Demo Documentation

Release 0.1.3

Andras Gefferth

Nov 04, 2021

CONTENTS

The primary purpose of autocalc is to foster the creation of interactive apps from your Jupyter notebooks by providing a framework to setup the dependencies between your variables and ensuring they are kept in sync. Although other similar framework exist, autocalc shines in its simplicity and ease-of-use. It is targeted mainly at data-scientist who would like to turn their notebook into an app without having to learn external libraries.

CHAPTER

ONE

GETTING STARTED

1.1 Introduction

1.1.1 Purpose

The purpose of the autocalc package is to help the creator of Python-based interactive app developers (primarily those using Jupyter notebook) by providing a framework to set-up the dependencies between their internal and external variables.

Ok, now what does this mean? Building an app requires interaction from the user. Output values (numbers, graphs, tables, etc.) are based on the input parameters either directly or indirectly through other variables, hidden from the user. As long as the user specifies the variables in their "proper order" all other variables can be calculated in the order which is specified by their dependencies, or dependency-graph, if you like. But what happens if user changes the value of an already defined input parameter? All other variables (internal, or public) which depend on it either directly or indirectly need to be recalculated.

Without the use of autocalc or a similar framework the user would need to implement tons of callback functions , which would need to be maintained whenever new variables are introduced in the system. Note, that the code also needs to properly update variables which depend indirectly on the input parameters. Coding and maintaining these callback functions can easily become a nightmare.

That's where autocalc comes to help. Define the depencies and the functional relationships between your variables using a very simple construct and let the framework take care of keeping your variables in sync. Think in terms of "what", not "how"! Build interactive apps with with minimal (even zero) callback functions!

1.1.2 Do you know Excel?

Think about Excel and other similar spreadsheet apps. The main idea is to make the value of cells depend on other cells. You don't need to write any VBA to keep those cells in sync! Same with autocalc: set up the dependency graph and let the library take care of the rest!

1.1.3 Why autocalc?

Although other similar frameworks exist, where autocalc shines is its simplicity: use a single language construct to set up the dependency graph.

1.2 Install

autocalc can be installed using pip:

```
pip install autocalc
```

1.3 Hello World Example

- Note, that interactivity does not work in the static HTML docs.
- You can download the source of this page as a Jupyter notebook using the "View page source" or "Edit on GitHub" link in the upper right corner.

[1]:
```
from autocalc.autocalc import Var
import ipywidgets as widgets
```

[2]:
```
name_input=Var('Your name', description='Enter your name', widget=widgets.Text())

def greeting_text(name):
    return f'Hello {name}'

greeting = Var('Greeting', fun=greeting_text, inputs=[name_input], widget=widgets.Text(),
↪ read_only=True)
```

[3]:
```
display(name_input)
display(greeting)
```

```
HBox(children=(Button(description='Your name (?)', disabled=True,␣
↪style=ButtonStyle(button_color='lightgreen')...
```

```
HBox(children=(Button(description='Greeting', disabled=True, style=ButtonStyle()),␣
↪Text(value='', disabled=Tru...
```

[4]:
```
display(greeting.widget)
```

```
Text(value='', disabled=True)
```

CHAPTER

TWO

USER GUIDE

2.1 Concept

The idea of the autocalc framework is to build a dependency-graph from your variables. Assume you are building some document retrieval app, where users can first select the year of the document, then based on the year they can select the relevant topic from the list of topics available for the given year and then they can select the document itself.

In a linear workflow you can assume that users follow the above steps in the above order. And this is what they will do when they start to work with the app. But when they are working with the app they might just go back and select a different year. In this case, the previously generated list of topics is no longer valid, it needs to be reloaded.

The autocalc framwork lets you set up a logical relationship between the document year and the list of relevant topics, so that the latter gets reloaded when the former is changed.

Why do we call it a dependency graph? Because in any non-trivial system we won't just have two variables, but much more. There can be a complex dependency structure between those variables: variable A may depend on B which depends on C, so as a result A also depends indirectly on C. So when C changes, both A and B need to be recalculated.

The idea concept may be very similar to you if you work in Excel: If you set a formula for one cell, this cell will be automatically updated when the value of any cell in the formula changes.

2.2 The Var object

Fortunately for you, the autocalc package is very easy to use: There is one single class (`Var`) which you need to learn in order to utilize its features.

Var is a wrapper around any variable which you would like to be part of the dependency-graph. In addition to holding the variable, it stores all the necessary information to describe the position of the variable in the graph (ie. its neighbors upstream and downstream) some meta-data (e.g. default value) and the corresponding widget if any.

2.2.1 The task

So let's dive into it and build an app to calculate the roots of a quadratic equation: $ax^2 + bx + c = 0$. During this process we will introduce the *Var* object.

Remember, that the solutions are given by:

- $x_1 = \frac{-b+D}{2a}$, and
- $x_2 = \frac{-b-D}{2a}$, where
- $D = \sqrt{b^2 - 4ac}$.

In this example we will concentrate on real roots only, excluding complex ones. First we will give a "widget-less" solution, where we will use pure Python methods to manipulate the values. This will enable us to introduce the "low-level" interface to *Var* objects. Adding interactivity through widgets is straightforward and will be shown in section *Using widgets to interact with autocalc*.

2.2.2 Imports

As far as the autocalc package is concerned we need to import the *Var* object only:

```
from autocalc.autocalc import Var
```

For this particular example we additionally need:

```
import math
```

2.2.3 Input variables

The variables *a*, *b* and *c* are pure input variables, they do not depend on others, so they can simply be defined as:

```
a = Var('a', initial_value = 1)
b = Var('b', initial_value = -3)
c = Var('c', initial_value = 1)
```

The first, positional, argument to *Var* is its name. You can give arbitrary names to the Vars, they don't need to be distinct. They are only used when displaying the Var. All other arguments are keyword-only.

2.2.4 Calculated variables

The next Var we will introduce is *D*. It is calculated from *a*, *b* and *c* by the following function:

```
def Dfun(a, b, c):
    try:
        return math.sqrt(b*b-4*a*c)
    except ValueError:
        return math.nan
```

The corresponding Var is defined as:

```
D = Var('D', fun=Dfun, inputs=[a, b, c], read_only=True)
```

For *D* we had to define the function, which is used to calculate it, as well as the input Vars which need to be bound to the function. The *inputs* argument to *Var* is a list, the order needs to match the order of the positional arguments of the function.

2.2.5 Read-only variables

Note the we set *D* to be a *read-only* variable. It means, that its value can change, but only through its inputs; it is not allowed to change its value directly. At first it may seem obvious that if a value is a function of other variables then it should be read-only. However, there may be cases when the function value only gives a "default" value, which the user may override. For example: in your app you load a custom configuration file and you set your *custom_configuration_xml* variable accordingly. It is natural to set your *use_custom_configuration* boolean variable to *True* whenever the *custom_configuration_xml* variable changes as you can assume, that the user set a custom value in order to use it. So the *use_custom_configuration* variable depends on *custom_configuration_xml* variable trough some function which e.g. checks the correctness of *custom_configuration_xml* and returns *True* if it succeeds. This will assure that *use_custom_configuration* will be reset to *True* whenever *custom_configuration_xml* changes, as we can assume that the user just set it in order to use it. However you want to give the user the chance to switch off the usage of the custom configuration without having to manipulate the *custom_configuration_xml* variable. So in this case *use_custom_configuration* is the result of another variable but can also be overwritten manually.

The reason why we introduced *D* is to share the results between the two solutions: x_1 and x_2. (Actually the real reason is to show you how this can be done :))

2.2.6 Solutions

The two solutions (x_1 and x_2) can also be defined with the help of calculation functions and their corresponding inputs as:

```
def x1fun(a,b,D):
    return (-b-D)/2/a
def x2fun(a,b,D):
    return (-b+D)/2/a

x1 = Var('X1', fun=x1fun, inputs=[a, b, D], read_only=True, description='The first␣
↪solution of the equation')
x2 = Var('X2', fun=x2fun, inputs=[a, b, D], read_only=True, description='The second␣
↪solution of the equation')
```

So, as you can see, x_1 and x_2 depend on *a*, *b* and *D* and through *D* also indirectly on *c*. Also note, that we can add a descritpion to the Var, which will be used when displaying the Var in Jupyter notebook. See *Using widgets to interact with autocalc*.

2.2.7 Reading and writing the variables

So by now we've set up our variables, but how do we give values to them and how do we read their values? This would be trivial if we used *widgets*, but this is not the only way we can do these operaions.

Reading and writing the variables is achieved through the *.set* and *.get* methods:

```
a.set(10)
b.set(-12)
print(x1.get())
print(a.get()*x1.get()*x1.get() + b.get()*x1.get() + c.get())
```

We also need to mention the *.recalc()* method which will force a recalculation of the value, even if the value is already calculated. This may be useful for those "non read-only" Vars which depend on other Vars and we would like to reset their "default" value.

We also need to mention the optional output variable *undefined_inputs* of the *.get()* method. If a set is passed as input then it will collect all other Vars on which directly or indirectly this Var depends but are in an *Undefined* state and therefore do not allow the calculation of this Var. This can be used for "debugging" purposes on the user level, i.e. to give a meaningful message to the user as to why the calculation failed.

2.3 Using widgets to interact with autocalc

Using widgets is relatively straightforward. At the moment autocalc only supports Ipywidgets. Simply define the value of the *widget* parameter when defining your Var:

```
name_input=Var('Your name', description='Enter your name', widget=widgets.Text())
```

It is your responsibility to display the widget when and where you wish, which you can refer to as *name_input.widget*.

There is one feature, which autocalc provides: you can *display* the Var itself, like:

```
display(name_input)
```

This will display not only the widget, but will also put a label in front of it showing its name and its optional description when hovered over.

If you would like to get access to the set of widgets display by the *display* function you can get it with the *.w* member. E.g.:

```
display(HBOX([some_other_widget, name_input.w]))
```

When using the widgets, you don't need to use explicit *.set()* and *.get()* methods as they are handled by autocalc. Except when using *Lazy variables*, in which case the *.get()* method needs to be called explicitly.

2.3.1 Undefined values

Note, that the state of the Var may be *undefined*. This state can not be intuitively represented by most of the widgets. E.g. an empty TextBox can't be distinguished from an undefined one. But be assured, the state of the Var will be undefined.

2.4 Advanced features

2.4.1 Lazy variables

In many cases there are some variables, which depend on more than one input variable and are slow to calculate. In these cases it is usually not the best approach to recalculate them every time when one of their input changes. It is a waste of resources and gives a frustrating experience to the user.

We will can flag such variables with the "lazy" flag, by setting their *lazy* flag to True:

```
my_slow_var = Var('MY_VAR', fun=my_fun, inputs=[a, b], lazy=True)
```

For such variables, the function will only be evaluated once the user calls the *.get()* method explicitly. In an interactive Jupyter interface this can easily be achieved by adding a "Calc" button, whose action is to call *.get()* on the relevant Var.

Another option is to use the `autocalc.tools.PreviewAcc` class, which is derived from *ipywidgets.Accordion* and does a recalculation when the Accordion is opened.

Note, that although lazy variables are not recalculated when one of their inputs change, leaving them with the previous value would result in an inconsistent state. In this case this vars will be invalidated and assigned an "undefined" state, which brings us to the next topic:

2.4.2 Undefined variables

The Vars you defined do not necessarily have a value. For example they may be input variables, where user input is expected, or lazily calculated variables before triggering their recalculation. Such variables are in an undefined state (my_var.is_undefined() evaluates to True). Ideally all values are defined before they are being used, but is in everyday situation that user skips the setting of some relevant variables and so the calculation cannot be performed.

In order to be prepared for such cases in our UI we can either directly inquire the relevant variables and check their state through the *.is_undefined* method, or more conveniently can pass in a Python *set* object as the optional *undefined_inputs* parameter of the *.get()* method, which will collect all necessary but undefined objects.

Note, that a Var can be explicitly invalidated by using the *.clear()* method.

By default, the calculation of a Vars function is not triggered when any of its inputs is not defined. However, there may be cases, when depending on the value of some other parameters, one of the inputs parameters value is not relevant. E.g.:

```
def custom_function(a, b):
    if a > 5:
        return a
    else:
        return b
```

For such cases the user may allow the calculation of such functions by setting the *undefined_input_handling* parameter of the Var constructor to 'function'. To guard against unexpected errors during function evaluation one can check whether a value is undefined using the *is undefined* syntax, like:

```
from autocalc.autocalc import undefined

def custom_function(a, b):
    if a is undefined:
        return undefined
    if a > 5:
        return a
    else:
        if b is undefined:
            return undefined
        else:
            return b
```

CHAPTER

THREE

WHAT'S NEW

3.1 v0.1.3

3.1.1 Documentation

- Created a proper README.rst file introducing the purpose, main features and a code example.
- Added the whole "Getting Started" and "User Guide" sections to the sphinx doc

3.2 v0.1.0

3.2.1 New Features

- There is a new `undefined_inputs` parameter to the `get` and `recalc` methods. If the function can not be evaluated because some of the inputs are undefined, these inputs are collected into this variable.
- The `PreviewAcc` class now reports which inputs are missing to calculate the results.
- The `titles` parameter of `PreviewAcc` now allows to specify an optional 'open'-key, which will be displayed, when the Accordion is in an open state.

3.2.2 Bug fixes

- More robust handling of undefined values with widgets

3.2.3 Internal Changes

- Unittests

CHAPTER

FOUR

INDICES AND TABLES

- genindex
- modindex
- search