# pyslate Documentation

***Release 1.1***

**Aleksander**

January 28, 2016

Contents

CHAPTER 1

# User guide (for translators)

Managing text translations from the perspective of the translator.

## 1.1 Introduction

In this article you'll learn how to translate (internationalize&localize) the messages as a translator - when you have no idea about their programmatic context.

All these examples will be posted in the same form: a box with all tags in English, a box with all tags in Pirate English. Then there will be some examples of using these tags in the real-life context.

**Note:** Such examples are something you usually don't have access to as a translator.

First I'll explain what tag is. Tag is a pair of text strings: a key and a value.

- **Value** is the text visible for users of the program - the part you have to translate.
- **Key** is text's identifier invisible for users - you should keep it the same, because translating is basically supplying a different tag value for the same key.

All examples will use the simple syntax: `TAG_KEY => TAG_VALUE`

## 1.2 Hello message

As I cannot assume you know any language besides English, we'll start with translating English to Pirate English.

**English**

```
hello => Hello guys
```

It's easy. You read the value of the "hello" tag and provide a translation.

**Pirate English**

```
hello => Ahoy comrades!
```

This tag value is exactly what users will see, so there's no need to show any examples.

## 1.3 Interpolated variables

Sometimes it's necessary to mention in the message something specified from outside. Just think about a message dialog 'Do you want to remove a file "mypicture.png"?' This file name is something that does change, so there must be a way for a programmer to deliver it and there needs to be a way for a translator to show it as part of the message. That's why there exists a special structure, called a **variable field**. It's denoted as **%{variable_name}** which is later replaced by Pyslate during the program execution. The "variable_name" is identifier of value which should be interpolated here. The only easy way to learn what can stand for "variable_name" is to thoroughly read the original translation.

**English**

```
file_removal => Do you want to remove a file "%{file_name}"?
```

So now we know that name of the file is specified using the identifier "file_name". So all we need to do is translate it in the same manner.

**Pirate English**

```
file_removal => Do ye want to scuttle a file "%{file_name}"?
```

Example is easy to guess...

---

**English Example**

Do you want to remove a file "ship.png"?

---

---

**Pirate English Example**

Do ye want to scuttle a file "ship.png"?

---

## 1.4 Interpolated variables - numbers

The values interpolated into the variable fields can also be the numbers.

**English**

```
rum_barrel   => I posess a barrel of the finest rum.
rum_barrel#p => I posess %{number} barrels of the finest rum.
```

What's that? When the programmer calls a variable identifier "number" then some magic happens. As you see there are two forms of the same tag. "rum_barrel" is called a base tag, while "#p" is called a tag variant (because it's a variant of the base tag). Then, depending on the value of the **%{number}**, a different version of a tag can be selected. In English it's "rum_barrel" (singular) when **%{number}** is 1, and "rum_barrel#p" (**p**lural) when **number** is not 1. There are just two forms, but some languages have much more. Let's assume our Pirate English has a different form of noun when **%{number}** is 2, so "-es" is then appended instead of "-s" to the end of the noun. We assume the programmer already took care of specifying pluralization rules for our language, so all we have to do is learning what

letter is used when the **%{number}** is 2. After a quick look into the cheatsheet (TODO LINK) we learn that in such situation we should add "#t" variant (**t**wo) to make it work. Okay, here we go.

**English**

```
rum_barrel   => I'ave a barrel o' best rum.
rum_barrel#t => I'ave %{number} barreles o' best rum.
rum_barrel#p => I'ave %{number} barrels o' best rum.
```

Now some examples:

---

**English Example**

I posess a barrel of the finest rum.
I posess 2 barrels of the finest rum.
I posess 5 barrels of the finest rum.
I posess 17 barrels of the finest rum.

---

---

**Pirate English Example**

I'ave a barrel o' best rum.
I'ave 2 barreles o' best rum.
I'ave 5 barrels o' best rum.
I'ave 17 barrels o' best rum.

---

Curious what language has a different pluralization when there are exactly two items? It's the case for Arabic and many others. We are prepared for that.

## 1.5 Fallbacks in Pyslate

Pyslate has a powerful fallback mechanism. It means if something is not available in the expected form/language, then Pyslate is selecting the best alternative.

**Tag variant fallback**

Every tag key is composed of base and variant: e.g. *sweet_cookie*#**p**. In case the expected tag with variant doesn't exist, then its base tag is used: sweet_cookie#p -> sweet_cookie

It should always be guaranteed that a base tag exists if any variant tag with the same base exists. If you have a tag with variant consisting of many variant letters then matching is done from the most to least exact:

```
. sweet_cookie#png -> sweet_cookie#pn -> sweet_cookie#p -> sweet_cookie
```

It's useful especially for fluent languages, where form of the word depends on the context.

**Language fallback**

Pyslate supports incremental translations, so the system can be used before all the translations are completed. If there's no matching tag in the target language, then the whole procedure (described above) is run again for the fallback language. E.g. when the fallback language for Portuguese is Spanish:

```
(pt)sweet_cookie#p -> (pt)sweet_cookie -> (es)sweet_cookie#p -> (es)sweet_cookie
```

If there's no tag for target language or its fallback language, then its global fallback is used in the same manner (usually it means English).

## 1.6 Switch fields - different forms of the same text

Now it's time for another special structure called a **switch field**. It's denoted '%{identifier:option1?answer1|option2?answer2}' which means "if value for 'identifier' is like 'option1' then show 'answer1', if 'identifier' is like 'option2' then use 'answer2'. If none of these, then use the first answer from the left - 'answer1' in this case". 'identifier' is name of some variable, very similar to 'variable_name' or 'number' from the previous examples.

**English**

```
sabre_statement => I have a sabre, %{state:sharp?a finely sharped one|blunt?which is going to be shar
```

Okay, so we shouldn't translate the identifier or its options ("state", "sharp", "blunt"), as we have no control over these. But we can translate answers, which are visible for users.

**Pirate English**

```
sabre_statement => Arr! I'ave a saber, %{state:sharp?a well sharp'd one|blunt?which be goin' to be sh
```

---

**English Example**

I have a sabre, a finely sharped one.
I have a sabre, which is going to be sharpened soon.

---

---

**Pirate English Example**

Arr! I'ave a saber, a well sharp'd one.
Arr! I'ave a saber, which be goin' to be sharp'd before I sail out.

---

If you see above, I wrote "if 'identifier' is like 'option1'", because LIKE doesn't mean the same as "equals to". In fact it means "if 'option1' is part of 'identifier' string", but it doesn't matter in this particular example and will be further explained.

## 1.7 Inner tag fields

Now it's time for the last special structure available - an **inner tag field**. In short, it allows you to show any other tag on any position in the text. It's denoted **${tag_name}**, where tag_name is any of existing tag keys.

**English**

```
eat_breakfast   => I was eating breakfast. ${was_good}.
eat_supper:     => I was eating supper. ${was_good}.
was_good:       => It was really good.
```

It's quite easy. We translate, but don't touch stuff inside of ${}. It's a quite simple example menat to just have a bit less to copy&paste (even though we are pirates), but there happen complicated situations where using this structure is unavoidable.

**Pirate English**

```
eat_breakfast   => I was eatin' breakfast. ${was_good}.
eat_supper      => I was eatin' supper. ${was_good}.
was_good:       => 'twas really jolly.
```

---

**English Example**

I was eating breakfast. It was really good.
I was eating supper. It was really good.

---

**Pirate English Example**

I was eatin' breakfast. 'twas really jolly.
I was eatin' supper. 'twas really jolly.

---

## 1.8 Variable tag field in inner tag field

We need to go deeper.

**English**

```
look_at:        => Hey! Look at ${state_%{item}}.
state_sabre:    => a sharp sabre
state_gun:      => a shiny pistol
```

Oh, look, a **variable field** inside of **inner tag field**. It means **variable field** is evaluated first, which produces *some* text (e.g. "ABC"), which is merged with `"state_"`, which created a name of the inner tag (e.g. "state_ABC"), which is then looked for on the list of tag keys. Quite complicated, but is it a problem for a translator like you? **%{item}** can potentially hold any value you can think, but it's possible to guess that the only possible values are de facto "sabre" and "gun", because we see that inner tag must start with `"state_"`. We can assume it always produce the valid (existing) tags. There cannot be any other in our Pirate language if there aren't such in original language.

**Pirate English**

```
look_at:        => Ahoy! Look at ${state_%{item}}.
state_sabre:    => a sharp saber
state_gun:      => a nice firearm
```

---

**English Example**

Hey! Look at a sharp sabre.
Hey! Look at a shiny pistol.

---

---

**Pirate English Example**

Ahoy! Look at a sharp saber.
Ahoy! Look at a nice firearm.

---

Another success, so now something what our Pirate English will not cope with.

## 1.9 Switch field and inner tag field cooperation

The already presented features are enough for our Pirate English example, but unfortunately Pirate English looks quite similar to English. All the difference is changing a few words, but there are real languages which are much different. I'm speaking about fusional languages. If you are not working with them, then you don't have to read further, but maybe you'll find it interesting. The following example will be much more complicated, but I hope it'll be explained precisely. In Polish (and Russian, German... and many others) every noun has a grammatical form (gender). Let's see: "szabla" (a sabre) is feminine (f), while "pistolet" (a pistol) is masculine (m). This grammatical form is very important to set the correct suffix for adjectives describing the noun. Let's see an example:

This is a new pistol. => To jest now**y** pistolet.
This is a new sabre. => To jest now**a** szabla.

"To jest" ("This is") part is the same for both items, but the suffix appended to the stem "now-" is based on the gender of the noun:
"m" => "-y"
"f" => "-a"
"n" => "-e"

**English**

```
presentation_text:  => This is a new ${item_%{item_name}}.
item_sabre:         => sabre
item_pistol:        => pistol
```

I hope this part is quite easy. Using the same deduction as in the previous example we know that item_name can be only "sabre" or "pistol". Now we need to prepare a translation for Polish. We start with translating the items. It's possible to specify the grammatical form for every tag, so we do it here:

**Polish**

```
item_sabre: => szabla
         form: f
item_pistol: pistolet
         form: m
```

Okay, we have translated items, but there's the toughest part. At the first glance it should be something like: presentation_text: To jest now%{**SOMETHING**:m?y|f?a|n?e} ${item_%{item_name}}.

What to set into **SOMETHING**? How can we guess what item is it? Should we ask a programmer to create a special variable for us? It's a very bad idea, because it would significantly complicate the translation process. That's why there's a special way in which inner tag fields can cooperate with switch fields.

**Polish**

```
presentation_text:  => To jest now%{obj_g:m?y|f?a|n?e} ${obj_g:item_%{item_name}}.
```

That's right. We have specified an identifier for an inner tag (*obj_g*), which is then used as an identifier of a variable which is used in a switch field. The inner tag's identifier gets the grammatical form contained in an inner tag. It is then transported to the switch field which makes the correct decision.

So the full Polish translation looks like that:

**Polish**

```
presentation_text:  => To jest now%{obj_g:m?y|f?a|n?e} ${obj_g:item_%{item_name}}.
item_sabre:            => szabla
                  form: f
item_pistol:           => pistolet
                  form: m
```

If you are translating to a fusional language then I hope you've learned how does it work. If you don't know any of such, then these examples can be hard to understand.

## 1.10 Appendix I - correct variant letters for numbers and cases

As it was already mentioned, variants are specified by single-letter identifiers. Every letter has some contractual meaning and specific letters are not imposed by Pyslate (with exception of pluralization letters, which are based on language locale).

**Letter reserved to never be used to work as default value:**

- x

Letters that are reserved to be used for pluralization forms:

- "" (empty) - singular - base form
- z - **z**ero - when there are no elements

- t - **T**wo - plural form for 2 or numbers treated like 2.
- w - fe**W** - form used for *a few* elements (usually 3, 4) or treated like *a few*
- p - **p**lural (a.k.a. many) - form used for all the rest

They are unused for most of languages.

Suggestion what letters should be used for the following gender forms:

- m - masculine
- f - feminine
- n - neuter

There's suggestion what letters should be used for the following (latin) cases in fusional languages:

- "" (empty) - nominative - base form
- g - genitive
- d - dative
- a - accusative
- b - ablative
- l - locative
- v - vocative

It's worthless to try to supply all the forms, even if the language supports them. Use only those really needed in the translation system. If language you are translating to supports more than that - you can use any of "unused" letters. It's just advised to avoid using "x".

If variant tag contains all these data, then letters in a variant are advised to be used in the following order: plural form, gender form, case. For example: small_stone#pmg (plural, masculine, genitive). This order guarantees the fallback process most effective.

CHAPTER 2

# Pyslate syntax reference

## 2.1 Decorators

Decorators are constructs applicable to tags or variable fields to modify their value. They are python functions which take tag value string as input and return output string. They are added after the end of tag key and are prefixed by "@". They are left-associative e.g. "**some_tag@article@capitalize**" means first adding an article, then capitalizing the first letter.

```
{
    "buying_toy": {
        "en":  "I've bought ${toy_%{name}@article}."
    },
    "toy_rocking_horse": {
        "en": "rocking horse"
    },
    "toy_autosan": {
        "en": "autosan"
    }
}

>>> pyslate_en.t("buying_toy", name="rocking_horse")
I've bought a rocking horse.
>>> pyslate_en.t("buying_toy", name="autosan")
I've bought an autosan.
```

Apart from built-in decorators it's possible to define custom ones.

```
{
    "some_message": {
        "en":  "Important message"
    },
    message_container": {
        "en":  "Message is: ${some_message@add_dots}"
    }
}

def add_dots(value):
    return ".".join(value)

pyslate_en.register_decorator("add_dots", add_dots)

>>> pyslate_en.t("message_container")
Message is: I.m.p.o.r.t.a.n.t. .m.e.s.s.a.g.e
```

```
>>> pyslate_en.t("message_container@add_dots")
M.e.s.s.a.g.e. .i.s.:. .I...m...p...o...r...t...a...n...t... ...m...e...s...s...a...g...e
```

It's possible to decorate both requested tag, inner tag fields and variable fields. In the last command, value of "some_message" tag gets dots added and then the whole texts gets dots added. There are three dots between the letters, because second decorator adds dots between every single character, including dots added by first decorator.

### 2.1.1 Available decorators

By default Pyslate provides the following decorators in the default scope:

- `capitalize` - make the first character have upper case and the rest lower case
- `upper` - convert to uppercase
- `lower` - convert to lowercase

For English an additional decorator is available:

- `article` - add *a* or *an* article to a word. *an* is added if the first letter of the word is a vowel, *a* otherwise.

## 2.2 Custom functions

Custom functions allow you to do almost unlimited manipulation over the produced text. They are kind of magic tag, which, when referenced, executes python code to produce the correct tag value on-the-fly based on input arguments.

Every custom function needs to take 3 parameters:

- `helper` - a class being a wrapper for Pyslate object that allows you to perform certain operations, instance of *pyslate.PyslateHelper*
- `name` - full name of requested tag (including its variant)
- `params` - dict of all parameters (names and values of variable fields) specified for the tag

```
{
    "some_tag": {
        "en":  "%{person:m?He|f?She} is ${person:some_custom}."
    },
    "template_person": {
        "en": "%{firstname} %{lastname} (SSN: %{ssn})"
    }
}

def some_custom_function(helper, name, params):
    people_database = {
        9203139: {"firstname": "John", "lastname": "Johnson", "sex": "m"},
        9203312: {"firstname": "Judy", "lastname": "Brown", "sex": "f"},
        9493839: {"firstname": "Edward", "lastname": "Smith", "sex": "m"},
    }

    ssn_number = params["ssn"]
    person = people_database[ssn_number]

    helper.return_form(person["sex"])  # it's to make tag's grammar form available outside

    return helper.translation("template_person", firstname=person["firstname"], lastname=person["last
```

Let's register this function:

```
>>> pyslate_en.register_function("some_custom", some_custom_function)
>>> pyslate_en.t("some_tag", ssn=9203312)
She is Judy Brown (SSN: 9203312).
```

A few things to notice: it's possible to set grammatical form for the text generated by a custom function. It would look like "template_person" tag is unnecessary and it would be ok to replace the last line in the some_custom_function to:

```
return "{} {} (SSN: {})".format(person["firstname"], person["lastname"], ssn_number)
```

It would work, but it's not a good idea. We want to make text fully internationalizable, while in some countries the opposite order of first and lastname is used. It should be possible to format such texts for every language to look natural. Translator usually doesn't have knowledge nor ability to change python code, so it's better to keep text format in a separate tag.

'params' argument contains a full dict of key-value pairs consisting of: explicit parameters, context parameters and special parameters (e.g. *tag_v*), so they are all available in the function body.

Let's see it in another example, where func_print_all displays full name of called pseudo-tag and all its parameters:

```
def func_print_all(helper, name, params):
    return name + " | " + str(params)

>>> pyslate_en = Pyslate("en", backend=JsonBackend("translations.json"))
>>> pyslate_en.register_function("print_all", func_print_all)
>>> pyslate_en.context = {"name": "John", "age": 18}

>>> pyslate_en.t("print_all#f", arg1=True, arg2="help me")
"print_all#f | {'arg1': True, 'age': 18, 'tag_v': 'f', 'name': 'John', 'arg2': 'help me'}"
```

CHAPTER 3

# Pyslate API reference

## 3.1 Config variables

**class** pyslate.config.**DefaultConfig**

Default values for configuration options of Pyslate.

If you want to overwrite defaults, create a subclass of DefaultConfig and overwrite interesting values in your class' constructor. You can also create a function which instantiates DefaultConfig, overwrites some values and then gives it to Pyslate's constructor as keyword-argument "config". Please note you can use keyword-arguments of Pyslate constructor to specify own parser, cache and backend. Keyword arguments set in Pyslate constructor have higher precedence than values from the config.

## 3.2 pyslate module

**class** pyslate.pyslate.**Pyslate**(*language*, *backend=None*, *config=<pyslate.config.DefaultConfig object>*, *context=None*, *cache=None*, *locales=None*, *parser=None*, *on_missing_tag_key_callback=None*)

Main class responsible for all the translation and localization. When constructed it's necessary to set language, backend. It's possible to set custom config, context dict and instance of cache class.

**l**(*value*, *short=False*)

Method returning localized string representation of a value specified in the argument. Currently it guarantees to correctly localize the following types: float, datetime.date, datetime.datetime, datetime.time If value cannot be localized then its string representation is returned.

**Parameters** **value** – value to be localized

**Returns** string representation of the value, localized if being instance of the supported types

**localize**(*value*, *short=False*)

Method returning localized string representation of a value specified in the argument. Currently it guarantees to correctly localize the following types: float, datetime.date, datetime.datetime, datetime.time If value cannot be localized then its string representation is returned.

**Parameters** **value** – value to be localized

**Returns** string representation of the value, localized if being instance of the supported types

**register_decorator**(*decorator_name*, *function*, *is_deterministic=False*, *language=None*)

Registers a new decorator which will be available in the translation system. Overwrites any other decorator or function with the same name.

**Parameters**

- **decorator_name** – name of the decorator available in the translation system
- **function** – a function whose only argument is input string string and returns an altered string
- **is_deterministic** – if True then return value of the decorator for specified arguments will be cached to be reused in the future. Keep it disabled unless you really know you need it.
- **language** – language for which decorator will be available, If unspecified then it's available for all languages

**register_function**(*tag_name*, *function*, *is_deterministic=False*, *language=None*)

Registers a new custom function which will be available in the translation system. Overwrites any other decorator or function with the same name.

**Parameters**

- **tag_name** – name base tag key for which the function is accessible in an inner tag field. See the examples.
- **function** – function with 3 arguments: - a helper (instance of PyslateHelper), which is a facade for translating specified tags or setting grammatical form of the custom function - tag_name - params (keyword arguments specified in Pyslate.translate)
- **is_deterministic** – if True then return value and grammatical form of the function for specified arguments will be cached to be reused in the future. Keep it disabled unless you really know you need it.
- **language** – language for which function will be available. If unspecified then it's available for all languages

**t**(*tag_name*, ***kwargs*)

Method returning fully translated tag value for a specified tag_name using kwargs as a list of keyword arguments. If there's no tag value for specified language, then tag value for fallback language (or global fallback language) is used.

**Parameters**

- **tag_name** – tag key which should be translated. It can contain decorators
- **kwargs** – arguments which can be interpolated into tag value

**Returns** translated value for specified tag key.

**translate**(*tag_name*, ***kwargs*)

Method returning fully translated tag value for a specified tag_name using kwargs as a list of keyword arguments. If there's no tag value for specified language, then tag value for fallback language (or global fallback language) is used.

**Parameters**

- **tag_name** – tag key which should be translated. It can contain decorators
- **kwargs** – arguments which can be interpolated into tag value

**Returns** translated value for specified tag key.

**class** pyslate.pyslate.**PyslateHelper**(*pyslate*)

Class given as a first argument of the custom functions. It's a facade which allows for translating or getting a grammatical form for specified tag keys. It also allows for setting a grammatical form of an entity represented

by this custom function. This way a custom function can be a black-box treated exactly the same as a normal inner tag field.

**form**(*tag_name*, ***kwargs*)
Returns grammatical form of the tag_name tag (which can be None).

**get_suffix**(*tag_name*)

**pass_the_suffix**(*tag_name*)

**return_form**(*form*)
Specifies grammatical form of the entity represented by the custom function. It can later be retrieved by other fields of the tag value.

**Parameters** **form** – grammatical form of this custom function

**translation**(*tag_name*, ***kwargs*)
Returns a translated string for specified tag_name and kwargs. Delegates to Pyslate.translate method.

**translation_and_form**(*tag_name*, ***kwargs*)
If you need both translation and grammatical form, then it's more efficient to use it to get both at once. Returns a tuple whose first element is a translated string for specified tag_name and kwargs. The second element is grammatical form of the tag (which can be None).

CHAPTER 4

# Advanced examples

This page contains advanced examples formatted similarly to the basic examples available in the README: code, tags in multiple languages and result. So it's not to present core functions of the library, but show general examples what things can be done using Pyslate. It's most usefeul for the programmers, but tag key-value pairs (in examples 1-3) contain valuable info for translators as well (all tags are presented in JSON format).

## 4.1 Example 1 - Advanced switch fields

It's possible to use multiple variables to control switch fields.

```
{
    "talking_the_same": {
        "en": "I told %{sb:m?him|f?her} it's stupid and %{sb:m?|f?s}he told me the same.",
        "pl": "Powiedział%{me:m?em|f?am} %{sb:m?mu|f?jej}, że to głupie, a on%{sb:m?|f?a} powiedział%
    }
}

>>> pyslate_en.t("talking_the_same", me="f", sb="f")
I told her it's stupid and she told me the same.
>>> pyslate_en.t("talking_the_same", me="m", sb="m"))
I told him it's stupid and he told me the same.

>>> pyslate_pl.t("talking_the_same", me="f", sb="f")
"Powiedziałam jej, że to głupie, a ona powiedziała mi to samo."
>>> pyslate_pl.t("talking_the_same", me="m", sb="m"))
"Powiedziałem mu, że to głupie, a on powiedział mi to samo."
```

## 4.2 Example 2 - Advanced tag variants

Somewhere in the README I've said that switch field matches the value which is equal to a specified variant.

It's not entirely true. It matches an option which is **contained in** switch variable's value.

It means letters representing specific variants can be joined into one string e.g. "pma" to have accusative masculine for plural.

It's advised to use exactly this order: NUMBER, GENDER, CASE. It is required to have NUMBER left-most letter if it exists.

```
{
    "show_the_way": {
        "en": "I have shown the way to the ${%{benefactor}}.",
        "pl": "Wskazałem drogę ${%{benefactor}#d} ."
    },
    "traveler": {
        "en": "traveler",
        "pl": "podróżnik"
    },
    "traveler#p": {
        "en": "travelers",
        "pl": "podróżnicy"
    },
    "traveler#pd": {
        "pl": "podróżnikom"
    },
    "driver": {
        "en": "driver",
        "pl": "kierowca"
    },
    "driver#p": {
        "en": "drivers",
        "pl": "kierowcy"
    },
    "cyclist": {
        "en": "cyclist",
        "pl": "cyklista"
    },
    "cyclist#d": {
        "pl": "cykliście"
    }
}

# first example - correct
>>> pyslate_en.t("show_the_way", number=5, benefactor="traveler")
I have shown the way to the travelers.
>>> pyslate_pl.t("show_the_way", number=5, benefactor="traveler")
Wskazałem drogę podróżnikom.

# second example - fallback to shorter variant: driver#pd -> driver#p
>>> pyslate_en.t("show_the_way", number=5, benefactor="driver")
I have shown the way to the drivers.
>>> pyslate_pl.t("show_the_way", number=5, benefactor="driver")
Wskazałem drogę kierowcy.

# third example - fallback to base form: cyclist#pd -> cyclist#p -> cyclist
>>> pyslate_en.t("show_the_way", number=5, benefactor="cyclist")
I have shown the way to the cyclist.
>>> pyslate_pl.t("show_the_way", number=5, benefactor="cyclist")
Wskazałem drogę cyklista.
```

In the first code example, it correctly guesses the number and case for both English and Polish.

In the second example it's ok in English, but there's no *driver#pd* variant for Polish, so it removes the right-most letter from variant and tries again. It finds the *driver#p* tag so it's used as a fallback.

In the third example, Polish and English have no plural form, so it's not used. In Polish *cyclist#d* variant is defined, but the fallback mechanism tries: *cyclist#pd*, then *cyclist#p* and *cyclist*. So both English and Polish fallback to base

form.

## 4.3 Example 3 - Complicated number format

First thing: We have both singular and plural forms in the single translation tag, even though it's less readable.
It's possible thanks to "tag_v" special variable, which always contains original tag variant, even if for example "item_mug#y" had to be fallbacked to "item_mug" because there was no specific variant tag.
Even more, it's possible to mix these two ways of representing variants, especially useful if target language is more complicated than the original one.

```
{
    "giving_thing": {
        "en": "I give you ${item_%{name}}",
        "pl": "Daję ci ${item_%{name}#a}",
    },
    "item_mug": {
        "en": "${number} mug%{tag_v:s?|p?s}",
        "pl": "${number} kub%{tag_v:s?ek|w?ki|p?ków}",
    },
    "item_cup": {
        "en": "${number} cup%{tag_v:s?|p?s}",
        "pl": "filiżank%{tag_v:x?a|a?ę}",
    }
    "item_cup#w": {
        "pl": "${number} filiżanki",
    },
    "item_cup#p": {
        "pl": "${number} filiżanek",
    }
}

>>> pyslate_en.t("giving_thing", number=1, name="cup")
I give you 1 cup.
>>> pyslate_en.t("giving_thing", number=5, name="cup")
I give you 5 cups.
>>> pyslate_pl.t("giving_thing", number=1, name="cup")
Daję ci filiżankę.
>>> pyslate_pl.t("giving_thing", number=5, name="cup")
Daję ci 5 filiżanek.
```

As you can see, for Polish you have to use a different case (accusative), but only for a singular form of a word "filiżanka" ("cup").
It's not necessary for a word "kubek" ("mug"), though.
tag value "filiżank%{tag_v:x?ała?ę}" contains
Another trick (which was already used somewhere else too) is having option "x?" in a switch field.
"x" variant is required to be never used, so it can never be matched with value of variable. But it's first left, so it is matched as default option when nothing else can be matched.
That's the case when you request the most basic form of a word (singular nominative form).

## 4.4 Example 4 - Setting your own config

You can alter the default configuration of Pyslate by creating subclass of `config.DefaultConfig` and passing instance as an *config* argument to constructor - `Pyslate.__init__()`.

**Warning:**
Do not create custom Config as an independent class with the same set of attributes. It can get broken when a config option is added in the new version of Pyslate.
The good way is to subclass DefaultConfig and overwrite some values in its constructor (just remember to call parent constructor)
It's also correct to create factory function instantiating DefaultConfig and monkey-patching attributes of DefaultConfig.

```
class MyConfig(DefaultConfig):
    def __init__(self):
        super().__init__()
        self.ON_MISSING_VARIABLE = lambda name: "Variable {} is missing".format(name)
        self.FALLBACKS = {"pl": "en",
                          "fr": "de"}

pyslate = Pyslate("en", config=MyConfig(), backend=JsonBackend("tags.json"))
```

## 4.5 Example 5 - Pyslate learning which tags are missing

Pyslate is easily customizable to meet your needs: instead of allowing to select one of a few options it's possible to supply your own callback function. For example it's possible to specify a callback which is fired when a tag is not found by the backend. It's controlled by `config.DefaultConfig.ON_MISSING_TAG_KEY` attribute in config or *on_missing_tag_key_callback* parameter in constructor - `pyslate.Pyslate.__init__()`.

This callback function takes two parameters: tag name and dict of variables what were possible to be interpolated into this tag. It returns string which is text shown instead of missing tag.

The default implementation of this function is very simple:

```
self.ON_MISSING_TAG_KEY = lambda name, params: "[MISSING TAG '{0}']".format(name)
```

So in case we ask for a tag that doesn't exist in the backend:

```
>>> pyslate = Pyslate("en", backend=JsonBackend("tags.json"))
>>> pyslate.t("some_tag")
[MISSING TAG 'some_tag']
```

That's nice. User of our program can see which tag is missing and report it to us, but it'd be better to happen automatically. It'd also be nice to remember what variables were interpolated into the tag to make it easier to create default (English) translation.

::

```
    >>> def on_missing_key(name, params):
    >>>     with open("missing_tags.txt", "w") as file:
    >>>         file.write("{0} - {1}\n".format(name, params.keys()))
    >>>     return "[MISSING TAG'{0}']".format(name)
    >>>
    >>> pyslate = Pyslate("en", backend=JsonBackend("tags.json"), on_missing_tag_key_callback=on_mis
```

```
>>> pyslate.t("some_tag", param1="hello", param2=23)
[MISSING TAG 'some_tag']
```

The file "missing_tags.txt" contains logged info about this tag:

```
some_tag - ['tag_v', 'param2', 'param1']
```

We see it logs two explicit and one implicit "tag_v" which is added to every tag value. So it's easy to add these tags to your backend.

CHAPTER 5

# What's new? Changelog

version 1.1 [2016-01-27]:

- result of custom functions will no longer be interpreted implicitly
- lexer bugfix for strings ending with "$" or "%"

version 1.0 [2015-12-18]:

- fully working code and documentation

CHAPTER 6

# Indices and tables

- genindex
- modindex
- search

## p

## D



## F



## G



## L



## P



## R



## T