Jupytr++
Functional Specification
Elena Cokova, Isaac Rothschild, Kalina Allen
Last Updated: 11/27/2017

Contents

Overview

Jupyter is a tool that allows data scientists to combine results/code with analysis in one document. It is a tool to write papers that use examples to demonstrate ideas.

Our overall goal is to add restricted revision control to the Jupyter workflow. We will not need many standard revision control features since the data science writing workflow is very different from the software development workflow. We will implement a subset of 'normal' revision control features (as shown in user stories below). Additionally, we will add a new feature to allow people to view a timeline of their changes and the meaningful solutions they arrived at.

User Scenario

The section describes the kind of person who would use our product.

Geoff is an expert analyst. He works for a data science company that calculates optimal placement of beanie baby toys in an office. During a typical day, he splits his time fairly evenly between working on his Jupyter based project, listening to his colleagues Jupyter based presentations, giving his own Jupyter based presentations, and then playing with beanie babies.

While Geoff is working on his Jupyter project about beanie babies he sometimes reaches a dead end. Instead of forgetting about this solution, Geoff uses the Jupyter Source Control system to tag the notebook state so he can return to it. After an intensive session meditating with the beanie babies Geoff takes the project in a new direction and discovers something cool. Geoff saves this state but then finds all sorts of typos that originated in the first solution he created. After a break to listen to his friends rocking-horse based presentation, Geoff fixes the typos in the old state and pushes the changes forwards using the timeline feature so that both tagged states are typo free.

Non Goals

This version of our product will not support the following features:

- multiple users working on the same notebook
- surgical operations other than removing a single snapshot and squashing a number of snapshots
- searching revisions by description
- long/complex snapshot descriptions

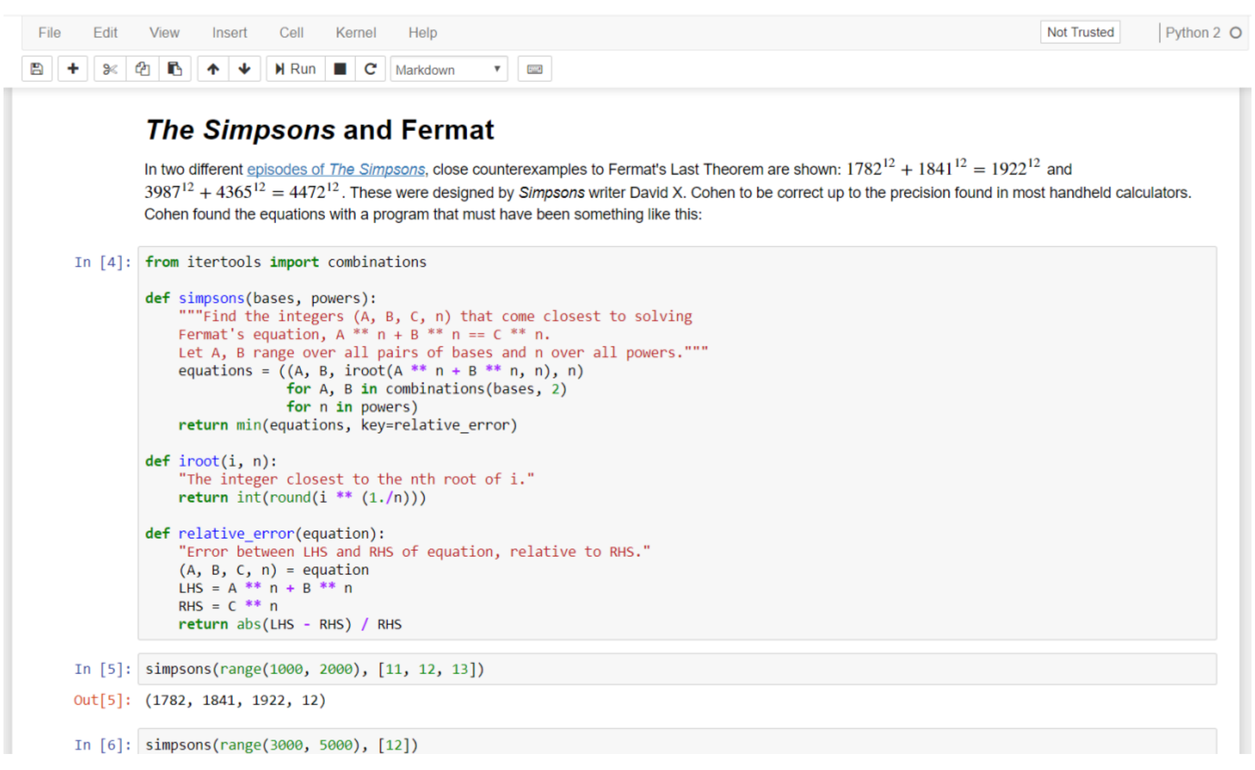
What is a Jupyter Notebook?

Jupyter is an open-source web application that allows users to create and share documents that contain live code, equations, visualizations and narrative text. Some of its main uses include: data cleaning and transformation, numerical simulation, statistical modeling, and data visualization.

Jupyter projects are structured in the form a notebook made of a collection of cells. Each cell contains either documentation, runnable code, or the output of a code cell. This format offers the flexibility to rerun code on custom data as needed to support the analysis in the documentation sections.

Current Notebook Interface

Current view:



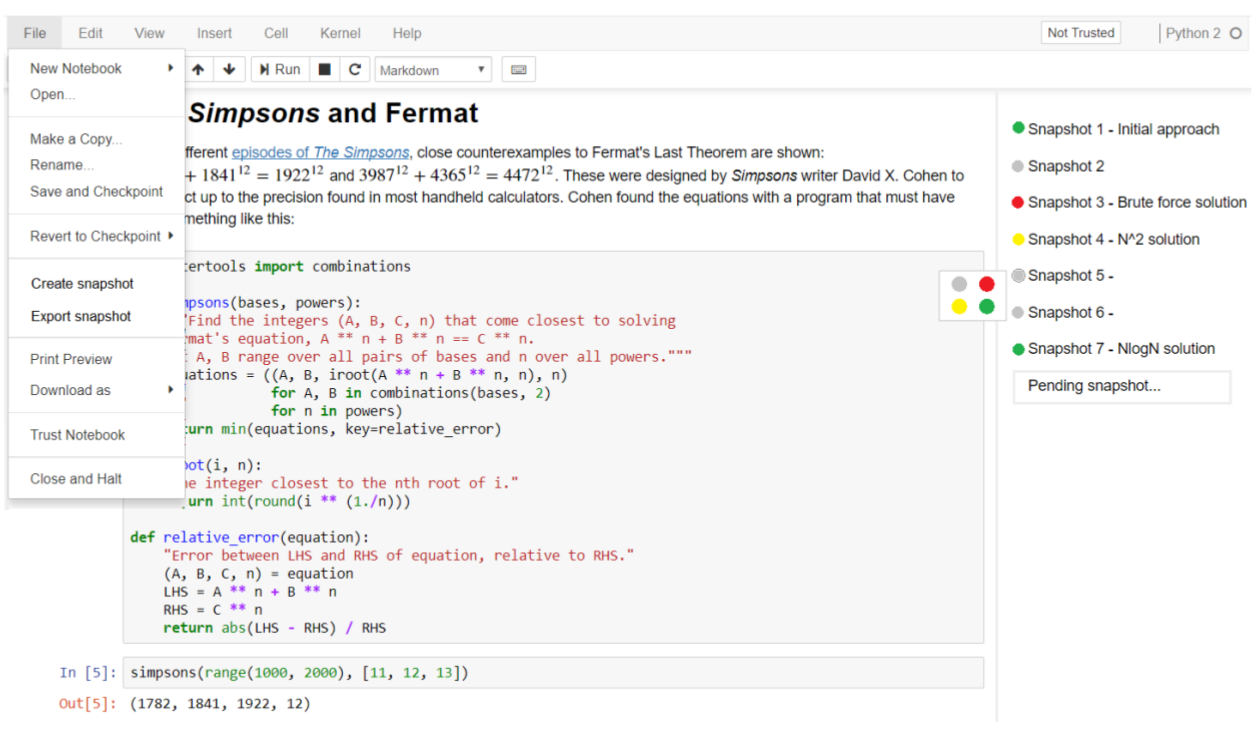
***The Simpsons* and Fermat**

In two different episodes of *The Simpsons*, close counterexamples to Fermat's Last Theorem are shown: $1782^{12} + 1841^{12} = 1922^{12}$ and $3987^{12} + 4365^{12} = 4472^{12}$. These were designed by *Simpsons* writer David X. Cohen to be correct up to the precision found in most handheld calculators. Cohen found the equations with a program that must have been something like this:

```
In [4]: from itertools import combinations

        def simpsons(bases, powers):
            """Find the integers (A, B, C, n) that come closest to solving
            Fermat's equation, A ** n + B ** n == C ** n.
            Let A, B range over all pairs of bases and n over all powers."""
            equations = ((A, B, iroot(A ** n + B ** n, n), n)
                         for A, B in combinations(bases, 2)
                         for n in powers)
            return min(equations, key=relative_error)

        def iroot(i, n):
            "The integer closest to the nth root of i."
            return int(round(i ** (1./n)))

        def relative_error(equation):
            "Error between LHS and RHS of equation, relative to RHS."
            (A, B, C, n) = equation
            LHS = A ** n + B ** n
            RHS = C ** n
            return abs(LHS - RHS) / RHS

In [5]: simpsons(range(1000, 2000), [11, 12, 13])
Out[5]: (1782, 1841, 1922, 12)

In [6]: simpsons(range(3000, 5000), [12])
```

The current interface allows users to create, modify, and rearrange cells within a notebook. Notebook state is persisted in an underlying SQLite database and represented in the frontend in a basic JSON/XML format. This format can be backed up in git as-is, but there does not currently exist a good interface to interpret it in a human readable manner.

Envisioned Notebook Interface



***Simpsons* and Fermat**

fferent episodes of *The Simpsons*, close counterexamples to Fermat's Last Theorem are shown:
$+ 1841^{12} = 1922^{12}$ and $3987^{12} + 4365^{12} = 4472^{12}$. These were designed by *Simpsons* writer David X. Cohen to
ct up to the precision found in most handheld calculators. Cohen found the equations with a program that must have
mething like this:

```
tertools import combinations

npsons(bases, powers):
'Find the integers (A, B, C, n) that come closest to solving
rmat's equation, A ** n + B ** n == C ** n.
 A, B range over all pairs of bases and n over all powers."""
uations = ((A, B, iroot(A ** n + B ** n, n), n)
           for A, B in combinations(bases, 2)
           for n in powers)
turn min(equations, key=relative_error)

oot(i, n):
ne integer closest to the nth root of i."
urn int(round(i ** (1./n)))

def relative_error(equation):
    "Error between LHS and RHS of equation, relative to RHS."
    (A, B, C, n) = equation
    LHS = A ** n + B ** n
    RHS = C ** n
    return abs(LHS - RHS) / RHS

In [5]: simpsons(range(1000, 2000), [11, 12, 13])
Out[5]: (1782, 1841, 1922, 12)
```

The window above gives the user a set of powers over the state of the notebook. The snapshot list on the right shows the sequence of tagged states the notebook for this notebook, as well as a colored identifier for each state. Green, Yellow, Red, and Grey have no intrinsic meaning but should be useful for the developer to classify each snapshot. This list lets users select a revision and return to it. This is similar to (`$ git checkout foo)` in normal source control workflows. On the left hand side, we see the `File>...` drop-down menu which shows two additional options, `Create snapshot` and `Export snapshot`. The first, `Create snapshot`, saves the current state of the notebook and adds a new entry to the snapshot list on the right. This is equivalent to (`$ git commit …`) in normal source control workflows. Next, `Export snapshot` gives the users power to clone and download the current notebook state as a ZIP. There isn't a strict equivalence in the traditional source control workflow, but this is like fetching a repo and compressing it.

Finally, there are some invisible features here. The Jupyter notebook will back up its state every 5 minutes, and whenever the user presses 'Save' in the editor. These states are less important than the tagged states but still should be trackable and we should be able to return them.

Not shown in the interface are optional features like deleting snapshots, rearranging snapshots, and splicing notebook histories.

User Workflow
This section describes the different ways that a user can interact with our product.

1. Snapshotting notebook state
- User gets to a meaningful state (successful or unsuccessful).
- User exports the state and gives it a quick summary.
  - Option A: Simple "thumbs up"/"thumbs down" marker for quick view. State unnamed.
  - Option B: Short label, with the option to add metadata like a longer summary or a green/yellow/red status marker. This is demonstrated above.

2. Viewing timeline of changes
- User wants to view previously saved states.
- User pulls up a timeline and can vertically click between snapshots in time.
- Each snapshot can be associated with a color rating representing how good the state is, or may be left unrated.
- User can click on a snapshot to replace their current state with that snapshot's state.
- While in a previously snapshotted state, the user can view all their files, run code, etc.
- The user can click "Pending snapshot..." and get back to the current working version.

3. Downloading snapshotted state
- While in a previously snapshotted state, the user decides that they would like to share this version of their project.
- The user can download this state as a separate zip file by selecting "Export snapshot" from the File menu.

4. Squashing old changes
- User looks through their timeline and sees that there are a bunch of bad states.
- They decide that these snapshots are no longer necessary and would like to remove them from the timeline.
- User selects the unnecessary states and "squashes" them.
- Now the user can only see states before and after the snapshots they just deleted.

5. Continuing with from previous state with linear history
- User has snapshots A, B, and C. State C is the end of some line of inquiry. User checks out state B and continues down some other path and snapshots a new state D. This state logically follows C, and so timeline of snapshots is now: A,B,C,D
- That is, for the new state D:
  - D = C - diff(C,B) + diff(D,B)

6. Patching individual changes from other states
- User has snapshots A, B, C and D. If user's local copy is B, then user can patch files from state A, C, and D onto current copy and continue working. Next snapshot will be E and follow: A, B, C, D, and E.
- Changes patched onto B will not be propagated onto C or D.

7. Propagating changes forward
- User has states A, B, C, and D. User makes modifications to D that she wants to include in all states (ie, updates license info, documentation). User patches desired (ie, a strict subset of) changes onto A and changes are propagated through states B-D. No new snapshots are created.