

Table of Contents

Introduction	0
Key concepts	1
Tax and Benefit System	1.1
Variables	1.2
Parameters	1.3
Person, entities, role	1.4
Periods, Instants	1.5
Input Data	1.6
Simulation, Computation	1.7
Reforms	1.8
From law to code	2
Coding a formula: basic example	2.1
Introducing an input variable	2.2
Vectorial computing	2.3
Periods	2.4
Legislation evolutions	2.5
Entities	2.6
Reforms	2.7
Inferences	2.8
Writing YAML tests	2.9
Parameters	2.10
Bootstrapping a new country package	2.11
OpenFisca Web API	3
Endpoints	3.1
Input and output data	3.2
Troubleshooting	4
Recipes	5
Use OpenFisca on the web (no installation required on your computer)	5.1
Install OpenFisca in an offline environment	5.2
Work on OpenFisca on a Windows without being administrator	5.3
Test your changes on "ready to use" situations	5.4
Publishing results	6
Community	7
Contribute	8
Contributor guidelines	8.1
Language	8.1.1
Commit messages	8.1.2
Variables naming	8.1.3
Semantic versionning	8.1.4

Enhance the documentation	8.2
Extensions	8.3
Developer guide	8.4
Tests	8.5
Release process	8.6



Download a [PDF offline version](#) of this documentation.

Introduction

OpenFisca is a versatile [microsimulation](#) software. OpenFisca allows users to :

- Calculate many variables of the tax and benefit system of a country given input variables.
OpenFisca can calculate social benefits and taxes on test cases (a person or a household).
- Simulate the budgetary consequences of a reform and its distributional impact when plugged on a survey.
OpenFisca can calculate social benefits and taxes on population data (real data or survey data)

To achieve both, computations are vectorial and use [NumPy](#) package. It is coded in the C language under the hood, more performant than Python. Its engine is independent of the country, it is therefore possible to simulate any country. For now the main supported country is France.

OpenFisca is a free software published under the [GNU Affero General Public Licence](#) version 3 or later. It is written in the [Python](#) programming language (compatible with version 2.7).

Project Components

OpenFisca is a modular project. Depending on your goals, you will install and interact with one or several of the OpenFisca Components.



Web API

The Web API lets you access the legislation [Parameters](#) and [Variables](#).

Example: [Mes Aides](#) uses the OpenFisca Web API to calculate OpenFisca-France benefits.

- To explore the OpenFisca-France Web API services, use the [French Legislation Explorer](#)
- To query the Openfisca Web API in your app, see the [Web API endpoints description](#)
- To make sure the OpenFisca-France Web API (and all the related services) are still up and running, check out our [status page](#)
- To host your own instance of the Openfisca API, go to the [installation documentation](#)

Extensions Packages

Extensions add on the capacities of a country-package.

Example: See [Paris extension](#) and [Rennes extension](#) on top of OpenFisca France.

- To install an Extension, head to the [Extensions documentation](#)

Country package

Country Packages are the basic modules of OpenFisca. They define the [Parameters](#), [Entities](#) and [Variables](#) of a country.

- To install an existing Country Package, head to that package's documentation.

Example: [Openfisca-france's repository](#)

OpenFisca Core

OpenFisca-Core is the main engine: it is the common interface to every Country Package. It binds the Country Package(s), Extension(s) and the engine together. OpenFisca-Core is also where the API is packaged.

- To install OpenFisca-Core, read the [OpenFisca-core Documentation](#).

What's the purpose?

OpenFisca is more a platform than an application: its first target is not the end user but economists, software developers, researchers, teachers, administrations, interested citizens, etc.

Final products can be built on the top of OpenFisca. They trigger tax and benefit variables computations via the web API. For example: [Mes aides](#).

Then, the web API of OpenFisca is used by the team itself to develop tools like the [legislation explorer](#).

These tools are designed to help developers understand the legislation when they write it down into source code, and allow citizens to browse the tax and benefit legislation.

The current version implements a large set of taxes, social benefits and housing provision for France for the last 10 years. But this is only due to a shortage in manpower to enter and update the Tunisian legislation.

The project is 100% free software, it is published on [GitHub](#). It uses the GitHub infrastructure (issues, pull requests, etc.) to communicate internally or with external participants. The team discusses publically on those issues and pull requests and tries to be as transparent as possible.

The project is multi-actors: many people and organizations are involved in the project, reading the legislation and transforming it into source code, developing the Core or web tools, developing external products, etc.

Among them: [Etalab](#), the [Incubateur des services numériques](#), the [IPP](#), the [IDEP](#), the [MSA](#), and [France Stratégie](#).

OpenFisca provides a basic infrastructure, in particular a public instance of its web API, hosted on cheap servers. As it is free software, anyone can reproduce the OpenFisca infrastructure on its server.

Key concepts

This section presents the key concepts required to have a good understanding of OpenFisca, without being too technical.

The first entries are dedicated to the **definitions of the structure** of OpenFisca:

- [Tax and Benefit System](#) as the matrix of the software
- [Variables](#)
- [Parameters](#)
- [Person, entities, role](#)
- [Periods, Instants](#)

Input Data Section gives an insight on the **data** used to compute in OpenFisca.

- [Input Data](#)

Last sections present different **applications** of OpenFisca.

- [Simulation, Computation](#)
- [Reforms](#)

We use the French legislation to illustrate these concepts as it is the only actively maintained country for now. French names are kept as it is.

Tax and Benefit System

Definition

The tax and benefit system is the higher-level instance in OpenFisca. Its goal is to model the legislation of a country.

Basically a tax and benefit system contains simulation [variables](#) (source code) and [legislation parameters](#) (data).

This instance may host so many versions as there are countries in the world.

The OpenFisca core engine is able to simulate any country legislation once it is (partially) represented as source code.

Therefore you have to call at the beginning of your work, the one corresponding to your country of interest.

Application: how to call the Python module

For now only France system is well implemented, your first action should then be:

```
# Call module describing the French System
from openfisca_france import FranceTaxBenefitSystem

# Initialize the legislation
tax_benefit_system = FranceTaxBenefitSystem()
```

Variables and formulas

A variable is property of a person, or an entity (e.g. a family).

For instance:

- The *birth date* of a person
- The amount of *basic income* (in France, RSA) a family can get in a month.
- The amount of *income tax* a household has to pay in a year.
- Whether a family is *living in Paris*, or not.

Input variables

Some variables can only be given as inputs of a simulation. For instance, the *birth date* of a person.

Formulas

Other variables can be calculated thanks to a **formula**.

A formula is a **function** that calculates the value of given variable, for a given period. To do so, it performs (usually arithmetic) operations on the values of other variables, the formula **dependencies**.

For instance:

- The *basic income* of a family can be calculated from its income, and some other information about its situation.
- The *income tax* of a tax household can be calculated the same way.

It is important to note that **all variables can be used as inputs**. This means that even if the *basic income* can be calculated from other variables, I can, for a given simulation, provide it as an input. Then, if another formula asks for the value of *basic income* for a month, the input value will be returned, and the *basic income* formula **won't be executed**.

Default values

When OpenFisca is not able to calculate the value of a variable for a requested period, it returns a **default value**.

The default value of a variable is returned:

- When the value of an input variables is requested, if this variable has not been set in the input for the requested period.
For example: Let's assume the input variable `student` default value is `False`. If the value of `student` for `2017-09` has **not** been set in the input of the simulation, then computing `student` for `2017-09` will return `False`.
- When the value of a variable with formulas is requested, if no formula is defined for the requested period.
For example: Let's assume the variable `basic_income` 's formula is defined starting `2015-01-01`, and its default value is `0`. Computing `basic_income` for `2014-01-01` will return `0`, while computing `basic_income` for `2015-01-01` will use the formula.

Legislation writers can [define a specific default value](#) for each variable.

Parameters

A parameter is a numeric property of the legislation which can evolve over time.

Unlike a [variable](#), a parameter is **not** specific to a specific person or family.

For instance:

- The amount of the *minimum wage*
- The amount of *family allowance per children*
- The *marginal tax scale* used to calculate the income tax

Parameters are used in [formulas](#) to calculate variable values.

[Read more about their implementation in OpenFisca](#)

Person, entities, role

Taxes and benefits can be calculated for different entities: persons, household, companies, etc.

Person

Some openfisca variables are defined for a *person*.

Example: a "salary" is defined as the individual level.

Group entities

Group entities are clusters of *persons* such as the family, the household or the company. A tax and benefit system can define several entities and specifies each time which tax and benefit applies to which entity.

In France the legislation has these *group entities*:

- "familles" ,
- "foyers_fiscaux" and
- "menages" .

Example: the "local tax" is calculated over the "menages" .

Roles

Each person related to a *group entity* has a *role* inside this entity.

The *roles* are:

- for ' familles ': parents and enfants ,
- for ' foyers_fiscaux ': declarants and personnes_a_charge ,
- for ' menages ': personne_de_reference , conjoint , enfants and autres .

You can define as many entities as you want and dispatch persons into them.

Application: module used by OpenFisca

The entities definitions are closely related to a country, therefore they are defined in a Python package (OpenFisca-France) independent from the core engine (OpenFisca-Core).

Periods, Instants

Definition

OpenFisca manipulates time via *periods* and *instants*.

- *Instant*: the atomic unit is a day, so instants are day dates.

Example: the 15th June 2015.

- *Period*: a succession of days.

Example: a month ("July 2015"), a year ("2015"), several months ("July and August 2015") or the eternity.

API

In OpenFisca, periods are encoded in strings. All the valid period formats are referenced in this table:

Period format	Period type	Example	Represents	Disambiguation
AAAA	Calendar year	'2010'	The year 2010.	From the 1st of January 2010 to the 31st of December 2010, inclusive.
AAAA-MM	Month	'2010-04'	The month of April 2010.	From the 1st of April 2010 to the 30th of April 2010, inclusive.
year:AAAA-MM	Rolling year	'year:2010-04'	The 1 year period starting in April 2010.	From the 1st of April 2010 to the 31st of March 2011, inclusive
year:AAAA:N	N years	'year:2010:3'	The years 2010, 2011 and 2012.	From the 1st of January 2010 to the 31st of December 2012, inclusive.
year:AAAA-MM:N	N rolling years	'year:2010-04:3'	The three years period starting in April 2010.	From the 1st of April 2010 to the 31st of March 2013, inclusive.
month:AAAA-MM:N	N months	'month:2010-04:3'	The three months from April to June 2010.	From the 1st of April 2010 to the 30th of June 2010, inclusive.

The smallest unit for OpenFisca periods is the **month**. Therefore:

- All periods are presumed to start on the first day of their first month.
- A period cannot be smaller than a month.

Internally, time is stored as a start instant, a unit (MONTH, YEAR) and a quantity of units.

[Helper functions](#) exist to transform periods or turn them into an instant.

Input Data

You can use OpenFisca with two kind of input information:

- either *test case*: you simulate the legislation for one standard situation
- or *data*: you give a whole population (survey with aggregated data for example) on which you want to apply the legislation.

Scenario

The interface between input information and *input variables* that OpenFisca can handle is called *Scenario*.

Technically speaking, OpenFisca is using [vector computing](#) for performance reasons via the [NumPy](#) Python package

Whatever the input is, *test case* or *data*, the scenario converts it into vectors internally.

Application: how to create a scenario

After initializing the [Tax and Benefit System](#), you now want to create a *scenario* that will allow you in a second step to give input information.

```
# Create a scenario
scenario = tax_benefit_system.new_scenario()
```

Test cases

Test case describes persons and entities with their input variables or attributes.

You may add information at *individual* level or at *entity* level. One input is crucial and shouldn't be forgotten: the *period* of the simulation.

Application: how to initialize a scenario

Test cases can be expressed in Python or in JSON when using the Web API (see the [specific section](#) of the documentation).

In Python you have to use the `init_single_entity` function based on the *scenario*. To give to every person of your *test case* attributes, you have to use the Python dictionary object.

We show here the Python expression for a family constituted by:

- two parents (with attributes: her `age` or her `date_naissance` and her `salaire_de_base`),
- two children (with attribute: their `age`),
- a house (with attributes: the `loyer` and the `statut_occupation_logement`)

```

# Initialize test case
scenario.init_single_entity(
    period = 2015,
    # Variable describing the individuals
    parent1 = dict(
        age = 30,
        salaire_de_base = 15000, # Annual basis
    ),
    parent2 = dict(
        date_naissance = date(1980, 1, 1),
        salaire_de_base = 70000, # Annual basis
    ),
    enfants = [
        dict(age = 12),
        dict(age = 18),
    ],
    # Variable describing the entity
    menage = dict(loyer = 12000, # Annual basis
        statut_occupation_logement = u"Locataire ou sous-locataire
                                     d'un logement loué vide non-HLM",
    ),
)

```

Notice that some input variables are associated to *individus* ("parent1", "parent2" and "children") whereas other are related to *entity* ("menage").

WARNING: Declare the *input variables* on an annual basis.

HINT: For categorical variable you may use either the modality or its number. Example with the [statut d'occupation du logement](#):

```

# Declaration of categorical variable
menage = dict(loyer = 12000,
    statut_occupation_logement = 4,
)

```

Data

Using data as input is not documented yet. Please consult this repository: <https://github.com/openfisca/openfisca-france-data>

Simulation, Computation

Simulation: the framework of computation

A *Simulation* is basically the OpenFisca frame for calculating taxes or benefits.

To calculate any variable you need to create a *Simulation* from the *TaxBenefitSystem* that is to say the framework where you will compute your result.

Technically speaking it is the cache of input data and previously computed results.

It's possible to run many independent simulations using the same `TaxBenefitSystem`.

Application: how to launch a simulation

As soon as you've loaded the *TaxBenefitSystem* of a country and a *Scenario*, you may now create a simulation.

```
# Create a simulation from a scenario
simulation = scenario.new_simulation()
```

Computing variables

Now all the settings are given to run computation of taxes or benefits.

WARNING: Be aware of the period over which you want to have your result. Some measures are calculated on a monthly basis other an annual one.

For further information: see the [tutorial](#) "How to handle periods"

Application: how to calculate a variable

```
# Calcul of the 'impot sur le revenu des personnes physiques'
impot = simulation.calculate('irpp', '2015')
allocations_familiales = simulation.calculate('af', '2015-01')
```

HINT: Don't forget to give the period.

The output is an array:

- positive if it is an amount the *entity* receives from the state.
- negative if it is an amount the *entity* has to pay.

Reforms

OpenFisca can be used to evaluate the quantitative impact of legislation changes.

You may for instance use it to determine who would win or lose from an income tax reform, what would be the impact of a social welfare redesign, or how to finance a universal basic income.

To do so, we use OpenFisca **reforms**. A reform is a set of modifications to be applied to a **reference** tax and benefit system. It generates a reformed tax and benefit system that slightly differs from the original one. We can then run calculations on both of them, and compare results.

To use reforms or code your own ones, check the [reform documentation](#).

Note that OpenFisca simulates only the *mechanics* of taxes and benefits, but doesn't take into account the retro-action of economic agents. For instance, you can estimate the increase of the households disposable income in case a universal basic income is introduced, but OpenFisca won't tell you anything about the consumption increase this policy may generate.

Differences between reforms and extensions

Reforms are sometimes confused with another mechanism: [extensions](#). These two mechanisms do not have the same purpose:

- Use a reform if you want to modify a tax and benefit system in order to study the impact of a legislation change.
- Use an extension if you want to write formulas that are based on a main tax and benefit system, while keeping their code separated from the main country package (e.g. for local prestations).

From law to code

Now let's think practical. The following tutorial and documentation offer you an overview of OpenFisca principles.

If you want to try OpenFisca quickly by yourselves in a [Jupyter Notebook](#), please read the tutorials [here](#).

Tutorial

- [Coding a formula: basic example](#)
- [Introducing an input variable](#)
- [Vectorial computing](#)
- [Periods](#)
- [Legislation evolutions](#)
- [Entities](#)

Additional documentation

- [Writing tests](#)
- [Coding Parameters](#)
- [Coding Reforms](#)
- [Bootstrapping a new country package](#)
- [Inferences used in OpenFisca](#)

Coding a formula

Basic Example

The following piece of code creates a variable named `flat_tax_on_salary`, representing an imaginary tax of 25% on salaries, paid monthly by individuals (not households).

```
class flat_tax_on_salary(Variable):
    value_type = float
    entity = Person
    definition_period = MONTH
    label = u"Individualized and monthly paid tax on salaries"

    def formula(person, period):
        salary = person('salary', period)
        return salary * 0.25
```

Let's explain in details the different parts of the code:

- `class flat_tax_on_salary(Variable):` declares a new variable with the name `flat_tax_on_salary`.
- Metadatas:
 - `value_type = float` declares the type of the variable. Possible types are the basic python types:
 - `bool` : boolean
 - `date` : date
 - `Enum` : discrete value (from an enumerable). [See details](#) in the next section.
 - `float` : float
 - `int` : integer
 - `str` : string
 - `entity = Person` declares which entity the variable is defined for, e.g. a person, a family, a tax household, etc. The different available entities are defined by each tax and benefit system. In `openfisca-france`, a variable can be defined for an `Individu`, a `Famille`, a `FoyerFiscal`, or a `Menage`.
 - `label = u"Individualized..."` gives, in a human-readable language, concise information about the variable.
 - `definition_period = MONTH` states that the variable will be computed on months.
- Formula:
 - `def formula(person, period):` declares the formula that will be used to calculate the `flat_tax_on_salary` for a given `person` at a given `period`. Because `definition_period = MONTH`, `period` is constrained to be a month.
 - `salary = person('salary', period)` calculates the salary of the person, for the given month. This will, of course, work only if `salary` is another variable in the tax and benefit system.
 - `return salary * 0.25` returns the result for the given period.
 - [Dated Formulas](#) have a start and/or an end date.

Testing a formula

To make sure that the formula you have just written works the way you expect, you have to test it. Tests about legislation are written in a [YAML syntax](#). The `flat_tax_on_salary` formula can for instance be tested with the following test file:


```

- name: "Flax tax on salary - No income"
  period: 2017-01
  input_variables:
    salary: 0
  output_variables:
    flat_tax_on_salary: 0

- name: "Flax tax on salary - With income"
  period: 2017-01
  input_variables:
    salary: 2000
  output_variables:
    flat_tax_on_salary: 500

```

You can check the [YAML tests documentation](#) to learn more about how to write YAML tests, and how to run them.

Example with legislation parameters

To access a common legislation parameter, a third parameter can be added to the function signature. The previous formulas could thus be rewritten:

```

class flat_tax_on_salary(Variable):
    value_type = float
    entity = Person
    label = u"Individualized and monthly paid tax on salaries"
    definition_period = MONTH

    def formula(person, period, parameters):
        salary = person('salary', period)

        return salary * parameters(period).taxes.salary.rate

```

`parameters` is here a function that be be called for a given period, and returns the whole legislation parameters (in a hierarchical tree structure). You can get the parameter you are interested in by navigating this tree with the `.` notation.

Introducing an input variable

The syntax to introduce an input variable is very similar to the one we used to code a formula.

For instance:

```
class salary(Variable):
    value_type = float
    entity = Person
    label = u"Salary earned by a person for a given month"
    definition_period = MONTH
```

The only difference is that we do **not** have a formula to calculate the value of a variable.

If we ask the value of `salary` for a given month, the returned result will be:

- The **input** that was provided when initializing the simulation if it exists.
- The **default value** of the Variable if no input has been provided.

Setting a default value

When declaring an input variable, you can change its default value by adding the `default_value` attribute:

```
class french_citizen(Variable):
    value_type = bool
    default_value = True
    entity = Person
    label = u"Whether the person is a French citizen"
    definition_period = YEAR
```

If you do not explicitly define a default value, the following will be used:

- For numeric variables: `0` .
- For boolean variables: `False` .

Advanced example: enumerations (enum)

Usecases

Enumerations are variables that have a limited set of possible values. For instance:

- A person's relationship status: married, single, divorced.
- A household housing occupancy status: owner, tenant, free-lodger, homeless.
- The main occupation of a person: employee, freelance, retired, student, unemployed.

Defining and using an enumeration variable

As an example, let's code a `housing_tax` that is paid by households who own or rent their main homes, but does not apply to households that do not have a stable residence, or are accommodated for free.

The variable `housing_tax` will thus depend on `housing_occupancy_status` , an enumeration variable that can take 4 values: `tenant` , `owner` , `free_lodger` and `homeless` .

First, we can create an [enumerated type](#) `HousingOccupancyStatus` :

```
class HousingOccupancyStatus(Enum):
    tenant = u'Tenant or lodger who pays a rent'
    owner = u'Owner'
    free_lodger = u'Free lodger'
    homeless = u'Homeless'
```

OpenFisca enums are based on Python 3 native enums. Each enum item (for instance `HousingOccupancyStatus.tenant`) has:

- a `name` attribute that contains its key (e.g. `tenant`)
- a `value` attribute that contains its description (e.g. `"Tenant or lodger who pays a monthly rent"`)

Then, create an OpenFisca variable `housing_occupancy_status` :

```
class housing_occupancy_status(Variable):
    value_type = Enum
    possible_values = HousingOccupancyStatus
    default_value = HousingOccupancyStatus.tenant # The default is mandatory
    entity = Household
    definition_period = MONTH
    label = u"Legal housing situation of the household concerning their main residence"
```

You can now use the enum in variable formulas !

For instance, assuming the enumeration and the formula using it are defined in the same file:

```
class housing_tax(Variable):
    value_type = float
    entity = Household
    definition_period = MONTH.
    label = u"Tax paid by each household proportionnally to the size of its accommodation"

    def formula(household, period, legislation):
        accommodation_size = household('accommodation_size', period)
        housing_occupancy_status = household('housing_occupancy_status', period)
        tenant = (housing_occupancy_status == HousingOccupancyStatus.tenant)
        owner = (housing_occupancy_status == HousingOccupancyStatus.owner)

        # The tax is applied only if the household owns or rents its main residency
        return (owner + tenant) * accommodation_size * 10
```

If the enumeration and the formula using it are **not** defined in the same file, an extra step is necessary:

```
class housing_tax(Variable):
    value_type = float
    entity = Household
    definition_period = MONTH.
    label = u"Tax paid by each household proportionnally to the size of its accommodation"

    def formula(household, period, legislation):
        accommodation_size = household('accommodation_size', period)
        housing_occupancy_status = household('housing_occupancy_status', period)
        HousingOccupancyStatus = housing_occupancy_status.possible_values # "Import" the enum type. Careful: do not use python i
        tenant = (housing_occupancy_status == HousingOccupancyStatus.tenant)
        owner = (housing_occupancy_status == HousingOccupancyStatus.owner)

        # The tax is applied only if the household owns or rents its main residency
        return (owner + tenant) * accommodation_size * 10
```

You can now test the formula in a YAML test:

```
- name: Household with free lodger status living in a 100 sq.meters accomodation
period: 2017
input_variables:
  accomodation_size:
    2017-01: 100
  housing_occupancy_status:
    2017-01: free_lodger
output_variables:
  housing_tax: 0
```

Vectorial computing

OpenFisca calculations are all **vectorial**. That means they operate on arrays rather than single (“scalar”) values.

The practical benefit is that computations are almost as expensive for one entity as they are for hundred thousands. This is how datasets can be analysed and how reforms can be modelled accurately. However, to support this feature, you will need to apply some constraints on how you write formulas.

Formulas always return vectors

Each [formula](#) computation in OpenFisca must return a vector.

For instance, for a simulation containing 3 persons whose ages are 41, 42 and 45, executing the following formula:

```
def formula(persons, period, parameters):
    age = persons('age', period)
    print(age)
    ... # do some computation and return a value
```

will print `array([41, 42, 45])`.

This formula code will work the same if there is one Person or three or three million in the modelled situation. Formulas always receive as their first parameter an array of the [entity](#) on which they operate (e.g. *n* Person, Household...) and they should return an array of the same length.

Most of the time, formulas will refer to other variables and NumPy will do the appropriate computation without you even noticing:

```
def formula(persons, period, parameters):
    tax_rebate = parameters(period).tax_rebate # let's say this is 500
    eligibility_multiplier = persons('eligibility_multiplier', period) # and this is [2, 0, 1]: there are three Persons
    return eligibility_multiplier * tax_rebate # this is [1000, 0, 500]. We've returned a vector, yay!
```

What happens if you don't return a vector

As programmers, we more often work with scalars than vectors. We thus have a tendency to write straightforward code that returns a scalar rather than a unidimensional vector (in other words, an array of length 1), and get stuck when wanting to loop over it:

```
# THIS IS NOT A VALID OPENFISCA FORMULA
def formula(persons, period, parameters):
    tax_rebate = parameters(period).tax_rebate # let's say this is worth 500
    rebate_threshold = tax_rebate * persons[0].eligibility_multiplier # so this is 1000; see how we've accidentally left out others
    return rebate_threshold # and this returns 1000. But it's not a vector!
```

OpenFisca will help you notice this mistake by raising an error:

```
The formula 'tax_rebate@2018' should return a NumPy array; instead it returned '1000.0' of type 'float'.
```

In a similar fashion, if you expect a formula to return a boolean and forget that you will actually get an array of boolean values (one for each entity in the situation), you will receive the following safeguard error:

```
ValueError: The truth value of an array with more than one element is ambiguous. Use a.any() or a.all().
```

The rest of this page gives practical replacements for situations in which you get such errors.

Control structures

Some usual control structures such as `if...else`, `switch`, and native Python logical operators such as `or` and `not` do not work with vectors. Semantically however, they all have alternatives, and the only change is in syntax.

`if` / `else`

Let's say you want to write that logically reads as:

```
# THIS IS NOT A VALID OPENFISCA FORMULA
def formula(person, period):
    salary = person('salary', period)
    if salary < 1000:
        return 200
    else:
        return 0
```

This code does not work: it makes the assumption that there is always one single person, and that its salary is provided as a number, while `salary` is actually a vector of salaries that could be of any length.

In such a case, apply the comparison to the *vector* of salaries, which will create a vector of booleans, and then multiply it:

```
def formula(persons, period):
    condition_salary = persons('salary', period) < 1000
    return condition_salary * 200
```

What happens is that for every Person in `persons`, if `condition_salary` is `True` (equivalent to `1` in logical algebra), the returned value will be `200`. And if `condition_salary` is `False` (equivalent to `0`), the returned value will be `0`.

Ternaries

Let's now write a formula that returns `200` if the Person's salary is lower than `1000`, and `100` otherwise.

The NumPy function `where` offers a simple syntax to handle these cases.

```
def formula(persons, period):
    condition_salary = persons('salary', period) < 1000
    return where(condition_salary, 200, 100)
```

`where` takes 3 arguments: a vector of boolean values (the “condition”), the value to set for this element in the vector if the condition is met, and the value to set otherwise.

This `where` function is provided directly by NumPy. There are many other [NumPy functions](#) provided that can be useful.

Multiples conditions

Let's consider a more complex case, where we want to attribute to a person:

- `200` if their salary is less than `500` ;
- `100` if their salary is strictly more than `500`, but less than `1000`;
- `50` if their salary is strictly more than `1000`, but less than `1500`;
- `0` otherwise.

We can use the NumPy function `select` to implement this behaviour:

```
def formula(person, period):
    salary = person('salary', period)
    return select(
        [salary <= 500, salary <= 1000, salary <= 1500, salary > 1500],
        [200, 100, 50, 0],
    )
```

If the first condition is met, the first value will be assigned, without considering the other conditions. For instance, if `salary = 100` , `salary <= 500` is true and therefore `200` will be assigned. It doesn't matter that `salary <= 1000` is also true.

If the first condition is not met, then only the second condition will be considered, and so on. If no condition is met, `0` will be assigned.

Complex conditions

If no [NumPy function](#) helps you express a very specific condition, you can code arbitrary conditions using `*` instead of `and` , and `+` instead of `or` .

For instance, let's consider that a person will be granted `200` if either:

- they are more than 25 *and* make less than `1000` per month;
- or they are disabled.

```
def formula(person, period):
    condition_age = person('age') >= 25
    condition_salary = person('salary', period) < 1000
    condition_handicap = person('handicap')
    condition = condition_age * condition_salary + condition_handicap
    return condition * 200
```

You should always use [NumPy function](#) such as `where` and `select` when they are relevant: logical operations using arithmetic operators should be used as last resort as they are not very readable.

Arithmetic operations

Basic arithmetic operations such as `+` or `*` behave the same way on vectors than on numbers, you can thus use them in OpenFisca formulas. However, some operations must be adapted.

Scalar (won't work)	Vectorial alternative
<code>min</code>	<code>min_(x,y)</code>
<code>max</code>	<code>max_(x,y)</code>
<code>round</code>	<code>round_(x,y)</code>

Boolean operations

Scalar (won't work)	Vectorial alternative
<code>not</code>	<code>not_(x,y)</code>
<code>and</code>	<code>x * y</code>
<code>or</code>	<code>x + y</code>

String concatenation

The `+` operator, as well as formatted `%s` strings for concatenation should be replaced by a call to `concat(x, y)` .

Periods

A period can be a month, a year, `n` successive months, `n` successive years or the eternity.

Periods for variable

Most of the quantities calculated in openfisca can change over time. Therefore, each formula calculates a variable for a person (or a family, etc.) **for a given period**.

This period is always the second argument of the formulas:

```
class salary(Variable):
    value_type = float
    entity = Person
    label = u"Salary for a month"
    definition_period = MONTH

    def formula(person, period):
        ...
```

The size of the period is constrained by the class attribute `definition_period` :

- `definition_period = MONTH` : The variable may have a different value each month. *For example*, the salary of a person. When `formula` is executed, the parameter `period` will always be a whole month. Trying to compute `salary` with a period that is not a month will raise an error before entering `formula` .
- `definition_period = YEAR` : The variable is defined for a year or it has always the same value every months of a year. *For example*, if taxes are to be paid yearly, the corresponding variable is yearly. When `formula` is executed, the parameter `period` will always be a whole year (from January 1st to December 31th).
- `definition_period = ETERNITY` : The value of the variable is constant. *For example*, the date of birth of a person never changes. `period` is still the 2nd parameter of `formula` . However when `formula` is executed, the parameter `period` can be anything and it should not be used.

Calculating dependencies for a period different than the one they are defined for

Calling a formula with a period that is incompatible with the attribute `definition_period` will cause an error. For instance, if we assume that a person `salary` is paid monthly:

```
class taxes(Variable):
    value_type = float
    entity = Person
    label = u"Taxes for a whole year"
    definition_period = YEAR

    def formula(person, period): # period is a year because definition_period = YEAR
        salary_past_year = person('salary', period) # salary is computed on a year while it's a montly variable, openfisca will
        ...
```

However, sometimes, we do need to estimate a variable for a different period than the one it is defined for.

We may for example want to get the sum of the salaries perceived on the past year, or the past 3 months. The option `ADD` tells openfisca to split the period into months, compute the variable for each month and sum up the results:


```

class taxes(Variable):
    value_type = float
    entity = Person
    label = u"Taxes for a whole year"
    definition_period = YEAR

    def formula(person, period): # period is a year because definition_period = YEAR
        salary_last_year = person('salary', period, options = [ADD])
        ...

```

The option `DIVIDE` allows you to do the opposite: evaluating a quantity for a month while the variable is defined for a year. Openfisca computes the variable for the whole year that contains the specified month and then divides the result by 12.

```

class salary_net_of_taxes(Variable):
    value_type = float
    entity = Person
    label = u"Monthly salary, net of taxes"
    definition_period = MONTH

    def formula(person, period): # period is a month because definition_period = MONTH
        # The variable taxes is computed on a year, monthly_taxes equals the 12th of that result
        monthly_taxes = person('taxes', period, options = [DIVIDE])

        # salary is a monthly variable, period is a month: no option is required
        salary = person('salary', period)

        return salary - monthly_taxes

```

Calculating dependencies for a specific period

It happens that the formula to calculate a variable at a given period needs the value of another variable for another period. Usually, the second period is defined relatively to the first one (previous month, last three month, current year).

For instance, we want to compute an unemployment benefit that equals half of last year's salary, if the person had no income for the past 3 months.

```

class unemployment_benefit(Variable):
    value_type = float
    entity = Person
    label = u"Unemployment benefit"
    definition_period = MONTH

    def formula(person, period):
        salary_last_3_months = person('salary', period.last_3_month)
        salary_last_year = person('salary', period.last_year)

        is_unemployed = (salary_last_3_months == 0)
        return 0.5 * salary_last_year * is_unemployed

```

You can generate any period with the following properties and methods:

Period	Meaning
<code>period.this_month</code>	First month-length period that includes the start of <code>period</code>
<code>period.last_month</code>	Month preceding <code>period.this_month</code>
<code>period.this_year</code>	First year-length period that includes the start of <code>period</code>
<code>period.last_year</code>	Year preceding <code>period.this_year</code>
<code>period.n_2</code>	2 years before <code>period.this_year</code>
<code>period.last_3_months</code>	The three-month period preceding <code>period.this_month</code>
<code>period.offset(n, 'month')</code>	<code>period</code> translated by n months (backwards if n < 0)
<code>period.offset(n, 'year')</code>	<code>period</code> translated by n years (backwards if n < 0)
<code>period.start.period('year')</code>	Year-long period starting a the same time than <code>period</code>
<code>period.start.period('month')</code>	Month-long period starting a the same time than <code>period</code>
<code>period.start.period('year', n)</code>	n-year-long period starting a the same time than <code>period</code>
<code>period.start.period('month', n)</code>	n-month-long period starting a the same time than <code>period</code>

You can find more information on the `period` object in the [reference documentation](#) (*not available yet*)

Automatically process variable inputs defined for periods not matching the `definition_period`

By default, when you provide a simulation inputs, you won't be able to set a variable value for a period that doesn't match its `definition_period`.

For instance, as the `definition_period` of `salary` is `MONTH`, if you provide as an input a value of `salary` for `2015`, an error will be raised.

It is however possible to define an automatic behaviour to cast yearly inputs into monthly values. To do this, add a `set_input` class attribute to a variable.

- if `set_input = set_input_divide_by_period`, the 12 months are set equal to the 12th of the input value,
- if `set_input = set_input_dispatch_by_period`, the 12 months are set equal to input value.

For instance, let's slightly modify the code of `salary`:

```
class salary(Variable):
    value_type = float
    entity = Person
    label = u"Salary for a month"
    definition_period = MONTH
    set_input = set_input_divide_by_period

    def formula(person, period):
        ...
```

We can now provide an input for `2015` for `salary`: no error will be raised, and the value will be automatically split between the 12 months of '2015.

Legislation evolutions

Openfisca handles the fact that the legislation changes over time.

Parameter evolution

Many legislation parameters are regularly re-evaluated while the variables using them stay the same.

Example: the `taxes` parameter can change without altering the code of the `flat_tax_on_salary` variable that uses that parameter.

In that case, add the new parameter values and their start dates in the appropriate parameter files.

How to update a parameter

Open the file where the parameter is described

```
taxes:
  salary:
    rate:
      description: Rate for the flat tax on salaries
      values:
        2016-01-01:
          value: 0.25
          reference: https://www.legislation-source.com/2016
        2015-01-01:
          value: 0.20
          reference: https://www.legislation-source.com/2015
```

Add a new value to this parameter

```
taxes:
  salary:
    rate:
      description: Rate for the flat tax on salaries
      values:
        2017-01-01:
          value: 0.25
          reference: https://www.legislation-source.com/2017
        2016-01-01:
          value: 0.25
          reference: https://www.legislation-source.com/2016
        2015-01-01:
          value: 0.20
          reference: https://www.legislation-source.com/2015
```

After this change `legislation(period).taxes.salary.rate` will return the corresponding values:

- `legislation('2015-04').taxes.salary.rate` will return `0.2`
- `legislation('2017-01').taxes.salary.rate` will return `0.3`
- `legislation('2022-01').taxes.salary.rate` will return `0.3`

[Read more about how to code parameters.](#)

Formula evolution

Some fiscal or benefit mechanism significantly evolve over time and call for a change in the formula that computes them. In this case, a simple parameter adjustment is not enough.

For instance, let's assume that from the 1st of Jan. 2017, the `flat_tax_on_salary` is not applied anymore on the first `1000` earned by a person.

We implement this rule by adding a new formula to our variable, and *dating* it:

```
class flat_tax_on_salary(Variable):
    value_type = float
    entity = Person
    label = u"Individualized and monthly paid tax on salaries"
    definition_period = MONTH

    def formula_2017(person, period, parameters):
        salary = person('salary', period)
        salary_above_1000 = min(salary - 1000, 0)
        return salary_above_1000 * parameters(period).taxes.salary.rate

    def formula(person, period, parameters):
        salary = person('salary', period)

        return salary * parameters(period).taxes.salary.rate
```

If the `flat_tax_on_salary` is calculated for a person **before** the 31st of Dec. 2016 (included), `formula` is used. If it is called **after** the 1st of Jan 2017 (included), `formula_2017` is used.

Formula naming rules:

- A formula name must always start with `formula` .
- To define a starting date for a formula, we add to its name a suffix made of an underscore followed by a date.
 - For instance, `formula_2017_01_01` is active from the 1st of Jan. 2017.
- When defining a date, the month is given **before** the day.
- When no month or day is specified, OpenFisca uses '01' as default value.
 - For instance, `formula_2017` is equivalent to `formula_2017_01_01` .
- If no date is specified for a formula, OpenFisca will consider that this formula has been active since the dawn of time (or more precisely, since `0001-01-01` , as Python does not handle B.C. dates).
 - For instance, `formula` is active on `2010` .
- A formula is active until another formula, starting later, becomes active and replaces it (or until the variable `end` date is reached, as we'll see further down in the [Variable end](#) section).
 - For instance, `formula` is active until `2016-12-31` (included). On the day after, `2017-01-01` , `formula_2017` becomes active, and `formula` becomes inactive.

Formula introduction

In our previous example, we assumed that `flat_tax_on_salary` had *always* had a formula, since the dawn of time. This is a reasonable hypothesis if we are only interested in running computations for recent years.

But most fiscal and benefit mechanisms have been introduced at some point. Let's for instance assume that our `flat_tax_on_salary` only appeared in our legislation on the 1st of June 2005.

This is easily implemented by *dating* the two formulas:

```

class flat_tax_on_salary(Variable):
    value_type = float
    entity = Person
    label = u"Individualized and monthly paid tax on salaries"
    definition_period = MONTH

    def formula_2017(person, period, parameters):
        salary = person('salary', period)
        salary_above_1000 = min(salary - 1000, 0)
        return salary_above_1000 * parameters(period).taxes.salary.rate

    def formula_2005_06(person, period, parameters):
        salary = person('salary', period)

        return salary * parameters(period).taxes.salary.rate

```

Only a few characters changed in comparison with the last example: the suffix `_2005_06` has been added to the second formula name.

Note that if `flat_tax_on_salary` is calculated **before** 2005-05-31 (included), *none* of the two formulas is used, as they are *both inactive* at this time. Instead, **the variable default value is returned**.

Variable end

As the legislation evolves, some fiscal or benefit mechanisms disappear.

Let's for instance assume that a `progressive_income_tax` used to exist before the `flat_tax_on_salary` was introduced. This progressive tax then disappeared on the 1st of June 2005.

This is implemented with an `end` attribute that define the *last day* a variable can be calculated:

```

class progressive_income_tax(Variable):
    value_type = float
    entity = Person
    label = u"Former tax replaced by the flat tax on the 1st of June 2005"
    definition_period = MONTH
    end = '2005-05-31'

    def formula(person, period, legislation):
        # Apply a marginal scale to the person's income
        ...

```

If `progressive_income_tax` is called **before** 2005-05-31 (included), `formula` will be used.

However, if `progressive_income_tax` is calculated **after** 2005-06-01 (included), `formula` is **not** used, as it is not active anymore at this time. Instead, **the variable default value is returned**.

Note that:

- The `end` day is **inclusive**: it is the last day a variable and its formulas are active (and not the first day it is not active anymore).
- The `end` value is a string of format `YYYY-MM-DD` where `YYYY`, `MM` and `DD` are respectively a year, month and day.
- When defining a date, the month is given **before** the day.

Entities

Every variable is defined for a type of **entity**: for instance persons or households.

However, I may for instance:

- in a formula defined for a person, want to know some property of their household.
- in a formula defined for a household, want to know some property of the household members.

Group entity composition

You can get the number of person with a given role in an entity with the `nb_persons(role)` method. If no role is given, it will return the numbers of people in the entity.

```
def formula(household, period):  
    nb_persons = household.nb_persons()  
    nb_adults = household.nb_persons(Household.ADULT)  
    nb_children = household.nb_persons(Household.CHILD)
```

Note that roles are constants that can be accessed from their entity with the notation `Entity.ROLE` (in uppercase).

Check if a person has a given role

You can know whether a person has a certain role with the `has_role(role)` method:

```
def formula(person, period):  
    is_adult = person.has_role(Household.ADULT)  
    is_child = person.has_role(Household.CHILD)
```

Aggregation

For an entity, several methods allow you to aggregate the values of a quantity defined for its members.

`entity.members('variable_name', period)` allows you to calculate the value of a variable for all members of an entity.

`entity.sum(result)` sums previously calculated results. Similar functions such as `min`, `max`, `any`, and `all` work the same way.

For instance, let's imagine a basic income paid to households with the following rules:

- Any household is entitled to 500€ a month per adult, and 200€ a month per children.
- The sum of salaries from all household members are deducted from the amount of the benefit.

```

class basic_income(Variable):
    value_type = float
    entity = Household
    label = u"Basic income paid to households"
    definition_period = MONTH

    def formula(household, period):
        nb_adults = household.nb_persons(Household.ADULT)
        nb_children = household.nb_persons(Household.CHILD)
        salaries = household.members('salary', period)
        sum_salaries = household.sum(salaries)

        result = nb_adults * 500 + nb_children * 200 - sum_salaries
        result = max_(result, 0)

    return result

```

Projection

`person.entity('variable_name', period)` allows you to get the value of `variable_name` for the entity containing `person` .

Let's for example consider that any college student whose family benefits from the basic income will also individually be granted a scholarship of 100€ per month:

```

class college_scholarship(Variable):
    value_type = float
    entity = Person
    label = u"College Scholarship for basic income recipients."
    definition_period = MONTH

    def formula(person, period):
        is_student = person('is_student', period)
        has_household_basic_income = person.household('basic_income', period) > 0

        return is_student * has_household_basic_income * 100

```

Similarly, `entity.unique_role('variable_name', period)` allows you to get the value of `variable_name` for `person` who has the role `unique_role` in `entity` .

For instance, let's assume `Household` has two unique roles, `main_declarant` and `partner` .

```

def formula(household, period):
    household.main_declarant('salary', period) # main declarant's salary
    household.partner('salary', period) # partner's salary

```

Reforms

A [reform](#) is a set of modifications to be applied to a tax and benefit system, usually to study the quantitative impact of a possible change of the law.

See the reference documentation of the class [Reform](#).

Writing a reform

Let's for instance assume that we want to simulate the effect of a reform that changes the way the `income_tax` is calculated.

We would write such a reform this way:

```
class income_tax(Variable):
    entity = Household
    label = u'Alternative formula to calculate the income tax, under experimentation'

    def formula(household, period):
        # (...)

class income_tax_reform(Reform):
    name = u'Reform on income tax'

    def apply(self):
        self.update_variable(income_tax)
```

A `Reform` **must** define an `apply()` method that describes all the modifications to be applied to the original tax and benefit system to get the reformed one.

Note that the reference tax and benefit system won't be modified. The `apply()` function will be applied to a copy of the tax and benefit system.

All the [methods](#) used to build a tax and benefit system can also be used to reform it.

A reform that modifies a formula (such as our `income_tax_reform` example) is called a *structural reform*. It redefines the way a variable is calculated.

Parametric reforms

A reform that apply changes to legislation parameters is called a *parametric reform*.

Note that a reform can be both structural and parametric, modifying and/or adding variables *and* parameters. In that case, it is common practice to call it a structural reform anyway, the structural part outweighing the parametric one.

To modify the legislation parameters in the reform, you can call the method `self.modify_parameters`, which takes a function as a parameter.

This function performs the modifications you want to apply to the legislation. It takes as a parameter a copy of the reference tax and benefit system parameters: `parameters`. You can then modify and return `parameters`.

Update the value of a parameter


```
def modify_parameters(parameters):
    reform_period = periods.period("2015")
    parameters.tax_on_salary.scale[1].threshold.update(period = reform_period, value = 4000)
    return parameters

class increase_minimum_wage(Reform):
    name = u'Increase the minimum wage'

    def apply(self):
        self.modify_parameters(modifier_function = modify_parameters)
```

Add new parameters

You can load new parameters from a directory containing YAML files and add them to the reference parameters.

```
import os
from openfisca_core.parameters import load_parameter_file

dir_path = os.path.dirname(__file__)

def modify_parameters(parameters):
    file_path = os.path.join(dir_path, 'plf2016.yaml')
    reform_parameters_subtree = load_parameter_file(file_path, name='plf2016')
    parameters.add_child('plf2016', reform_parameters_subtree)
    return parameters

class some_reform(Reform):
    def apply(self):
        self.modify_parameters(modifier_function = modify_parameters)
```

Add new parameters dynamically

In some cases, loading new parameters from YAML files is not practical. For example, you may want to add parameters from values computed dynamically. In such cases you can use the python objects defined in the [parameters module](#) :

```
from openfisca_core.parameters import ParameterNode

def modify_parameters(parameters):
    reform_parameters_subtree = ParameterNode('new_tax', validated_yaml = {
        'decote_seuil_celib': {
            'values': {
                "2015-01-01": {'value': f(a, b, c)},
                "2016-01-01": {'value': None}
            }
        },
        'decote_seuil_couple': {
            'values': {
                "2015-01-01": {'value': g(a, b, c)},
                "2016-01-01": {'value': None}
            }
        },
    })

    parameters.add_child('new_tax', reform_parameters_subtree)

class some_reform(Reform):
    def apply(self):
        self.modify_parameters(modifier_function = modify_parameters)
```

Using a reform in Python

Reforms can be applied in Python with the following syntax:

```
from openfica_france import CountryTaxBenefitSystem

class income_tax_reform(Reform):
    # (...)

tax_benefit_system = CountryTaxBenefitSystem()

reformed_tax_benefit_system = income_tax_reform(tax_benefit_system)
```

Reforms can be chained:

```
from openfica_france import CountryTaxBenefitSystem

class income_tax_reform(Reform):
    # (...)

class increase_minimum_wage(Reform):
    # (...)

tax_benefit_system = CountryTaxBenefitSystem()

reformed_tax_benefit_system = income_tax_reform(
    increase_minimum_wage(tax_benefit_system)
)
```

The [Getting Started Notebook](#) contains an example of reform use.

Real examples

Examples can be found on the [OpenFisca-France reforms directory](#).

Inferences

Here are the places in which inferences take place in OpenFisca:

- In period management, through `calculate_output` , which can automatically sum or divide values over periods to match the requested period.
- In period management, through `set_input` , which can automatically sum or divide input values over periods to match the computable period.
- In some formulas, through `base_function` , which can yield values that the original requested formula could not compute on its own.
- In some formulas, through `max_nb_cycles` , which can block the computation toward the past and thus not allow some values to be computed.

These inferences are not considered good practice, as they tend to make computations less consistent and predictable. You should tend to avoid relying on them as much as possible.

Known issues

With `base_function`

Default values in input variables are the source of the issue: if a formula needs values that are undefined (e.g. because they are in previous months from the calculation), it won't crash nor log anything because the input variables return their default values. This inference behaviour seems to be fine for simulations based on a large population, but production would expect a time-based inference, where the value is copied from its closest defined value rather than a global default one.

The advised workaround is to *always* request the period when requesting parameters, and to never rely on the fragile concept of a “simulation period”.

Writing YAML tests

The recommended way to write tests is to use YAML tests.

Each formula should be tested at least with one test, and better with specific boundary values (thresholds for example).

Terminology: Python dictionary are called associative arrays in YAML.

Example

In `irpp.yaml` we see:

```
- name: "IRPP - Célibataire ayant des revenus salariaux (1AJ) de 20 000 €"
  period: 2012
  absolute_error_margin: 0.5
  input_variables:
    salaire_imposable: 20000
  output_variables:
    irpp: -1181
```

Common keys

- `name` (string)
- `period` (string with period syntax)
- `keywords` (list of strings, optional)
- `description` (string, optional, multiline)
- `absolute_error_margin` (number, optional)
- `relative_error_margin` (number, optional)
- `input_variables` (associative array, keys are variable names, values are numbers)
- `output_variables` (associative array, keys are variable names, values are numbers)
- other any key defined in the model

Syntax

Testing formulas by giving input variables

This is the simplest way to test formulas when you only need to give input values for only one individual.

- First, name your test. Start a test with `-`, which is the YAML list separator, followed by a space, the field `name`, and the test name as a string.

```
- name: "IRPP - Célibataire ayant des revenus salariaux (1AJ) de 20 000 €"
```

- Then add the other relevant keys to your test. Usually, one defines the keys `period`, `keywords`, `description`, `absolute_error_margin` (or `relative_error_margin`) and their associated chosen values as follows:

```
- name: "IRPP - Célibataire ayant des revenus salariaux (1AJ) de 20 000 €"
  period: 2012
  absolute_error_margin: 0.5
```

- Create nested dictionaries within the keys `input_variables` and `output_variables`, which keys are variable names and values are numbers, respectively input and expected values. For instance:

```
- name: "IRPP - Célibataire ayant des revenus salariaux (1A) de 20 000 €"
  period: 2012
  absolute_error_margin: 0.5
  input_variables:
    salaire_imposable: 20000
    salaire_brut: 20000
  output_variables:
    irpp: -1181
```

Testing formulas giving a test case

This is the simplest way to test formulas when you need to give input values for many individuals which are dispatched into entities.

See the last test of [cotisations_sociales_simulateur_IPP.yaml](#)

In this case, there is another convention:

- do not include the field `input_variables` but instead define new keys corresponding to the entities:

```
- name: "IRPP - Famille ayant des revenus salariaux de 20 000 €"
  period: 2012
  absolute_error_margin: 0.5
  familles:
  menages:
  foyers_fiscaux:
```

- define the individuals with their `id` and their variables:

```
individus:
  - id: "parent1"
    date_naissance: 1972-01-01
    depcom_entreprise: "69381"
    primes_fonction_publique: 500
  - id: "parent2"
    date_naissance: 1972-01-01
    depcom_entreprise: "69381"
    primes_fonction_publique: 500
    traitement_indiciaire_brut: 2000
  - id: "enfant1"
    date_naissance: 2000-01-01
  - id: "enfant2"
    date_naissance: 2009-01-01
```

- specify the relations between individuals and their entity:

```
familles:
  parents: ["parent1", "parent2"]
  enfants: ["enfant1", "enfant2"]
menages:
  personne_de_reference: "parent1"
  conjoint: "parent2"
  enfants: ["enfant1", "enfant2"]
foyers_fiscaux:
  declarants: ["parent1", "parent2"]
  personnes_a_charge: ["enfant1", "enfant2"]
```

- finally, define a dictionary of the expected values of the output variables. Each output variable takes a list of length equal to the number of individuals defined in the test. E.g, for a family of four individuals with two working parents and two unemployed children, the output variable `salaire_super_brut` is defined as follows:

```
output_variables:
  salaire_super_brut: [3500, 2500, 0, 0]
```

Testing formulas using variables defined for multiple periods

Input or output variables can be defined for multiple periods by giving an associated array which keys are a period expression and values are the value for that period.

Values can be arithmetic expressions too.

```
individus:
  salaire_de_base:
    2013-01: 35 * 52 / 12 * 9
    2013-02: 35 * 52 / 12 * 9
    2013-03: 35 * 52 / 12 * 9
```

Running a test

To run YAML tests, use the command line tool `openfisca-run-test`, documented [here](#):

```
openfisca-run-test path/to/file.yaml
```

You can also run tests programatically using the `test_runner` module.

Next steps

Other kinds of tests exist, see [contribute/tests](#).

Implementing legislation parameters

Legislation parameters can be found in the `parameters` directory of your country package.

The parameters are organized with in a [tree structure](#).

Example: `tax_on_salary.public_sector.rate` can be found in `parameters/tax_on_salary/public_sector/rate.yml`.

Example of a `parameters` directory:

- `parameters`
 - `tax_on_salary`
 - **`tax_scale.yml`**
 - `public_sector`
 - **`rate.yml`**
 - `universal_income`
 - **`minimum_age.yml`**
 - **`amount.yml`**

In this file structure:

- `tax_on_salaries`, `tax_on_salary.public_sector`, `universal_income` are **nodes**;
- `tax_on_salaries.tax_scale`, `tax_on_salary.public_sector.rate`, `universal_income.minimum_age`, `universal_income.amount` are **parameters** (or scales).

How to write a new parameter

if you wish to update a parameter, read our [legislation evolution page](#).

1. Find where the parameter fits

A parameter is located inside a **node**, that has the same name as the directory it is contained in.

Example: `tax_on_salary.public_sector` is the node that contains the `tax_on_salary.public_sector.rate` parameter.

1. Create a new parameter YAML file

A legislative parameter is defined by a YAML file of the same name. Possible attributes are:

- `description` (optional) Description;
- `reference` (optional) Legislative reference;
- `unit` (optional) Can be:
 - `year` : The values are years;
 - `currency` : The values are in the unit of currency of the country;
 - `/1` : The values are percentages, with `1.0` =100%;
- `values` : Value of the parameter for several dates.

Sample file `parameters/universal_income/amount.yml`

```
description: Universal income
unit: currency
values:
  1993-01-01:
    value: 1000
  2010-01-01:
    value: 1500
  reference: http://law.reference.org/universal_income
  2020-01-01:
    expected: 1700
```

In this example, the parameter `universal_income.amount` is:

- undefined before 1993;
- equal to 1000 from 1993 to 2010;
- equal to 1500 in 2010
- expected to be raised to 1700 "local currency" in 2020.

The ordering of the dates has no effect. It is recommended to add legislative references for every value?

1. Use the parameter in a variable

See [this example of a variable using legislation parameters](#).

Naming conventions and reserved words

Names should begin with a lowercase letter and should contain only lowercase letters and the underscore (`_`).

The following keywords are reserved and should not be used as names : `description` , `reference` , `values` , `brackets` .

YAML parameter files should not be name `index.yaml` .

Advanced uses

Use a YAML files to define nodes

A node can be defined with a YAML file instead of a directory. In such a case, the name of the file defines the name of the node. Such a file can define children nodes (which can define grandchildren...).

Sample `parameters/tax_on_salary.yaml` :

```
description: Tax on salaries
reference: http://fiscaladministration.government/tax_on_salaries.html
tax_scale:
  bracket:
    ...
public_sector:
  description: Tax on salaries for public sector
  rate:
    values:
      ...
```

Create Scales

Scales are constituted of brackets. Brackets are defined by amounts, bases, rates, average rates and thresholds.

Sample `parameters/tax_on_salary/tax_scale.yaml` :

```
description: Scale for tax on salaries
brackets:
- rate:
  1950-01-01:
    value: 0.0
  2010-01-01:
    value: 0.02
  threshold:
  1950-01-01:
    value: 0.0
- rate:
  1950-01-01:
    value: 0.2
  threshold:
  1950-01-01:
    value: 2000
```


Example: [the french tax scale on income](#)

Import parameters from IPP tables

This section applies only to OpenFisca-France.

The [IPP](#) is a French centre in economics which produces tax and benefit tables in the `xlsx` format, with parameters history.

The OpenFisca team works on importing those data into the YAML parameter files of OpenFisca-France.

See [this README](#) for more information.

Computing a parameter that depends on a variable (fancy indexing)

Sometimes, the value of a parameter depends on a variable (e.g. a housing benefit that depends on the zone the house is built on).

To be more specific, let's assume that:

- Households who rent their accomodation can get a `housing_benefit`
- The amount of this benefit depends on which `zone` the household lives in. The `zone` can take only three values: `zone_1`, `zone_2` or `zone_3`.
- The amount also depends on the composition of the household.

The parameters of this benefit can be defined in a `housing_benefit.yaml` file:

```

zone_1:
  single:
    description: "Amount of housing benefit for a single person, in zone 1"
    values:
      2015-01-01:
        value: 150
  couple:
    description: "Amount of housing benefit for a couple, in zone 1"
    values:
      2015-01-01:
        value: 250
  single:
    description: "Amount of housing benefit per child, in zone 1"
    values:
      2015-01-01:
        value: 80
zone_2:
  single:
    description: "Amount of housing benefit for a single person, in zone 2"
    values:
      2015-01-01:
        value: 120
  couple:
    description: "Amount of housing benefit for a couple, in zone 2"
    values:
      2015-01-01:
        value: 220
  per_child:
    description: "Amount of housing benefit per child, in zone 2"
    values:
      2015-01-01:
        value: 60
zone_3:
  single:
    description: "Amount of housing benefit for a single person, in zone 3"
    values:
      2015-01-01:
        value: 100
  couple:
    description: "Amount of housing benefit for a couple, in zone 3"
    values:
      2015-01-01:
        value: 180
  per_child:
    description: "Amount of housing benefit per child, in zone 3"
    values:
      2015-01-01:
        value: 50

```

Then the formula calculating `housing_benefit` can be implemented with:

```

def formula(household, period, parameters):
    is_couple = household('couple', period)
    nb_children = household('nb_children', period)
    zone = household('zone', period)

    P = parameters(period).housing_benefit[zone]

    return where(is_couple, P.couple, P.single) + nb_children * P.per_children

```

`parameters(period).housing_benefit[zone]` return the parameters for the zone corresponding to the household.

If there are many households in your simulation, this parameter will be **vectorial** : it may have a different value for each household of your entity.

To be able to use this notation, all the children node of the parameter node `housing_benefit` must be **homogenous**. In the previous example, `housing_benefit.zone_1` , `housing_benefit.zone_2` , `housing_benefit.zone_3` are homogenous, as they have the same subnodes.

However, let's imagine that `housing_benefit.yaml` had another subnode named `coeff_furnished`, which described a coefficient to apply to the benefit if the accommodation is rented furnished:

`housing_benefit.yaml` content:

```
coeff_furnished:
  description: "Coefficient to apply if the accommodation is rented furnished"
  values:
    2015-01-01:
      value: 0.75
zone_1:
  single:
    description: "Amount of housing benefit for a single person, in zone 1"
    values:
      2015-01-01:
        value: 150
(...)
```

In this case, `parameters(period).housing_benefit[zone]` would raise an error, whatever `zone` contains, as **the homogeneity condition is not respected**: `housing_benefit.zone_1` is a node, while `housing_benefit.coeff_furnished` is a parameter.

To solve this issue, the good practice would be to create an intermediate node `amount_by_zone`:

`housing_benefit.yaml` content:

```
coeff_furnished:
  description: "Coefficient to apply if the accommodation is rented furnished"
  values:
    2015-01-01:
      value: 0.75

amount_by_zone:
  zone_1:
    single:
      description: "Amount of housing benefit for a single person, in zone 1"
      values:
        2015-01-01:
          value: 150
(...)
```

And then to get `parameters(period).housing_benefit.amount_by_zone[zone]`

How to navigate the parameters in Python

Set-up your python file by importing a `country` package and building the `tax and benefits system`

Example :

```
import openfisca_country_template
tax_benefit_system = openfisca_country_template.CountryTaxBenefitSystem()
```

Access a parameter for all periods

To access a point in the parameter tree, call `tax_benefit_system.parameters`

Example : Access the `benefit` branch of the `openfisca-country-template` legislation

```
tax_benefit_system.parameters.benefits
```

Returns:

```
basic_income:
    2015-12-01: 600.0
housing_allowance:
    2016-12-01: None
    2010-01-01: 0.25
```

Access `basic_income` , a parameter of the `benefits` branch.

```
tax_benefit_system.parameters.benefits.basic_income
```

Returns:

```
2015-12-01: 600.0
```

Access a parameter for a specific period

Request a branch of a parameter at a given date with the `parameters.benefits('2015-07-01')` notation.

How to update parameters in python

To add an entry to an existing parameter, use `update` :

Example:

```
tax_benefit_system.parameters.benefits.basic_bro.update("2017-01", value = 2000)
tax_benefit_system.parameters.benefits.basic_bro
```

Returns:

```
2017-01-01: 2000
2015-12-01: 600.0
```

Bootstrapping a new country package

If you want to use OpenFisca to run simulations about your own country's legislation, our [country package template](#) will provide you all the instructions and boilerplate code you need to quickly get something working.

If you want to see more complex and complete examples of legislation coded in OpenFisca, you can check the [French](#), [Tunisian](#) and [Senegalese](#) country packages.

OpenFisca Web API

OpenFisca provides a web API package compatible with all country packages. Using a web interface, App Developers can access information and computations without installing anything locally.

Public France API

The latest version of the France web api is fr.openfisca.org/api/v21 . Its endpoints are documented in fr.openfisca.org/legislation/swagger . The stability of this API is guaranteed over time.

Use Cases

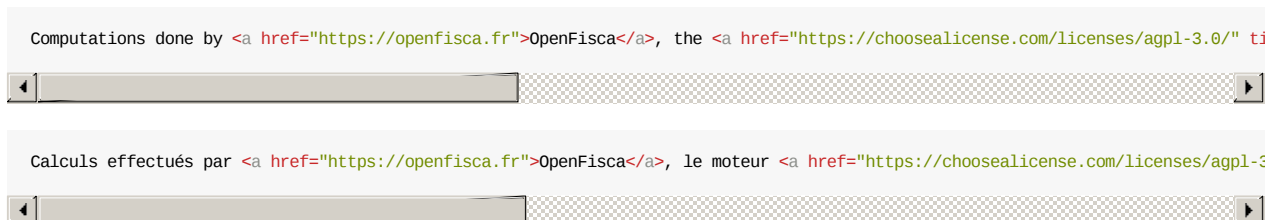
The following services use the OpenFisca Web API:

- fr.openfisca.org/legislation, giving you information on available OpenFisca variables.
- [Mes Aides](#), the French social benefits simulator.
- [PA-comp](#), a divorce fiscal impact simulator.

Conditions

Please remember that OpenFisca is free software, licensed under an [Affero GPL license](#). That means you have to provide access to the source code of the API you make available, including any changes you might have made on the original code. You also have to provide a link to the OpenFisca source code, and state its license, in a place that is easily discoverable by users of your software.

You could for example add one the following lines to a “credits” page:



Hosting an API instance

Let App Developers access your country package information and computations by serving the web API that comes bundled with the OpenFisca-Core module. See the [technical documentation](#) for serving instructions.

Track your API

If you want to track how your API is being used, you can install the [OpenFisca Tracker](#).

Endpoints

Each OpenFisca Country Package web API comes with a set of endpoints including an OpenAPI specification on the `/spec` route. You can check out the `OpenFisca-France` [swagger documentation](#) to see how their endpoints work.

The Openfisca Web API can be used to:

- access information about the parameters (e.g. `/parameters`) and variables (e.g. `/variables`) of the Country Package.
- run simulations (e.g. `/calculate`) on a specific situation. To describe a situation, learn more about the Web API [inputs and outputs](#).

Using the /calculate endpoint

All the examples provided here are from the [country package template](#).

In order to run a computation on the Web API, you will need to send information to the API concerning:

- The situation, meaning describe the entities (e.g. individuals, households) that you want to base your calculations on.
- The variable you need to compute.

Describing the situation

Describing entities

The most important rule in describing a situation in OpenFisca is:

Every person has to belong to one of each group entity (e.g. household). Every person in a group entity needs a role (e.g. parent)

For example, if you wish to run a calculation on 2 households:

- household_1 is composed of two adults;
- household_2 is composed of one adult and one child.

```
{
  "persons": {
    "Ricarda": {},
    "Bob": {},
    "Bill": {},
    "Janet": {}
  },
  "households": {
    "household_1": {
      "parents": [
        "Ricarda", "Bob"
      ]
    },
    "household_2": {
      "parents": [
        "Bill"
      ],
      "children": [
        "Janet"
      ]
    }
  }
}
```

Adding information to entities

To run a precise calculation, you can provide information on each person and group entity.

These are the input [variables](#) of your simulation.

To provide an input variable, insert the value in the json, for the corresponding time period (e.g. '2015-06') and entity (e.g. 'person', 'household').

The time period must respect the [definition period](#) of the variable, and the entity must be the one the variable is defined for.

For example, if Ricarda has a salary (defined monthly for a Person) of 3500/month until september 2016, and 4000/month after that and if household_2 were tenant and became homeowners in march 2016 (housing_occupancy_status is defined monthly for a household) of the 57 sqm apartment they live in, you would write:


```

{
  "persons": {
    "Ricarda": {
      "salary": {
        "2016-01": 3500,
        "2016-02": 3500,
        "2016-03": 3500,
        "2016-04": 3500,
        "2016-05": 3500,
        "2016-06": 3500,
        "2016-07": 3500,
        "2016-08": 3500,
        "2016-09": 4000,
        "2016-10": 4000,
        "2016-11": 4000,
        "2016-12": 4000
      }
    },
    "Bob": {},
    "Bill": {},
    "Janet": {}
  },
  "households": {
    "household_1": {
      "parents": [
        "Ricarda", "Bob"
      ]
    },
    "household_2": {
      "parents": [
        "Bill"
      ],
      "children": [
        "Janet"
      ]
    },
    "housing_occupancy_status": {
      "2016-01": "Tenant",
      "2016-02": "Tenant",
      "2016-03": "Owner",
      "2016-04": "Owner",
      "2016-05": "Owner",
      "2016-06": "Owner",
      "2016-07": "Owner",
      "2016-08": "Owner",
      "2016-09": "Owner",
      "2016-10": "Owner",
      "2016-11": "Owner",
      "2016-12": "Owner"
    },
    "accommodation_size": {
      "2016-01": 57,
      "2016-02": 57,
      "2016-03": 57,
      "2016-04": 57,
      "2016-05": 57,
      "2016-06": 57,
      "2016-07": 57,
      "2016-08": 57,
      "2016-09": 57,
      "2016-10": 57,
      "2016-11": 57,
      "2016-12": 57
    }
  }
}

```

Note that due to the default value system in OpenFisca, the variables that have not been defined explicitly are either calculated or take on their [default value](#).

Computing a variable

Once you have described the situation, you can compute all variables in the Country Package.

To indicate you want a variable computed, insert the variable in the corresponding entity and indicate the time period followed by the term

`null` .

for example, to compute Ricarda's june income tax (defined monthly for a person) and household_2's housing tax (defined yearly for a household), you would write:

```
{
  "persons": {
    "Ricarda": {
      "salary": {
        "2016-01": 3500,
        "2016-02": 3500,
        "2016-03": 3500,
        "2016-04": 3500,
        "2016-05": 3500,
        "2016-06": 3500,
        "2016-07": 3500,
        "2016-08": 3500,
        "2016-09": 4000,
        "2016-10": 4000,
        "2016-11": 4000,
        "2016-12": 4000
      },
      "income_tax": {
        "2016-06": null
      }
    },
    "Bob": {},
    "Bill": {},
    "Janet": {}
  },
  "households": {
    "household_1": {
      "parents": [
        "Ricarda", "Bob"
      ]
    },
    "household_2": {
      "parents": [
        "Bill"
      ],
      "children": [
        "Janet"
      ],
      "housing_occupancy_status": {
        "2016-01": "Tenant",
        "2016-02": "Tenant",
        "2016-03": "Owner",
        "2016-04": "Owner",
        "2016-05": "Owner",
        "2016-06": "Owner",
        "2016-07": "Owner",
        "2016-08": "Owner",
        "2016-09": "Owner",
        "2016-10": "Owner",
        "2016-11": "Owner",
        "2016-12": "Owner"
      },
      "accomodation_size": {
        "2016-01": 57,
        "2016-02": 57,
        "2016-03": 57,
        "2016-04": 57,
        "2016-05": 57,
        "2016-06": 57,
        "2016-07": 57,
```

```

    "2016-08": 57,
    "2016-09": 57,
    "2016-10": 57,
    "2016-11": 57,
    "2016-12": 57
  },
  "housing_tax": {
    "2016": null
  }
}
}
}
}
}

```

Understanding the result

The API will return an identical JSON file where all the `null` (the variable that you asked OpenFisca to compute, see above for details) have been replace by the computed value.

```

{
  "households": {
    "household_1": {
      "parents": [
        "Ricarda",
        "Bob"
      ]
    },
    "household_2": {
      "accomodation_size": {
        "2016-01": 57,
        "2016-02": 57,
        "2016-03": 57,
        "2016-04": 57,
        "2016-05": 57,
        "2016-06": 57,
        "2016-07": 57,
        "2016-08": 57,
        "2016-09": 57,
        "2016-10": 57,
        "2016-11": 57,
        "2016-12": 57
      },
      "children": [
        "Janet"
      ],
      "housing_occupancy_status": {
        "2016-01": "Tenant",
        "2016-02": "Tenant",
        "2016-03": "Owner",
        "2016-04": "Owner",
        "2016-05": "Owner",
        "2016-06": "Owner",
        "2016-07": "Owner",
        "2016-08": "Owner",
        "2016-09": "Owner",
        "2016-10": "Owner",
        "2016-11": "Owner",
        "2016-12": "Owner"
      },
      "housing_tax": {
        "2016": 570.0
      },
      "parents": [
        "Bill"
      ]
    }
  },
  "persons": {
    "Bill": {},

```

```
"Bob": {},
"Janet": {},
"Ricarda": {
  "income_tax": {
    "2016-06": 525.0
  },
  "salary": {
    "2016-01": 3500,
    "2016-02": 3500,
    "2016-03": 3500,
    "2016-04": 3500,
    "2016-05": 3500,
    "2016-06": 3500,
    "2016-07": 3500,
    "2016-08": 3500,
    "2016-09": 4000,
    "2016-10": 4000,
    "2016-11": 4000,
    "2016-12": 4000
  }
}
}
```

Note that elements might appear in a different order in the response. However the structure of the file stays the same.

Troubleshooting

When a computation raises an error or returns a wrong or strange result, you can use one of these techniques to find out the reason.

Enable the debug log

The debug log is printed by OpenFisca internals. It displays for each computed formula its inputs, its period and its result. The computed variable appears last.

It uses the `logging` module of Python and is disabled by default.

Here is how to enable it:

```
from openfisca_france import FranceTaxBenefitSystem
tax_benefit_system = FranceTaxBenefitSystem()
scenario = tax_benefit_system.new_scenario()
scenario.init_single_entity(
    period = 2015,
    parent1 = dict(
        salaire_de_base = 40000,
    ),
)
simulation = scenario.new_simulation(debug=True)
irpp = simulation.calculate('irpp', 2015)
```

It displays (not shown here entirely):

```
INFO:openfisca_core.formulas:<=> enfant_a_charge@individus<2015>(age@individus<2015>[45], handicap@individus<2015>[False], quifoy
INFO:openfisca_core.formulas:<=> enfant_majeur_celibataire_sans_enfant@individus<2015>(age@individus<2015>[45], handicap@individu
INFO:openfisca_core.formulas:<=> nbptr@foyers_fiscaux<2015>(nb_pac@foyers_fiscaux<2015>[0.0], maries_ou_pacses@foyers_fiscaux<201
INFO:openfisca_core.formulas:<=> indemnite_residence@individus<2015-01>(traitement_indiciaire_brut@individus<2015-01>[0.0], salai
[...]
INFO:openfisca_core.formulas:<=> cotsyn@foyers_fiscaux<2015>(f7ac@individus<2015>[0], salaire_imposable@individus<2015>[32353.7],
INFO:openfisca_core.formulas:<=> rfr@foyers_fiscaux<2015>(rni@foyers_fiscaux<2015>[29118.7], f3va@individus<2015>[0], f3vi@indivi
INFO:openfisca_core.formulas:<=> cehr@foyers_fiscaux<2015>(rfr@foyers_fiscaux<2015>[29118.7], nb_adult@foyers_fiscaux<2015>[1.0])
INFO:openfisca_core.formulas:<=> irpp@foyers_fiscaux<2015>(iai@foyers_fiscaux<2015>[3091.05], credits_impot@foyers_fiscaux<2015>[
```

Note: if you work in a Jupyter notebook, you can activate logging by inserting this first cell in your notebook:

```
import logging
logger = logging.getLogger()
logger.setLevel(logging.DEBUG)
```

Recipes for OpenFisca

As the OpenFisca community becomes larger, issues that only affect a small percentage of users arise.

The purpose of this section is to bring together the clever solutions the community came up with and share them with all OpenFisca users.

- [How to use OpenFisca on the web \(no installation required on your computer\)](#)
- [How to install OpenFisca in an offline environment](#)
- [How to work on OpenFisca on a Windows without being administrator](#)
- [How to test your changes on "ready to use" situations](#)

You're welcome to share your tips on how to solve technical issues! Please update [this section](#) (in english) or this [wiki FAQ](#) in your preferred language.

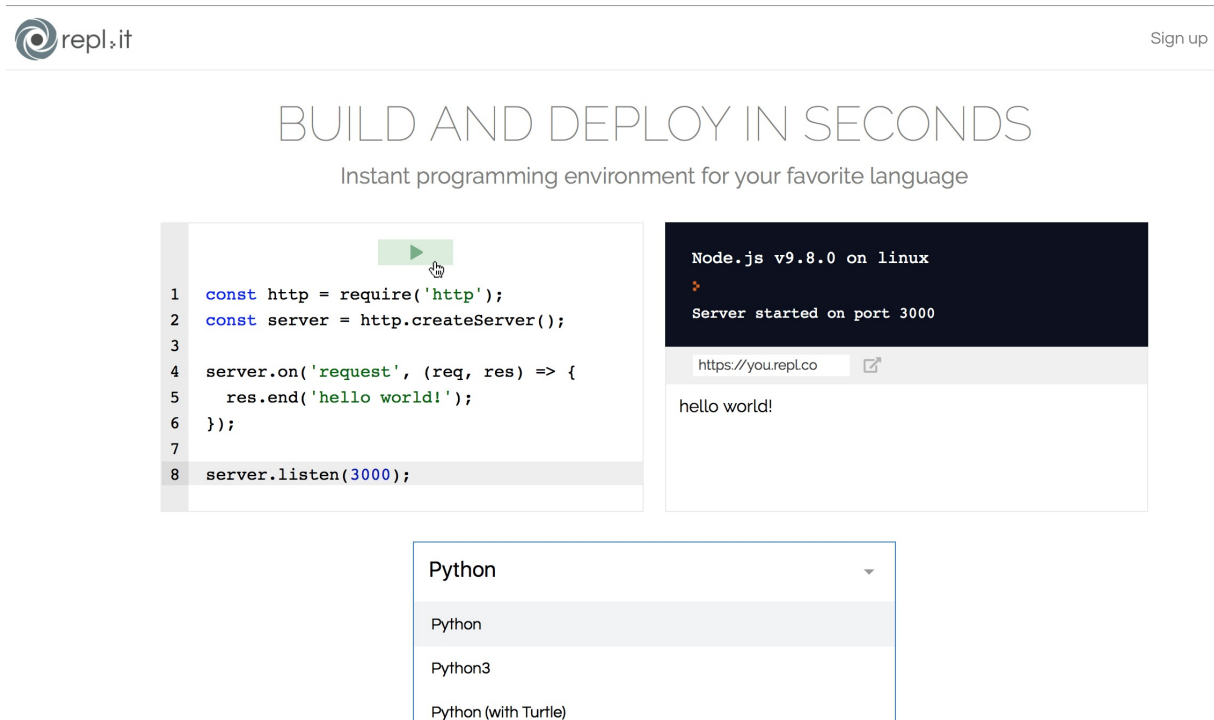
How to use OpenFisca on your browser (no installation required on your computer)

If you want to use OpenFisca for running a simulation or evaluating the impact of a reform, without installing anything in your computer, these online services allow you to run OpenFisca directly on your browser, no installation required: repl.it, [python anywhere](https://pythonanywhere.com) and [jupyterlab](https://jupyterlab.com).

Let's see how to use OpenFisca on one of those services: repl.it ☐

Instructions

1. Go to repl.it and select `Python` (i.e. `python2.7`):



The screenshot shows the repl.it website. At the top, there's a navigation bar with the repl.it logo and a 'Sign up' link. Below the navigation bar, the main heading reads 'BUILD AND DEPLOY IN SECONDS' with the subtitle 'Instant programming environment for your favorite language'. The central area is divided into two main sections. On the left, there's a code editor with a Python script that sets up a simple HTTP server. The script is as follows:

```
1 const http = require('http');
2 const server = http.createServer();
3
4 server.on('request', (req, res) => {
5   res.end('hello world!');
6 });
7
8 server.listen(3000);
```

On the right, there's a terminal window showing the output of the script. The terminal text is:

```
Node.js v9.8.0 on linux
Server started on port 3000
https://you.repl.co
hello world!
```

Below the code editor and terminal, there's a dropdown menu for selecting the programming language. The dropdown is currently set to 'Python' and shows a list of options: 'Python', 'Python3', and 'Python (with Turtle)'.

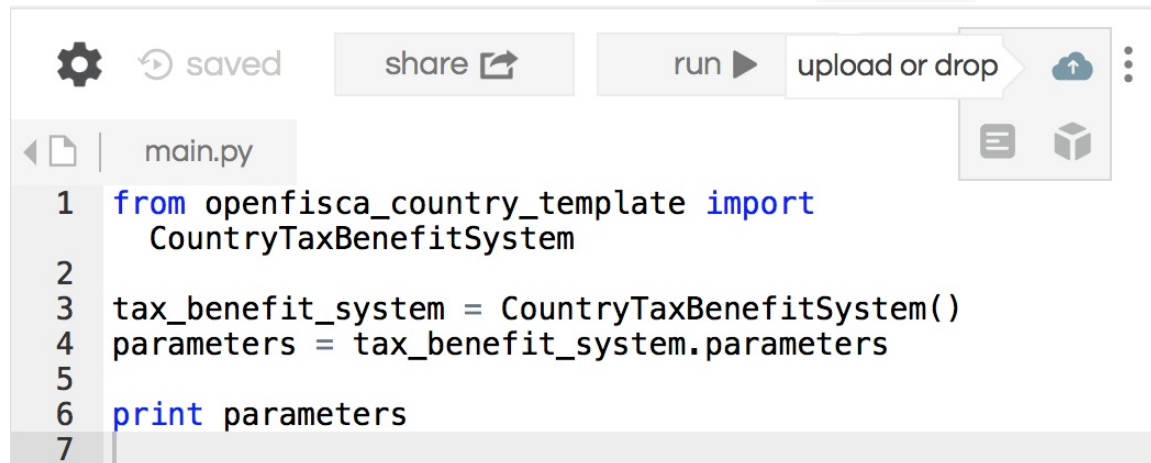
2. Write the OpenFisca code you wish to run. The default entry point is `main.py` python file. Example of `main.py` content using `openfisca-country-template`:

```
from openfisca_country_template import CountryTaxBenefitSystem

tax_benefit_system = CountryTaxBenefitSystem()
parameters = tax_benefit_system.parameters

print parameters
```

You can also import files (e.g. JSON files describing input [situations](#)) by clicking on the `import or drop` button.



```
1 from openfisca_country_template import
   CountryTaxBenefitSystem
2
3 tax_benefit_system = CountryTaxBenefitSystem()
4 parameters = tax_benefit_system.parameters
5
6 print parameters
7
```

3. Click the `run` button to execute your code

Your code dependencies are automatically analyzed and imported.

4. You're all set. Check your code results on the right sided python interpreter.

Example: evaluating a reform with OpenFisca France and repl.it

To see OpenFisca in action in your favourite browser, check out this [example of reform to the French tax-benefit system in repl.it](#) (in French)!

Installing OpenFisca in an offline environment

If you need to install OpenFisca on a server with no Internet access, here is how to do it.

The big picture: download Python packages on a machine with Internet access, copy them to the server and install them in a [virtualenv](#).

We assume that it is possible to copy files to the server, for example via an USB key. Or perhaps the server filters only outgoing connections, but accepts incoming connections allowing to copy the files.

On the machine with Internet access

We are going to create a first virtualenv in which we'll use `pip` to download the `.whl` files in a specific directory.

Here we use [pew](#) to simplify virtualenv management.

```
pip install pew
pew new openfisca-packages --python=python2.7

# Upgrade pip itself
pip install --upgrade pip
pip --version
# Should print at least 9.0 at the time we write this doc.

mkdir ~/openfisca-packages
cd ~/openfisca-packages
pip download OpenFisca-France
# You should see the downloaded files in the current directory.
```

Now copy these files on the server (say in the `~/openfisca-packages` directory), either via an USB key, or with `scp`, or any other way.

Example with `scp` :

```
scp -r ~/openfisca-packages user@server:
```

On the server

Starting from here we assume you copied the packages on the server, say in `~/openfisca-packages`.

The following commands show how to install Python packages without any Internet access. If you already have a virtualenv, activate it.

Otherwise create a new one following the same instructions as above (for example with `pew new`).

```
pip install ~/openfisca-packages/*
Processing ./isodate-0.5.4.tar.gz
[...]
Installing collected packages: pytz, Babel, Biryani, numpy, PyYAML, OpenFisca-Core, requests, OpenFisca-France, isodate
Successfully installed Babel-2.3.4 Biryani-0.10.4 OpenFisca-Core-7.0.0 OpenFisca-France-15.1.0 PyYAML-3.12 isodate-0.5.4 numpy-1.11.0

pip list | grep OpenFisca-France
OpenFisca-France 15.1.0
```

Run the basic tests which confirm that OpenFisca-France is correctly installed:

```
python -m openfisca_france.tests.test_basics
OpenFisca-France basic test was executed successfully.
```

Work on OpenFisca on a Windows without being administrator

*Warning: running OpenFisca on a machine **with** administrator privileges would make your life much easier. Using a MacOS or a Linux would be even better.*

If you do not have a choice other than using a restricted Windows, this guide sums up the "recipe" to install OpenFisca in such an environment.

1. Install git

Git is a tool that will help you version your work. It also comes with a shell **terminal** that allows you to type commands in a more standard way than the Windows command line tool.

- Download git from <https://git-for-windows.github.io>.
- Install it. While installing, keep the default options.

2. Install python

Python is the programming language used in OpenFisca. It can be installed without administrators rights through a software named Miniconda.

- Download miniconda from <https://conda.io/miniconda>. Make sure to choose the **Python 2.7** version for Windows. If you don't know if your system is 32-bit or 64-bit, pick 32-bit.
- Install it. At some point, the installer will ask you for a "Destination Folder". You can keep the default or choose another one, but in all case **copy paste the path to this folder somewhere**. It will be useful later. For instance, this path may look like `C:\Users\my-name\AppData\Local\Miniconda2`.
- Run the program "Git Bash" from the "Start" menu ("Démarrer"). This should open a command line. Copy and paste the following lines in the console, after **adapting the first line using the path you noted in the last step**:

```
echo 'MINICONDA_PATH="C:\Users\form\AppData\Local\Miniconda2"' >> .bashrc
echo 'function convert { echo /$1 | sed '\''s/\\/\\/g'\'' | sed '\''s/:/\/'\'' ; }' >> .bashrc
echo 'function add { export PATH=$(convert $1):$PATH ;}' >> .bashrc
echo 'add $MINICONDA_PATH' >> .bashrc
echo 'add "$MINICONDA_PATH/Scripts"' >> .bashrc
source .bashrc
conda create -n openfisca python=2.7 --offline --yes
echo 'source activate openfisca' >> .bashrc
source activate openfisca
```

To check that everything worked correctly, type in Git Bash:

```
pip --version
```

A version number should be printed, and no error message should appear. Congrats, you just set up a Python working environment!

3. Install OpenFisca

- Download the OpenFisca-France [installation files](#)
- Extract the content of this archive in a directory.
- Go to that directory, then to the `windows` subdirectory. If you installed Python in 32 bits, **right-click** on `32-bits`. If you installed Python in 64 bits, **right-click** on the `64-bits` subdirectory. Choose "Git Bash Here"
- Run the command `pip install *`

To check that everything worked correctly, type in Git Bash:

```
python -c "from openfisca_france import CountryTaxBenefitSystem; CountryTaxBenefitSystem()"
```

No error message should appear. Congrats, you just installed OpenFisca-France!

4. Install atom

Atom is a modern text editor that doesn't require administrator privileges to be installed. It will allow you to edit Python files with syntaxing coloring.

- Download atom from <https://atom.io/>

5. Write and run your own scripts

You can now write your own scripts, such as [this tutorial](#).

To edit a script, open it with atom.

To run it, save your modifications, go to the directory containing it, right click and chose "Gith Bash Here". Then type:

```
python name-of-the-script.py
```

In case you run into a problem, you can [open an issue](#) or send an email to contact@openfisca.fr.

How to test your changes on "ready to use" situations (for OpenFisca-France)

Often, when making changes to legislation, you need to test it on a situation that works with your country Tax & Benefit system. Sometimes, these situations can be quite complicated to model (such as roomates). Instead of re-writing them everytime, we have pre-packaged a few in a Python Package.

You can find the package along with a usage example in the [Tutorial repository](#)

Publishing results based on OpenFisca

OpenFisca is free software made available under an [AGPL license](#). This means that you are free to use, install and modify it, but that you have to contribute your changes back to the community. This is the only way a digital common can be sustainable, and we are very happy that you take part in this shared effort!

Computation results

If you provide results based on OpenFisca usage, crediting the project would be very welcome in order to increase its visibility and improve its contributor base. More people means more maintenance and more upgrades!

The following HTML snippet can be used to credit OpenFisca in your articles and publications:

English

```
<a href="https://openfisca.fr" target="_blank" rel="noopener">
  Computed by .
</a>
```

Français

```
<a href="https://openfisca.fr" target="_blank" rel="noopener">
  Calculé par .
</a>
```

Hosting an API instance

If you provide a service that uses OpenFisca, either bundled in a program or serving it over the network, be it directly or wrapped through another API layer, you have to credit OpenFisca, give a link to its source code and license.

The following HTML snippet can be used to credit OpenFisca, for example in your footer or “about” page. Please make sure to update the destination of the source code link if you use a modified version!

English

```
<span>Computations powered by
  <a href="https://openfisca.fr" target="_blank" rel="noopener">source code</a>
  is used under an <a href="https://choosealicense.com/licenses/agpl-3.0/" target="_blank" rel="noopener">AGPL</a> license.
</span>
```

Français

```
<span>Calculs fournis par
  <a href="https://openfisca.fr" target="_blank" rel="noopener">code source</a>
  est utilisé sous licence <a href="https://choosealicense.com/licenses/agpl-3.0/" target="_blank" rel="noopener">AGPL</a>.
</span>
```

Changes

If you modify or extend OpenFisca, you are [legally required](#) to make those changes available to the community. The easiest way to do it is to publish your fork, extension or reform on a source hosting platform such as GitHub, and to [notify](#) the core team of this publication.

Please note that, under AGPL provisions, serving over the network is considered as publishing changes, which means you also have to make your changes available even if you don't keep the running instance on your servers.

Community

Slack

The OpenFisca community gathers around a [Slack space](#), which you can ask to join by sending a mail to contact@openfisca.org.

This space provides both community and official support, and centralises all countries' channels.

Channels and naming conventions

In order to increase discoverability of channels and ease navigation, the following prefixes are used:

- `of-` channels are about quick discussions and requests for help on a technical module. To make decisions on changes to apply to these modules, we use GitHub issues and Pull Requests on dedicated repositories.
- `share-` channels are newsrooms on which everyone is encouraged to share their news, learnings and accomplishments :)

For tax and benefit systems models, the following conventions are applied:

- Channels that centralise discussions around a specific tax and benefit system are given the name of the distributed module, suffixed by `-system`. *For example:* `france-system`. If that full name is too long for the Slack channel character limit, then a shortened version of the name is used.
- Channels that centralise discussions around extensions to tax and benefit systems are given that system's module name, followed by `-ext-` and an identifier for that extension. *For example:* `france-ext-paris`.
- System-specific group channels are campfires around which some specific organisations of contributors to a tax and benefits system gather. When a country becomes large enough, it often happens that several employers of contributors work concurrently on different parts of the system, and the main `-system` channel would become unreadable. These channels are called that system's module name, followed by `-org-` and an identifier for that group. *For example:* `france-org-gouv`.

Contact

You can contact the OpenFisca maintainers through:

- [GitHub](#) if you have any technical issue.
- Twitter [@OpenFisca](#) for general inquiries and feedback.
- [email](#) for collaboration opportunities.

Project history

The development of OpenFisca began in May 2011 at the [CAS](#) (renamed France Stratégie / Commissariat général à la stratégie et à la prospective in April 2013) with the support of the [IDEP](#).

OpenFisca was originally developed as a desktop application using the [Qt](#) library with a Python API. This original source code was released under a free software license in November 2011.

In the early 2014, [Etalab](#) started using OpenFisca and soon became a major contributor. It then decided to:

- separate the computing engine from its desktop user interface;
- offer a web API in addition to the Python API;
- demonstrate the value of the web API by developing sample applications including a web interface to simulate personal cases;
- offer a public access to this web API;
- stop the development of the [Qt version](#).

The core was improved extensively by [Etalab](#), while the French model was being improved and updated by the [CGSP](#) with the help of the [IDEP](#) and the [IPP](#), soon to be joined by the French [State Startups incubator](#) which used and extended the model for digital public services

purposes.

The 2016 [OGP](#) Paris summit saw a demonstration that a small team could model the base of a tax and benefit system from scratch under 36 hours, when the [Sénégal](#) income revenue tax was modelled and made usable with a web UI during the OGP hackathon, winning the team the first prize.

This led in 2017 to a joint effort from Etalab and beta.gouv.fr with a major focus on stability, ease of contribution and reusability. This led to the full rewrite of the documentation, the opening of [openfisca.org](#) to replace openfisca.fr, and the addition of new contributors from other French agencies, as well as international reusers with [Barcelona](#) joining [Tunisia](#).

Contribute

OpenFisca is a free software project and contributors are very welcome!

Feel free to fork the [source code repositories](#) on GitHub and send us pull-requests.

You can [contact the community](#) to ask for help.

Thanks for enhancing OpenFisca anyway!

Why contribute to OpenFisca?

OpenFisca is a project being developed under the GPLv3 license or later. The source code is freely available and modifiable.

We encourage users to send their comments and suggestions for improvement, and to report any inaccuracy or error they might have found. If you want to participate more actively in its development, know that there are multiple ways contribute to the OpenFisca project.

How to contribute?

Use the API and direct its development

- Share your uses: you are welcome to keep us informed of the uses you make of the API including visualizations you may create. We'd love to be able to include them on the OpenFisca website.
- Suggest features: please tell us about the improvements to the API you would like to see, so that we can make it meet your needs.
- Participate directly in the [API's development](#).

Test and report errors (web API)

You can contribute to the development of OpenFisca by reporting errors you would find on the calculation of benefits and taxes.

To enable the OpenFisca developers to solve your problems quickly, please follow these few steps:

- try to create a minimal standard case that generates the error;
- verify [that this error is not already listed](#);
- try to identify the source of the error by inspecting [the formulas for the different benefits and taxes](#);
- [report the error](#), with as much information as possible. If possible, please provide the code that allows to reproduce the error or the JSON file of the standard case you created.

Complete the implementation of the French tax and benefit system

Some pieces of legislation are not yet integrated. Given the magnitude of the task, our ambition is to build a community of developers, economists and experts on taxes or social benefits to maintain and improve the software. You can help by following these steps:

- identify the incomplete or missing taxes or benefits;
- gather the necessary documentation to fix this issue;
- propose patches that implement the incomplete or missing benefits and taxes on [GitHub](#).

Write some legislation

From the point of view of someone (developer, economist, etc.) who wants to implement a part of the legislation, for example a new benefit, here are some key steps:

- understand the part of the legislation you want to implement
- identify the variable dependencies using the [legislation explorer](#)
- identify the new variables you need to implement

- write the new variables with their formulas, and make sure their names respect the guidelines you can find [here](#).
- store the new parameters
- if you implement a part of the official legislation, your code should go in OpenFisca-France, but if you implement a new idea or a future reform, your code should go in a reform.

Write reforms

Enhance other projects linked to OpenFisca

You can also participate in [other projects](#) that make use of OpenFisca.

Contributor guidelines

The OpenFisca project follows the [GitHub Flow](#).

Each Python package uses [Semantic Versioning](#).

Opening issues

Each OpenFisca repository has its own issues. See [OpenFisca repositories](#).

- Describe what you did.
- Describe what you expected to happen.
- Describe what happened.
- Include (or link to) any data that can help reproduce the issue you encountered.

Contributing to the code

Writing code

- If you modify/create/delete a simulation variable, please follow the [commit message rules](#).
- When adding new variables, please consider the [naming guidelines](#).
- Your code should be tested, if feasible:
 - bugfixes should include regression tests
 - new behavior should at least get minimal exercise
- Use atomic commits, in particular try to isolate "code-cleanup" commits

Opening a Pull Request

- All code contributions are submitted via a Pull Request towards `master`. The `master` branches are thus [protected](#).
- Opening a Pull Request means you want that code to be merged. If you want to only discuss it, send a link to your branch along with your questions through whichever communication channel you prefer.
- If the Pull Request depends on another opened Pull Request on another repository (like OpenFisca-Core/OpenFisca-France), the requirements should be updated in the dependent project via its `setup.py`.

It is considered a good practice to begin the name of the pull request with a verb in the present imperative tense:

```
# Good
Propose a new reform according to the French finance bill 2018

# Bad
new reform PLF 2018
```

Merging a Pull Request

Continuous integration

Before allowing you to merge a PR, the continuous integration server will ensure that:

- The automated tests are passing (they are triggered automatically and result is visible from the Pull Request page).
- The semantic version number has been updated. Check the [semantic versionning guidelines](#) to know more about how to increment the version number.
- The `CHANGELOG.md` has been updated. Make sure to briefly summarize your work, and to **mention any non backward-compatible changes**.

Web API version number

Due to a `pip` limitation, it is required to increment the major version number of OpenFisca-Web-API when it is adapted to a new major version of OpenFisca-Core. This rule avoids installing a version of OpenFisca-Core incompatible with the loaded country package (for example OpenFisca-France).

See also:

- [this old `pip` issue](#)
- [the `issue` leading to this decision](#)

Peer reviews

Pull requests should generally be **reviewed** by someone else than their authors.

This is mandatory for:

- Any Pull Request with **breaking changes** on `openfisca-france` , `openfisca-web-api` .
- Any Pull Request bringing **new features**, if these features are not relative to a specific scope.
 - Adding a new route to the API **requires** a review.
 - A review is yet not mandatory to add a new formula to social contributions in `openfisca-france` . It is though recommended.

To help reviewers, make sure to add to your PR a **clear text explanation** of your changes.

In case of breaking changes, you **must** give details about what features were deprecated. You must also provide guidelines to help users adapt their code to be compatible with the new version of the package.

Language

The development language is English. All comments and documentation in common repositories should be written in English, so that contributions can be made by developers around the world.

Country-specific repositories

Some repositories define the tax and benefit system of a specific country. In such cases, the language used throughout issues and pull requests should be one of the native ones of that country.

For instance, `openfisca-france` issues and pull-requests should be written in French.

The end goal is always to maximise contributions and collaboration. In that case, the main contributors will be experts from that country, and English should not be a barrier to entry.

Commit messages

Variable name changes

To avoid Openfisca users to be surprised by a non expected variable renaming breaking their code, we use standard commit messages when renaming a variable. The syntax is formalized below. It must be respected precisely, to allow automatic information extraction.

Renaming

Renaming one or several variables will be notified by a commit message with the following syntax, **on one independant line per renamed variable**:

```
Rename former_name to new_name
```

No other information must appear on this line.

Introducing

Introducing one or several new variables will be notified by a commit message with the following syntax, **on one independant line per created variable**:

```
Introduce new_name
```

No other information must appear on this line.

Deprecating

If a variable must not be used anymore, it will be notified by a commit message with the following syntax, **on one independant line per deprecated variable**:

```
Deprecate former_name
```

No other information must appear on this line.

Openfisca variables naming guidelines

General philosophy

If you consider naming variables, you are in a country-specific repository, where the [local language rule](#) apply. The domain language is thus one of the native ones of the modeled country. We consider each tax, collecting organism and country regulation as a domain-specific term. In the same fashion, well-known abbreviations of these domain-specific terms are accepted.

OpenFisca variables names should, as much as possible, be understandable by an external contributor who is **curious** about the country tax and benefits system, **without necessarily being an expert**.

One should be able to get a rough idea of the meaning of a variable by reading its name, or by quickly researching it on the web.

A particular effort should be made on variables that are likely to be reused.

Examples:

Good naming

`als_etudiant` : I don't know what `als` stands for. I look it up on a search engine, and I see ALS are a form of Aides Logement. I thus know this variable should be the amount of ALS for a student. This is enough to tell me if it is interesting in my context.

Bad naming

`apje_temp` : I could find the meaning of APJE online, but the temp suffix remains a mystery.

`rto_net` . I can guess it's an amount after some kind of deduction, but looking RTO on a search engine doesn't give me anything.

Do's and don'ts

Acronyms

Acronyms are ok as long as they are broadly accepted and their meaning is quickly findable online.

OK: RSA, RFR

KO: PAC

Abbreviations

Abbreviations should be avoided unless they are undoubtedly transparent.

OK: nb_parents

KO: nb_par, isol

Scopes and prefixes

To show a variable belongs to a specific scope, it is better to use a prefix rather than a suffix.

OK: rsa_nb_enfants

KO: nb_enfants_rsa

Not specifying the scope of a specific variable should be avoided, as it is confusing for other users.

OK: ir_nb_pac

KO: nb_pac

Entity suffixes

It happens that several variables have the same meaning, but for different entities (individus, familles, etc.). Standard suffixes should be used to distinguish them.

OK: ass_base_ressources_individu, statut_occupation_logement_famille

Legacy

Many variables on the current codebase of OpenFisca France do not respect the guidelines presented here. An exhaustive and global renaming is not considered as of today.

However, new variables should be compliant with these guidelines, and legacy ones should progressively and opportunistically be renamed.

Semantic versionning guidelines

Before merging your contribution to an openfisca package, you are required to increment the version of this package.

The [semantic versionning convention](#), applied here, requires you to:

Given a version number MAJOR.MINOR.PATCH, increment the:

MAJOR version when you make incompatible API changes,

MINOR version when you add functionality in a backwards-compatible manner, and

PATCH version when you make backwards-compatible bug fixes.

It is thus crucial to determine whether your changes are **backwards-compatible**. If, during a hackathon, a contributor has written a reform to Openfisca, would this reform still work after adding your changes ?

Examples in Openfisca context

Country package (e.g. openfisca-france)

Patch

- Correcting an error in a formula.
- Correcting the value of a parameter.

Minor

- Introducing a new formula.
- Introducing a parameter.

Major

- Renaming or deprecating a variable.
- Changing the default value of a variable.
- Deprecating a parameter.
- Changing the sctructure of the parameter tree.

About this documentation

This documentation is built with the excellent [GitBook](#) tool (see [GitBook documentation](#)).

It is written in [Markdown](#) and the source is hosted on this GitHub repository: [openfisca/openfisca-doc](#).

Collaborative editing

Everybody can participate to the redaction of the documentation.

On each page there is a link named "Edit this page". Just click on it and you'll jump on GitHub on the Markdown source file of the page. Then edit the file as explained on this GitHub documentation page: [editing-files-in-another-user-s-repository](#).

Then save the file and create a [pull request](#) which will be accepted if relevant.

Build it yourself

If you'd like to build it by yourself to work locally, here are the steps.

```
git clone git@github.com:openfisca/openfisca-doc
npm install
```

Then you can either build the documentation or launch a local HTTP server with watch mode:

```
npm run build
or
npm run watch
```

With watch mode, open <http://localhost:2050/> in your browser once the first build is done.

Deploy (for maintainers)

To deploy the documentation just push on the `master` branch.

OpenFisca extensions

Extensions allow you to define new variables or parameters for a tax and benefit system, while keeping their code separated from the main country package. They can only *add* variables and parameters to the tax and benefit system: they cannot *modify* or *neutralize* existing ones.

They are for instance used to code local prestations.

Extensions are sometimes confused with another mechanism: reforms. [Read more about their respective uses.](#)

Extensions can be manually loaded to a tax and benefit system using the `load_extension` method.

Extension architecture

The architecture of an extension folder is the following:

```
{extension_name}/ # The folder name is by convention the name of the extension.
  {extension_name}/__init__.py # Empty file.
  {extension_name}/{some_formula}.py # File containing formulas
  {extension_name}/{other_formula}.py
  {extension_name}/parameters # Optional parameters directory.
  {extension_name}/parameters/{new_tax}
  {extension_name}/parameters/{new_tax}/{rate}.yaml
  {extension_name}/tests/{some_formula}.yaml # Optional test files
  {extension_name}/tests/{other_formula}.yaml
```

All python files located directly in `{extension_name}/` are imported in the tax and benefit system.

The syntax of the formulas within extension python files is the same than in the general country package formulas (e.g. `from openfisca_france.model.base import *`).

Variables inside an extension should not have the same name than any existing formula, nor than any formula in another extension being used.

Developer guide

Source code repositories

The OpenFisca project is distributed across many Git repositories:

- [OpenFisca-Core](#)
- [OpenFisca-France](#)
- [OpenFisca-Web-API](#)
- [OpenFisca-Web-UI](#)

Debugging code

If you install [ipdb](#) (`pip install ipdb`) the API server will drop you into a debugger when an exception occurs:

```
$ paster serve --reload development-france.ini
Starting server in PID 3815.
serving on 0.0.0.0:2000 view at http://127.0.0.1:2000
> /home/harold/Dev/openfisca/openfisca-web-api/openfisca_web_api/model.py(52)get_cached_composed_reform()
51
---> 52     full_key = '.'.join(
53         tax_benefit_system.full_key + reform_keys

ipdb> tax_benefit_system
<openfisca_web_api.environment.TaxBenefitSystem object at 0x7f7eb8e88d10>
ipdb> tax_benefit_system.full_key
u'paris'
ipdb>
```

Profiling code

To profile the execution of a portion of code, wrap it with these lines:

```
+ import cProfile
+ pr = cProfile.Profile()
+ pr.enable()

    [...portion of code...]

+ pr.disable()
+ pr.dump_stats('result.profile')
```

Each time you call the endpoint a `result.profile` file is written. To prevent it to be overwritten, generate a dynamic name with `tempfile.mkstemp` .

Then you can use the [runsnakerun](#) GUI to inspect the profile data.

Under Debian GNU/Linux:

```
aptitude install runsnakerun
```

Tests

OpenFisca has three sorts of tests:

- unit tests
- test-case tests
- scenario tests

Run tests

OpenFisca uses [nose](#) to run its unit tests. Here are some useful commands.

- Run the whole test suite:

```
make test
```

which is available at least in Core, France and Web-API repositories.

- Run a specific test:

```
nosetests openfisca_france/tests/test_parameters.py
```

- Hide log of failing test:

```
nosetests --nologcapture openfisca_france/tests/test_parameters.py
```

- Display log of successful test:

```
nosetests --debug=openfisca_core openfisca_france/tests/test_parameters.py
```

YAML tests

Formulas are tested with [YAML tests](#).

ipdb debugger

If a test fails, you can execute it with the [debug](#) nose plugin:

```
nosetests --pdb openfisca_core/tests/test_tax_scales.py
```

You'll be dropped in the `pdf` debugger shell when an error occurs.

You can [specify the exact test to launch](#):

```
nosetests --pdb openfisca_core/tests/test_tax_scales.py:test_linear_average_rate_tax_scale
```

The [nose-ipdb](#) plugin is more user-friendly (because it uses the [ipdb](#) debugger instead of `pdb`). In this case, just use the `--ipdb` option rather than `--pdb`. See also the `--ipdb-failure` option.

In case you want to set a breakpoint manually, in order to enter the debugger shell before an errors occurs, copy-paste this line in your code:

```
import nose.tools; nose.tools.set_trace(); import ipdb; ipdb.set_trace()
```

This needs [ipdb](#) to be installed.

Hint: use the snippets feature of your favorite text editor to save this line, for example give it the name "breakpoint".

Travis automated tests

OpenFisca uses [Travis CI](#) to run tests automatically after each `git push`.

The repositories tested by Travis are:

- [OpenFisca-Core](#)
- [OpenFisca-France](#)
- [OpenFisca-Web-API](#)

The OpenFisca website hosts a summary page of the build statuses: <https://www.openfisca.fr/build-status>

Travis tests other git branches than `master` too. For example: [OpenFisca-Core](#).

For OpenFisca-France, when testing a branch, if there is a branch in OpenFisca-Core with the same name, Travis will checkout it before running the tests. This is done by [this script](#).

Idem for OpenFisca-Web-API with [this script](#) which is slightly different because it handles more dependencies.

Ludwig tests

To download tests from [Ludwig](#) (the tests tool from [Mes aides](#)), see the script [download_mes_aides_tests.py](#).

Release process

Automated releasing

The main openfisca packages (core, france, and web-api) are today **continously** and **automatically** released.

When a Pull Request is merged to master, this CI server (travis) automatically:

- Publish a version tag on github.
- Publish a release on the [PyPI](#) repository.

Set up automated releasing on a new repository

- Make sure you have installed the [Travis CI Command Line Client](#)
- Log in to Travis CI using your github credentials:

```
travis login
```

Automated tagging on git

- Create a **new** ssh key in your repository. Do **not** push it:

```
ssh-keygen -t rsa -b 4096 -C "bot@openfisca.fr"
```

- Encrypt the private key.

```
travis encrypt-file $id_rsa --add
```

- Add the public key to the repository in Settings > Deploy keys, with writing rights.
- Copy and **adapt the repository name** in the `release-tag.sh` script into the repository.
- Edit `.travis.yaml` to:
 - Decrypt the ssh key in `before_deploy` instead of `before_install` .
 - Add the [deployment instructions](#).
- Only push the encrypted key. You can remove both the private and public keys after making sure the automatic tagging works as expected.

Automated releasing on Pypy

- Make sure you know the credentials for the `openfisca-bot` Pypi user.
- Give `openfisca-bot` maintainer's right on Pypi.
- Encrypt the `openfisca-bot` password:

```
travis encrypt
```

- Edit `.travis.yaml` to:
 - [Compile](#) the [internationalization](#) catalog in `before_deploy` , if needed.
 - Add the [deployment instructions](#). Replace the encrypted password by what `travis encrypt` generated.

Deprecated

Manual releasing

Here are the steps to follow to build and release a Python package. Execute them on each Git repository you want to publish:

- [OpenFisca-Core](#)
- [OpenFisca-Web-API](#)

See also:

- [PEP 440: public version identifiers](#)
- [distributing packages guide](#)
- [setuptools](#)
- [semver](#)

Steps to execute

Tests

Open the [build-status](#) page to check that the build status of every project is "passing" (green color).

Check that there are no pending tests by clicking on the badges.

If there are errors, click on a badge to open the corresponding Travis page.

You can also execute the tests by yourself: do `git pull` in every project on the branch `master`, and run `make test` in each project.

Internationalization (i18n)

If the project does not use [Python-Babel](#) to translate strings, skip this section. Check by looking for the `Babel` entry in `install_requires` = of `setup.py`.

Extract strings to translate from source code to `.pot` file and update `.po` catalog files:

```
python setup.py extract_messages update_catalog
```

Translate `.po` catalog files using [poedit](#) for example:

```
poedit /path/to/i18n/fr/LC_MESSAGES/fr.po
```

Ensure that `Project-Id-Version` in `.pot` and `.po` files are correct.

Commit modified files if needed:

```
git commit -am "Update i18n translations"
```

Compile catalog `.po` files:

```
python setup.py compile_catalog
```

It should display `(100%) translated`.

Create the release commit

In `setup.py` check the `install_requires` and `version` keys.

Each Pull Request introducing a new dependency or a breaking change should have updated the `setup.py` file, but it's better to check it again.

The same for the `CHANGELOG.md` file, check that it's OK.

If the `CHANGELOG.md` file is not enough filled-in, you can use this command to extract some useful commit messages:

```
git log --pretty=format:"* %s" a..b
```

Commit changes if you made some.

Then create a release tag:

```
# Replace X by the actual version number!
git tag X
git push origin master X
```

Publish on PyPI

Build and upload the package to PyPI:

```
python setup.py bdist_wheel upload
```

Test the package installation

Let's check if the package is installable from PyPI without errors using [virtualenv](#):

```
cd /tmp
virtualenv test-openfisca
cd test-openfisca
source bin/activate
pip install OpenFisca-France
python -m openfisca_france.tests.test_basics
deactivate
```

Test OpenFisca-Web-API installation if you wish.

Next steps

Do the same for the remaining repositories to release.

When all the suited repositories are released, you can announce the new release on the website news, Twitter, the mailing list, etc.