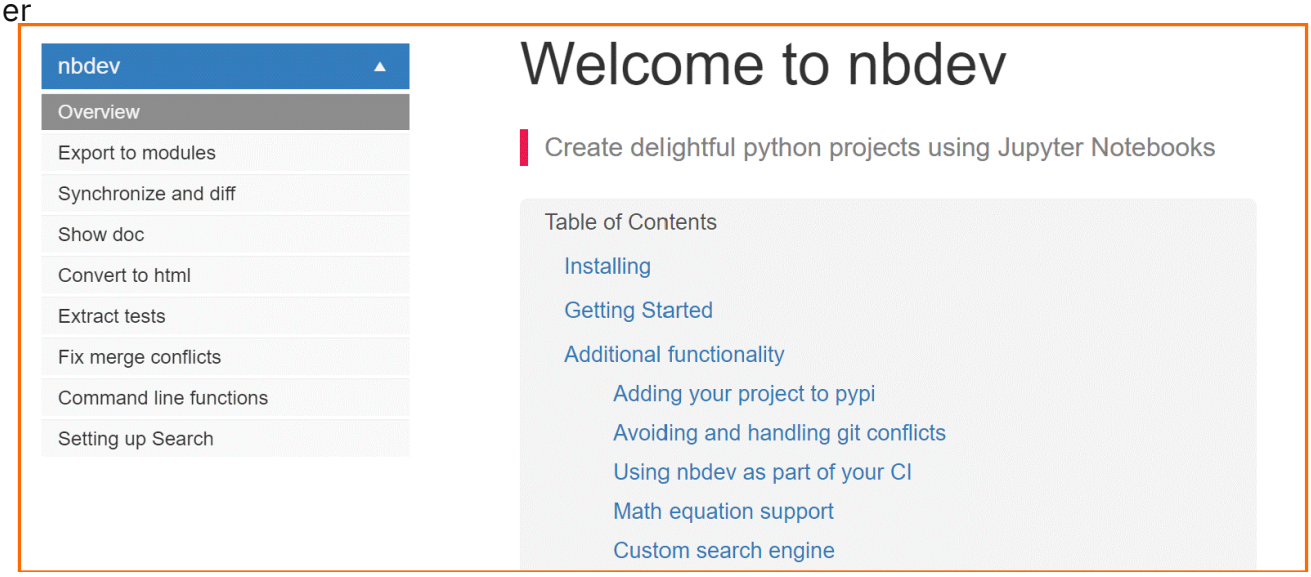
#Jupyter on Steroids: Create Packages, Tests, and Rich Documents

emy
ard
ding
archer
ai

December 5th 2019

TWEET THIS



> "I really do think [nbdev] is a huge step forward for programming environments": Chris Lattner, inventor of Swift, LLVM, and Swift Playgrounds.

My fast.ai colleague Sylvain Gugger and I have been working on a labor of love for the last couple of years. It is a Python programming environment called nbdev, which allows you to create complete python packages, including tests and a rich documentation system, all in Jupyter Notebooks.

We've already written a large programming library (fastai v2) using nbdev, as well as a range of smaller projects.

Nbdev is a system for something that we call *exploratory programming*. Exploratory programming is based on the observation that most of us spend most of our time as coders exploring and experimenting.

We experiment with a new API that we haven't used before, to understand exactly how it behaves. We explore the behavior of an algorithm that we are developing, to see how it works with various kinds of data. We try to debug our code, by exploring different combinations of inputs. And so forth...

nbdev: exploratory programming

We believe that the very process of exploration is valuable in itself, and that this process should be saved so that other programmers (including yourself in six months time) can see what happened and learn by example. Think of it as something like a scientist's journal.

You can use it to show the things that you tried, what worked and what didn't, and what you did to develop your understanding of the system within which you are working. During this exploration, you will realize that some parts of this understanding are critical for the system to work.

Therefore, the exploration should include the addition of tests and assertions to ensure this behavior.

This kind of "exploring" is easiest when you develop on the prompt (or REPL), or using a notebook-oriented development system like Jupyter Notebooks. But these systems are not as strong for the "programming"

part. This is why people use such systems mainly for early exploring, and then switch to an IDE or text editor later in a project.

They switch to get features like good doc lookup, good syntax highlighting, integration with unit tests, and (critically!) the ability to produce final, distributable source code files, as opposed to notebooks or REPL histories.

The point of nbdev is to bring the key benefits of IDE/editor development into the notebook system, so you can work in notebooks without compromise for the entire life-cycle. To support this kind of exploration, nbdev is built on top of Jupyter Notebook (which also means we get much better support for Python's dynamic features than in a normal editor or IDE), and adds the following critically important tools for software development:

- Python modules are automatically created for you, following best practices such as automatically defining `__all__` (more details) with your exported functions, classes, and variables

- Navigate and edit your code in a standard text editor or IDE, and export any changes automatically back into your notebooks

- Automatically create searchable, hyperlinked documentation from your code; any word you surround in backticks will by hyperlinked to the appropriate documentation, a sidebar will be created for you in your documentation site with links to each of your modules, and more

- Pip installers (uploaded to pypi for you)

- Testing (defined directly in your notebooks, and run in parallel)

- Continuous integration

- Version control conflict handling

Here's a snippet from our actual "source code" for nbdev, which is itself written in nbdev (of course!)

What's a notebook?

A jupyter notebook is a json file behind the scenes. We can just read it with the json module, which will return a nested dictionary of dictionaries/lists of dictionaries, but there are some small differences between reading the json and using the tools from `nbformat` so we'll use this one.

In [ ]:
```
#export
def read_nb(fname):
    "Read the notebook in `fname`."
    with open(Path(fname),'r', encoding='utf8') as f:
        return nbformat.reads(f.read(), as_version=4)
```

`fname` can be a string or a pathlib object.

In [ ]:
```
test_nb = read_nb('00_export.ipynb')
```

The root has four keys: `cells` contains the cells of the notebook, `metadata` some stuff around the version of python used to execute the notebook, `nbformat` and `nbformat_minor` the version of nbformat.

In [ ]:
```
test_nb.keys()
```

Out[ ]:
```
dict_keys(['cells', 'metadata', 'nbformat', 'nbformat_minor'])
```

As you can see, when you build software this way, everyone in your project team gets to benefit from the work you do in building an understanding of the problem domain, such as file formats, performance characteristics, API edge cases, and so forth.

> Since development occurs in a notebook, you can also add charts, text, links, images, videos, etc, that will be included automatically in the documentation of your library.

The cells where your code is defined will be hidden and replaced by standardized documentation of your function, showing its name, arguments, docstring, and link to the source code on GitHub.

> For more information about features, installation, and how to use nbdev, see its documentation (which is, naturally, automatically generated from its source code). I'll be posting a step by step tutorial in the coming days.

In the rest of this post, I'll describe more of the history and context behind the *why*: *why* did we build it, and *why* did we design it the way we did. First, let's talk about a little history... (And if you're not interested in the history, you can skip ahead to What's missing in Jupyter Notebook.)

Software Development Tools

Most software development tools are not built from the foundations of thinking about exploratory programming. When I began coding, around 30 years ago, waterfall software development was used nearly exclusively. It seemed to me at the time that this approach, where an entire software system would be defined in minute detail upfront, and then coded as closely to the specification as possible, did not fit at all well with how I actually got work done.

In the 1990s, however, things started to change. Agile development became popular. People started to understand the reality that most software development is an iterative process, and developed ways of working which respected this fact.

> However, we did not see major changes to the software development tools that we used, that matched the major changes to our ways of working.

There were some pieces of tooling which got added to our arsenal, particularly around being able to do test driven development more easily. However this tooling tended to appear as minor extensions to existing editors and development environments, rather than truly rethinking what a development environment could look like.

In recent years we've also begun to see increasing interest in exploratory testing as an important part of the agile toolbox. We absolutely agree! But we also think this doesn't go nearly far enough; we think in nearly *every* part of the software development process that exploration should be a central part of the story.

The legendary Donald Knuth was way ahead of his time. He wanted to see things done very differently. In 1983 he developed a methodology called literate programming.

> He describes it as "*a methodology that combines a programming language with a documentation language, thereby making programs more robust, more portable, more easily maintained, and arguably more fun to write than programs that are written only in a high-level language. The main idea is to treat a program as a piece of literature, addressed to human beings rather than to a computer.*"

For a long time I was fascinated by this idea, but unfortunately it never really went anywhere. The tooling available for working this way

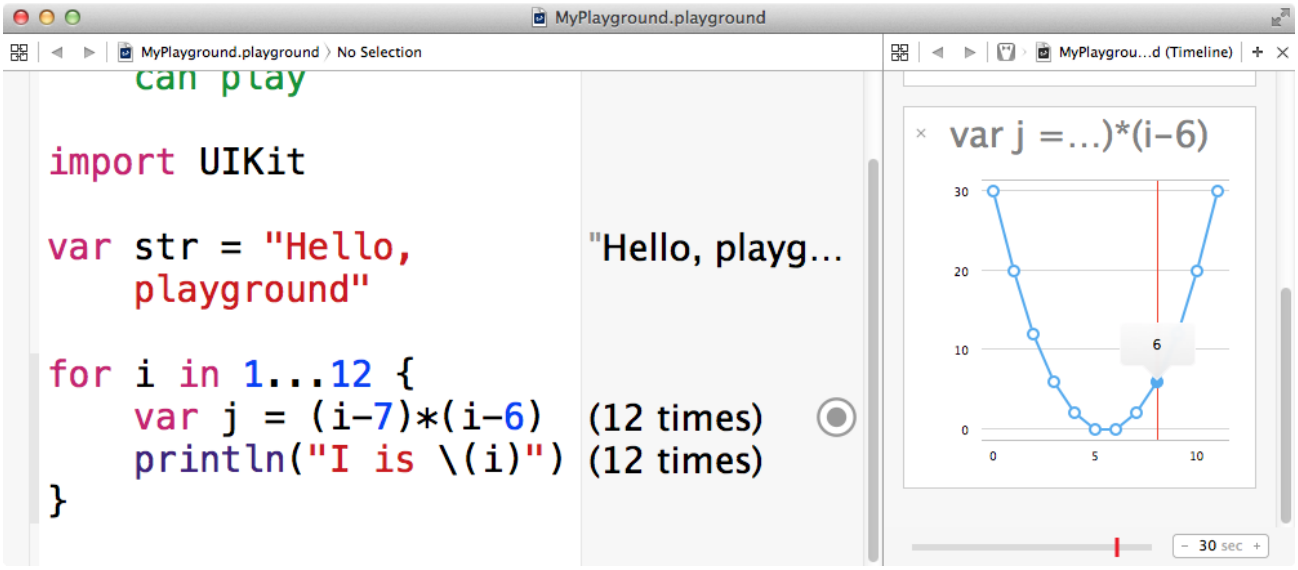
resulted in software development taking much longer, and very few people decided that this compromise was worth it.

Nearly 30 years later another brilliant and revolutionary thinker, Bret Victor, expressed his deep discontent for the current generation of development tools, and described how to design "a programming system for understanding programs".

> As he said in his groundbreaking speech "Inventing on Principle": "*Our current conception of what a computer program is 'a list of textual definitions that you hand to a compiler' that's derived straight from Fortran and ALGOL in the late '50's. Those languages were designed for punch-cards.*"

He laid out, and illustrated with fully worked examples, a range of new principles for designing programming systems. Whilst nobody has as yet fully implemented all of his ideas, there have been some significant attempts to implement some parts of them.

Perhaps the most well-known and complete implementation, including inline display of intermediate results, is Chris Lattner's Swift and Xcode Playgrounds.



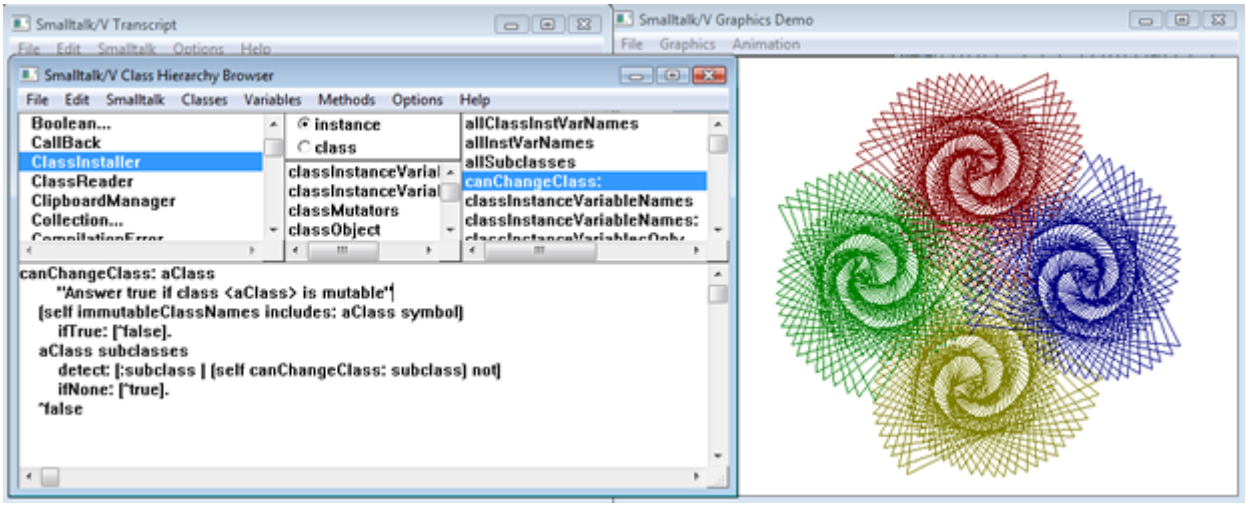
Whilst this is a big step forward, it is still very constrained by the basic limitations of sitting within a development environment which was not originally built with such explorations in mind.

For instance, the exploration process is not captured by this at all, tests cannot be directly integrated into it, and the full rich vision of literate programming cannot be implemented.

Interactive programming environments

There has been another very different direction in software development, which is interactive programming (and the related live programming). It started many decades ago with the LISP and Forth REPLs, which allowed developers to interactively add and remove code in a running application.

Smalltalk (1980) took things even further, providing a fully interactive visual workspace. In all these cases, the languages themselves were well suited to this kind of interactive work, for instance with LISP's macro system and "code as data" foundations.



Although this approach isn't how most regular software development is done today, it is the most popular approach in many areas of scientific, statistical, and other data-driven programming. (JavaScript front-end programming is however increasingly borrowing ideas from those approaches, such as hot reloading and in-browser live editing.) Matlab, for instance, started out as an entirely interactive tool back in the 1970's, and today is still widely used in engineering, biology, and various other areas (it also provides regular software development features nowadays).

> Similar approaches were used by SPLUS, and it's open source cousin R, which is today extremely popular in the statistic and data visualization communities (amongst others).

I got particularly excited when I first used Mathematica about 25 years ago. Mathematica looked to me like the closest thing I'd seen to something that could support literate programming, without compromising on productivity.

To do this, it used a "notebook" interface, which behaved a lot like a traditional REPL, but also allowed other types of information to be

included, including charts, images, formatted text, outlining sections, and so forth.

> In fact, not only did it not compromise on productivity, but I found it actually allowed me to build things that previously were beyond me, because I could try algorithms out and immediately get feedback in a very visual way.

In the end though, Mathematica didn't really help me build anything useful, because I couldn't distribute my code or applications to colleagues (unless they spent thousands of dollars for a Mathematica license to use it), and I couldn't easily create web applications for people to access from the browser.

In addition, I found my Mathematica code would often end up much slower and more memory hungry than code I wrote in other languages.

> So you can imagine my excitement when Jupyter Notebook appeared on the scene. This used the same basic notebook interface as Mathematica (although, at first, with a small subset of the functionality) but was open source, and allowed me to write in languages that were widely supported and freely available.

I've been using Jupyter not just for exploring algorithms, APIs, and new research ideas, but also as a teaching tool at fast.ai. Many students have found that the ability to experiment with inputs and view intermediate results and outputs, as well as try out their own modifications, helped them to more fully and deeply understand the topics being discussed.

We are also writing a book entirely using Jupyter Notebooks, which has been an absolute pleasure, allowing us to combine prose, code examples, hierarchical structured headings, and so forth, whilst ensuring that our sample outputs (including charts, tables, and images) always correctly match up to the code examples.

In short: we have really enjoyed using Jupyter Notebook, we find that we do great work using it, and our students love it. But it just seemed like such a shame that we weren't actually using it to build our software!

So what's Missing in Jupyter Notebook?

Whilst Jupyter Notebook is great at the "explorable" part part of "explorable programming", it's not so great at the "programming" part. For instance, it doesn't really provide a way to do things like:

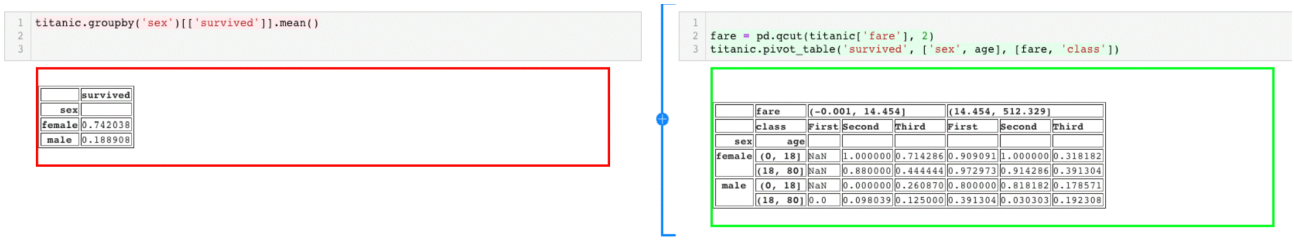
- Create modular reusable code, which can be run outside of Jupyter
- Creating hyperlinked searchable documentation
- Test code (including automatically through continuous integration)
- Navigate code
- Handle version control

Because of this, people generally have to switch between a mix of poorly integrated tools, with significant friction as they move from tool to tool, to get the advantages of each.

We decided that the best way to handle these things was to leverage great tools that already exist, where possible, and build our own where needed. For instance, for handling pull requests and viewing diffs, there's already a great tool: ReviewNB.

> When you look at graphical diffs in ReviewNB, you suddenly realize how much has been missing all this time in plain text diffs. For instance, what if a commit made your image generation blurry?

Or made your charts appear without labels? You really know what's going on when you have that visual diff, such as in this example of a diff containing an output cell with a Pandas DataFrame:



```
titanic.groupby('sex')[['survived']].mean()
```

	survived
sex	
female	0.742038
male	0.188908

```
fare = pd.qcut(titanic['fare'], 2)
titanic.pivot_table('survived', ['sex', age], [fare, 'class'])
```

	fare	(-0.001, 14.454]			(14.454, 512.329]		
	class	First	Second	Third	First	Second	Third
sex	age						
female	(0, 18]	NaN	1.000000	0.714286	0.909091	1.000000	0.318182
	(18, 80]	NaN	0.880000	0.444444	0.972973	0.914286	0.391304
male	(0, 18]	NaN	0.000000	0.260870	0.800000	0.818182	0.178571
	(18, 80]	0.0	0.098039	0.125000	0.391304	0.030303	0.192308

Many merge conflicts are avoided with nbdev, because it installs git hooks for you which strip out much of the metadata that causes those conflicts in the first place. If you get a merge conflict when you pull from git, just run nbdev_fix_merge.

With this command, nbdev will simply use *your* cell outputs where there are conflicts in outputs, and if there are conflicts in cell inputs, then *both* cells are included in the final notebook, along with conflict markers so you can easily find them and fix them directly in Jupyter.

```
<<<<<<< HEAD
```

```
In [5]:
path = Path.cwd()
path.ls()[0]
```

```
Out[5]: PosixPath('/home/sgugger/notebooks/01_core.ipynb')
```

```
=======
```

```
In [3]:
path = Path.cwd()
path.ls()[-1]
```

```
Out[3]: PosixPath('/home/jphoward/notebooks/01_core.ipynb')
```

```
>>>>>>> a7ec1b0bfb8e23b05fd0a2e6cafcb41cd0fb1c35
```

Modular reusable code is created by nbdev by simply creating standard Python modules. nbdev looks for special comments in code cells, such as #export, which indicates that a cell should be exported to a python module.

> Each notebook is associated with a particular python module by using a special comment at the start of the notebook. A documentation site (using Jekyll, so supported directly by GitHub Pages) is automatically built from the notebooks and special comments.

We wrote our own documentation system, since existing approaches such as Sphinx didn't provided all the features that we needed.

For code navigation, there are already wonderful features built into most editors and IDEs, such as vim, Emacs, and vscode. And as a bonus, GitHub even supports code navigation directly in its web interface now (in beta, in selected projects, such as fastai)!

> So we've ensured that the code that nbdev exports can be navigated and edited directly in any of these systems - and that any edits can be **automatically synchronized** back with the notebooks.

For testing, we've written our own simple library and command line tool. Tests are written directly in notebooks, as part of the exploration and development (and documentation) process, and the command line tool runs tests in all notebooks in parallel.

The natural statefulness of notebooks turns out to be a really great way to develop both unit tests and integration tests. Rather than having special syntax to learn to create test suites, you just use the regular collection and looping constructs in python.

So there's much fewer new concepts to learn. These tests can also be run in your normal continuous integration tools, and they provide clear information about the source of any test errors that come up. The default nbdev template includes integration with GitHub Actions for continuous integration and other features (PRs for other platforms are welcome).

Dynamic Python

One of the challenges in fully supporting Python in a regular editor or IDE is that Python has particularly strong *dynamic* features. For instance, you can add methods to a class at any time, you can change the way that classes are created and how they work by using the metaclass system, and you can change how functions and methods behave by using decorators.

> Microsoft developed the Language Server Protocol, which can be used by development environments to get information about the current file and project necessary for auto-completions, code navigation, and so forth.

However, with a truly dynamic language like python, such information will always just be guesses, since actually providing the correct information would require running the python code itself (which it can't really do, for all kinds of reasons - for instance the code may be in a state while you're writing it that actually deletes all your files!)

On the other hand, a notebook contains an actual running Python interpreter instance that you're fully in control of. So Jupyter can provide auto-completions, parameter lists, and context-sensitive documentation based on the actual state of your code.

> For instance, when using Pandas we get tab completion of all the column names of our DataFrames. We've found that this feature of Jupyter Notebook makes exploratory programming significantly more productive.

We haven't needed to change anything to make it work well in nbdev; it's just part of the great features of Jupyter that we get for free by building on that platform.

What now

In conjunction with developing nbdev, we've been writing fastai v2 from scratch entirely in nbdev. fastai v2 provides a rich, well-structured API for building deep learning models. It will be released in the first half of 2020. It's already feature complete, and early adopters are already

building cool projects with the pre-release version. We've also written other projects in fastai v2, some of which will be released in the coming weeks.

We've found that we're 2x-3x more productive using nbdev than using traditional programming tools. For me, this is a big surprise, since I have coded nearly every day for over 30 years, and in that time have tried dozens of tools, libraries, and systems for building programs.

> I didn't expect there could still be room for such a big job in productivity. It's made me feel excited about the future, because I suspect there could still be a lot of room to develop other ideas for developer productivity, and because I'm looking forward to seeing what people build with nbdev.

If you decide to give it a go, please let us know how you get along! And of course feel free to ask any questions. The best place for these discussions is this forum thread that we've created for nbdev. PRs are of course welcome in the nbdev GitHub repo.

Thank you for taking an interest in our project!

Acknowledgements: Thanks to Alexis Gallagher and Viacheslav Kovalevskyi for their helpful feedback on drafts of this article. Thanks to Andrew Shaw for helping to build prototypes of show_doc, and to Stas Bekman for much of the git hooks functionality. Thanks to Hamel Husain for helpful ideas for using GitHub Actions.

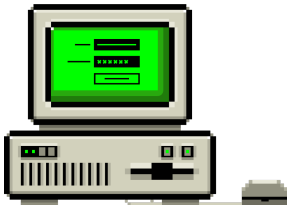
Tags

 # Ai **# Deeplearning** **# Fast.ai** **# Notebook**

 # Python Top Story **# Jupyter Notebook Alternative**

Exploratory Programming # Hackernoon Top Story

Continue the Discussion



Hackernoon Newsletter curates great stories by real tech professionals

Get solid gold sent to your inbox. Every week!

Email Address *

First Name

Last Name

TOPICS OF INTEREST

- [x] Software Development
- [x] Blockchain Crypto
- [x] General Tech
- [x] Best of Hacker Noon

Get great stories by email