handles calls to the `num()` method (whether interpolated or antecedent). The `inflect()` method has a related extra feature, in that it *automatically* cancels any “default number” value before it returns its interpolated string. This means that calls to `num()` which are embedded in an `inflect()`-interpolated

string do not “escape” and interfere with subsequent inflections.

MODERN VS CLASSICAL INFLECTIONS

Certain words, mainly of Latin or Ancient Greek origin, can form plurals either using the standard English “-s” suffix, or with their original Latin or Greek inflections. For example:

```
p.plural("stigma")             # -> "stigmas" or
p.plural("torus")              # -> "toruses" or
p.plural("index")              # -> "indexes" or
p.plural("millennium")         # -> "millenniums
p.plural("ganglion")           # -> "ganglions"
p.plural("octopus")            # -> "octopuses"
```

inflect.py caters to such words by providing an “alternate state” of inflection known as “classical mode”. By default, words are inflected using their contemporary English plurals, but if classical mode is invoked, the more traditional plural forms are returned instead.

The method `classical()` controls this feature. If `classical()` is called with no arguments, it unconditionally invokes classical mode. If it is called with a single argument, it turns all classical inflects on or off (depending on whether the argument is true or false). If called with two or more arguments, those arguments specify which aspects of classical behaviour are to be used.

Thus:

```
p.classical()                 # SWITCH ON CLASSI
print(p.plural("formula")         # -> "formulae
```

```
p.classical(all=False)                # SWITCH O
print(p.plural("formula")         # -> "formulas

p.classical(cmode=True)             # CLASSICAL M
print(p.plural("formula")         # -> "formulae
                              # -> "formulas" (O

p.classical(herd=True)           # SWITCH ON CLA
print(p.plural("wilderbeest")     # -> "wilderbe

p.classical(names=True)          # SWITCH ON CLA
print(p.plural("sally")           # -> "sallies"
print(p.plural("Sally")           # -> "Sallys")
```

Note however that `classical()` has no effect on the inflection of words which are now fully assimilated. Hence:

```
p.plural("forum")               # ALWAYS -> "foru
p.plural("criterion")           # ALWAYS -> "crit
```

LEI assumes that a capitalized word is a person's name. So it forms the plural according to the rules for names (which is that you don't inflect, you just add -s or -es). You can choose to turn that behaviour off (it's on by the default, even when the module isn't in classical mode) by calling `` classical (names=0) ``

USER-DEFINED INFLECTIONS

Adding plurals at run-time

inflect.py provides five methods which allow the programmer to override the module's behaviour for specific cases:

`defnoun(singular, plural)`

> *The `defnoun` method takes a pair of string arguments: the singular and the plural forms of the noun being specified. The singular form specifies a pattern to be interpolated (as `m/^(?:$first_arg)$/i`). Any noun matching this pattern is then replaced by the string in the second argument. The second argument specifies a string which is interpolated after the match succeeds, and is then used as the plural form. For example:*
>
> ```
> defnoun( 'cow'        , 'kine')
> defnoun( '(.+i)o'     , '$1i')
> defnoun( 'spam(mer)?' , '||$||%||@#||$||@#!
> ```
>
> *Note that both arguments should usually be specified in single quotes, so that they are not interpolated when they are specified, but later (when words are compared to them). As indicated by the last example, care also needs to be taken with certain characters in the second argument, to ensure that they are not unintentionally interpolated during comparison.*
>
> *The second argument string may also specify a second variant of the plural form, to be used when "classical" plurals have been requested. The beginning of the second variant is marked by a '|' character:*
>
> ```
> defnoun( 'cow'        , 'cows|kine')
> defnoun( '(.+i)o'     , '$1os|$1i')
> defnoun( 'spam(mer)?' , '||$||%||@#||$||@#!
> ```
>
> *If no classical variant is given, the specified plural form is used in both normal and "classical" modes.*

`defverb(s1, p1, s2, p2, s3, p3)`

> *The `defverb` method takes three pairs of string arguments (that is, six arguments in total), specifying the singular and plural forms of the three "persons" of verb. As with `defnoun`, the singular forms are specifications of run-time-interpolated patterns, whilst the plural forms are specifications of (up to two) run-time-interpolated strings:*
>
> ```
> defverb('am'       , 'are',
>         'are'      , 'are|art",
>         'is'       , 'are')
>
> defverb('have'     , 'have',
>         'have'     , 'have",
>         'ha(s|th)' , 'have')
> ```
>
> *Note that as with `defnoun`, modern/classical variants of plurals may be separately specified, subsequent definitions replace previous ones, and `None`'ed plural forms revert to the standard behaviour.*

`defadj(singular, plural)`

> *The `defadj` method takes a pair of string arguments, which specify the singular and plural forms of the adjective being defined. As with `defnoun` and `defadj`, the singular forms are specifications of run-time-interpolated patterns, whilst the plural forms are specifications of (up to two) run-time-interpolated strings:*

```
defadj( 'this'     , 'these')
defadj( 'red'      , 'red|gules')
```

As previously, modern/classical variants of plurals may be separately specified, subsequent definitions replace previous ones, and `None`'ed plural forms revert to the standard behaviour.

`defa(pattern)` and `defan(pattern)`

The `defa` and `defan` methods each take a single argument, which specifies a pattern. If a word passed to `a()` or `an()` matches this pattern, it will be prefixed (unconditionally) with the corresponding indefinite article. For example:

```
defa( 'error')
defa( 'in.+')

defan('mistake')
defan('error')
```

As with the other `def_...` methods, such redefinitions are sequential in effect so that, after the above example, "error" will be inflected with "an".

The `<$HOME/.inflectrc` file

THIS HAS NOT BEEN IMPLEMENTED IN THE PYTHON VERSION YET

When it is imported, inflect.py executes (as Perl code) the contents of any file named `.inflectrc` which it finds in the in the directory where `Lingua/EN/Inflect.pm` is

installed, or in the current home directory (`$ENV{HOME}`), or in both. Note that the code is executed within the inflect.py namespace.

Hence the user or the local Perl guru can make appropriate calls to `defnoun`, `defverb`, etc. in one of these `.inflectrc` files, to permanently and universally modify the behaviour of the module. For example

> *> cat /usr/local/lib/perl5/Text/Inflect/.inflectrc*
>
> *defnoun "UNIX" => "UN*X|UNICES"*
>
> ***defverb "teco" => "teco", # LITERALLY: "to edit with TECO"***
>
> *"teco" => "teco", "tecos" => "teco"*
>
> *defa "Euler.*"; # "Yewler" TURNS IN HIS GRAVE*

Note that calls to the `def_...` methods from within a program will take precedence over the contents of the home directory F<.inflectrc> file, which in turn takes precedence over the system-wide F<.inflectrc> file.

DIAGNOSTICS

THIS HAS NOT BEEN IMPLEMENTED IN THE PYTHON VERSION YET

On loading, if the Perl code in a `.inflectrc` file is invalid (syntactically or otherwise), an appropriate fatal error is issued. A common problem is not ending the file with something that evaluates to true (as the five `def_...` methods do).

Using the five `def_...` methods directly in a program may also result in fatal diagnostics, if a (singular) pattern or an interpolated (plural) string is somehow invalid.

Specific diagnostics related to user-defined inflections are:

`"Bad user-defined singular pattern:\t %s"`

> *The singular form of a user-defined noun or verb (as defined by a call to `defnoun`, `defverb`, `defadj`, `defa` or `defan`) is not a valid Perl regular expression. The actual Perl error message is also given.*

`"Bad user-defined plural string: '%s'"`

> *The plural form(s) of a user-defined noun or verb (as defined by a call to `defnoun`, `defverb` or `defadj`) is not a valid Perl interpolated string (usually because it interpolates some undefined variable).*

`"Bad .inflectrc file (%s): %s"`

> *Some other problem occurred in loading the named local or global F<.inflectrc> file. The Perl error message (including the line number) is also given.*

There are *no* diagnosable run-time error conditions for the actual inflection methods, except `number_to_words` and hence no run-time diagnostics. If the inflection methods are unable to form a plural via a user-definition or an inbuilt rule, they just "guess" the commonest English inflection: adding "-s" for nouns, removing "-s" for verbs, and no inflection for adjectives.

`inflect.py` can raise the following execeptions:

`BadChunkingOptionError`

> *The optional argument to `number_to_words()` wasn't 1, 2 or 3.*

`NumOutOfRangeError`

> *`number_to_words()` was passed a number larger than 999,999,999,999,999,999,999,999,999,999,999,999 (that is: nine hundred and ninety-nine decillion, nine hundred and ninety-nine nonillion, nine hundred and ninety-nine octillion, nine hundred and ninety-nine septillion, nine hundred and ninety-nine sextillion, nine hundred and ninety-nine quintillion, nine hundred and ninety-nine quadrillion, nine hundred and ninety-nine trillion, nine hundred and ninety-nine billion, nine hundred and ninety-nine million, nine hundred and ninety-nine thousand, nine hundred and ninety-nine :-)*
>
> *The problem is that `number_to_words` doesn't know any words for number components bigger than "decillion".*

`UnknownClassicalModeError`

`BadNumValueError`

`BadUserDefinedPatternError`

`BadRcFileError`

OTHER ISSUES

2nd Person precedence

If a verb has identical 1st and 2nd person singular forms, but different 1st and 2nd person plural forms, then when its

plural is constructed, the 2nd person plural form is always preferred.

The author is not currently aware of any such verbs in English, but is not quite arrogant enough to assume *ipso facto* that none exist.

Nominative precedence

The singular pronoun “it” presents a special problem because its plural form can vary, depending on its “case”. For example:

```
It ate my homework       ->  They ate my homewo
It ate it                ->  They ate them
I fed my homework to it  ->  I fed my homework
```

As a consequence of this ambiguity, `plural()` or `plural_noun` have been implemented so that they always return the *nominative* plural (that is, “they”).

However, when asked for the plural of an unambiguously *accusative* “it” (namely, `plural("to it")`, `plural_noun ("from it")`, `plural("with it")`, etc.), both methods will correctly return the accusative plural (“to them”, “from them”, “with them”, etc.)

The plurality of zero

The rules governing the choice between:

```
There were no errors.
```

and

```
There was no error.
```

are complex and often depend more on *intent* rather than *content*. Hence it is infeasible to specify such rules algorithmically.

Therefore, inflect.py contents itself with the following compromise: If the governing number is zero, inflections always return the plural form unless the appropriate "classical" inflection is in effect, in which case the singular form is always returned.

Thus, the sequence:

```
p.num(0)
print(p.inflect("There plural(was) no(choice)")
```

produces "There were no choices", whereas:

```
p.classical(zero=True)
p.num(0)
print(p.inflect("There plural(was) no(choice)")
```

it will print("There was no choice".)

Homographs with heterogeneous plurals

Another context in which intent (and not content) sometimes determines plurality is where two distinct meanings of a word require different plurals. For example:

```
Three basses were stolen from the band's equipm
Three bass were stolen from the band's aquarium

I put the mice next to the cheese.
I put the mouses next to the computers.

Several thoughts about leaving crossed my mind.
Several thought about leaving across my lawn.
```

inflect.py handles such words in two ways:

- If both meanings of the word are the *same* part of speech (for example, “bass” is a noun in both sentences above), then one meaning is chosen as the “usual” meaning, and only that meaning’s plural is ever returned by any of the inflection methods.
- If each meaning of the word is a different part of speech (for example, “thought” is both a noun and a verb), then the noun’s plural is returned by `plural()` and `plural_noun()` and the verb’s plural is returned only by `plural_verb()`.

Such contexts are, fortunately, uncommon (particularly “same-part-of-speech” examples). An informal study of nearly 600 “difficult plurals” indicates that `plural()` can be relied upon to “get it right” about 98% of the time (although, of course, ichthyophilic guitarists or cyber-behaviouralists may experience higher rates of confusion).

If the choice of a particular “usual inflection” is considered inappropriate, it can always be reversed with a preliminary call to the corresponding `def_...` method.

NOTE

There will be no further correspondence on:

“octopi”.

> *Despite the populist pandering of certain New World dictionaries, the plural is “octopuses” or (for the pendantic classicist) “octopodes”. The suffix “-pus” is Greek, not Latin, so the plural is “-podes”, not “pi”.*

“virus”.

> *Had no plural in Latin (possibly because it was a mass noun). The only plural is the Anglicized “viruses”.*

AUTHORS

Thorben Krüger (github@benthor.name) * established Python 3 compatibility

Paul Dyson (pwdyson@yahoo.com) * converted code from Perl to Python * added singular_noun functionality

Original Perl version of the code and documentation: Damian Conway (damian@conway.org), Matthew Persico (ORD inflection)

BUGS AND IRRITATIONS

The endless inconsistencies of English.

(*Please* report words for which the correct plural or indefinite article is not formed, so that the reliability of inflect.py can be improved.)

COPYRIGHT

Get help

Installing packages
Uploading packages
User guide
FAQs

About PyPI

Status: All Systems Operational
Package index name retention
Our sponsors

Contributing to PyPI

Bugs & feedback
Contribute on GitHub
Development credits

Using PyPI

Code of conduct
Security policy
Privacy policy
Terms of use

Developed and maintained by the Python community, for the Python community.
Donate today!

Desktop version

Elastic
Search

Pingdom
Monitoring

Google
BigQuery

Sentry
Error logging

CloudAMQP
RabbitMQ

AWS
Cloud computing

Fastly
CDN

DigiCert
EV certificate

StatusPage
Status page