6 surprising ways to use Jupyter

Jupyter is much more than a data-analysis tool. Learn about some of the most creative ways you can use the Python-based software.

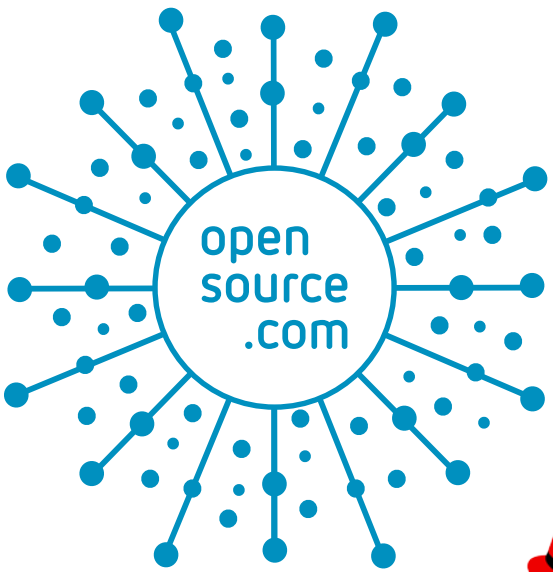
ABOUT OPENSOURCE.COM

What is Opensource.com?

OPENSOURCE.COM publishes stories about creating, adopting, and sharing open source solutions. Visit Opensource.com to learn more about how the open source way is improving technologies, education, business, government, health, law, entertainment, humanitarian efforts, and more.

Submit a story idea: opensource.com/story

Email us: open@opensource.com



Supported by
Red Hat

SETH KENLON

MOSHE ZADKA

Moshe has been involved in the Linux community since 1998, helping in Linux "installation parties". He has been programming Python since 1999, and has contributed to the core Python interpreter. Moshe has been a DevOps/SRE since before those terms existed, caring deeply about software reliability, build reproducibility and other such things. He has worked in companies as small as three people and as big as tens of thousands—usually some place around where software meets system administration.

Follow me at @moshezadka

CHAPTERS

Introduction

THE JUPYTER PROJECT offers interactive ways to write software with technology like JupyterLab and Jupyter Notebook. This software is commonly used for data analysis, but what you might not know (and the Jupyter community didn't expect) is how many things you can do with it. Here are unexpected and creative ways to use Jupyter.

Teach kids Python by building an interactive game

Open source tools can help anyone get started learning Python in an easy and fun way—making games.

PYTHON HAS EARNED a reputation as a wonderful beginner programming language. But where does one begin?

One of my favorite ways to get people interested in programming is by writing games.

PursuedPyBear (ppb) [1] is a game programming library optimized for teaching, and I recently used it to teach my children more about my favorite programming language [2].

The Jupyter [3] project is a browser-based Python console, initially designed for data scientists to play with data.

I have a Jupyter Notebook designed to teach you how to make a simple interactive game, which you can download from here [4]. In order to open the file, you will need to install the latest Jupyter project, JupyterLab.

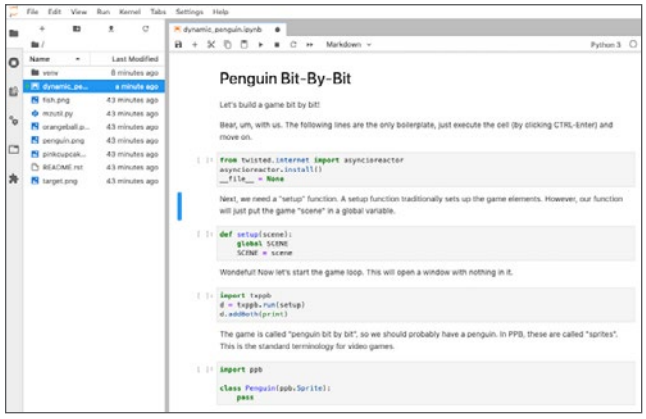
Prerequisites:

- Running a recent version of Python (instructions for Linux [5], Mac [6], and Windows [7])
- Running a recent version of Git (instructions here [8])

We will briefly configure a virtual environment to create a separate space for the needed libraries. (You can learn more about how virtual environments work here [9].)

```
$ git clone https://github.com/moshez/penguin-bit-by-bit.git
$ cd penguin-bit-by-bit
$ python -m venv venv
$ source ./venv/bin/activate
$ pip install -r requirements.txt
$ jupyter lab .
```

The last command should open JupyterLab in your default browser at the address http://localhost:8888/lab. Choose the **dynamic_penguin.ipynb** file in the left-hand column, and we can get started!



The event loop that will run the game

Jupyter runs an event loop internally, which is a process that manages the running of further asynchronous operations. The event loop used in Jupyter is asyncio [10], and PursuedPyBear runs its own event loop.

We can integrate the two using another library, Twisted [11], like glue. This sounds complicated, but thankfully, the complexity is hidden behind libraries, which will do all the hard work for us.

The following cell in Jupyter takes care of the first half—integrating Twisted with the asyncio event loop.

The `__file__ = None` is needed to integrate PursuedPyBear with Jupyter.

```
from twisted.internet import asyncioreactor
asyncioreactor.install()
__file__ = None
```

Next, we need a “setup” function. A setup function is a common term for the configuration of key game elements. However, our function will only put the game “scene” in a global

variable. Think of it like us defining the table on which we will play our game.

The following cell in Jupyter Notebook will do the trick.

```
def setup(scene):
    global SCENE
    SCENE = scene
```

Now we need to integrate PursuedPyBear's event loop with Twisted. We use the `txppb` module for that:

```
import txppb
d = txppb.run(setup)
d.addBoth(print)
```

The `print` at the end helps us if the game crashes because of a bug—it will print out a traceback to the Jupyter output.

This will show an empty window, ready for the game elements.

This is where we start taking advantage of Jupyter—traditionally, the whole game needs to be written before we start playing. We buck convention, however, and start playing the game immediately!

Making the game interesting with interaction

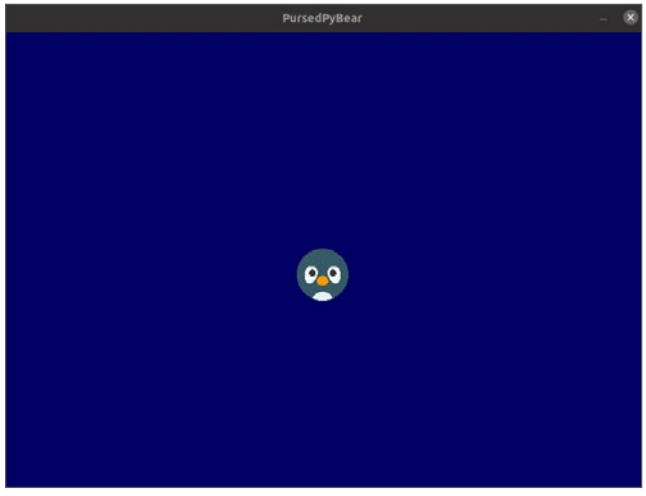
It is not a very interesting game, though. It has nothing and just sits there. If we want something, we better add it.

In video game programming, the things moving on the screen are called "sprites." In PursuedPyBear, sprites are represented by classes. A sprite will automatically use an image named the same as the class. I got a little penguin image from Kenney [12], a collection of free and open source video game assets.

```
import ppb

class Penguin(ppb.Sprite):
    pass
```

Now let's put the penguin riiiiight in the middle.

```
SCENE.add(Penguin(pos=(0,0)))
```



It carefully sits there in the middle. This is marginally more interesting than having nothing. That's good—this is exactly what we want. In incremental game development, every step should be only marginally more interesting.

Adding movement to our penguin game with ppb

But penguins are not meant to sit still! The penguin should move around. We will have the player control the penguin with the arrow keys. First, let's map the keys to vectors:

```
from ppb import keycodes

DIRECTIONS = {keycodes.Left: ppb.Vector(-1,0), keycodes.Right:
              ppb.Vector(1,0),
              keycodes.Up: ppb.Vector(0, 1), keycodes.Down:
              ppb.Vector(0, -1)}
```

Now we will use a utility library. The `set_in_class` function sets the method in the class. Python's ability to add functions to classes retroactively is really coming in handy!

```
from mzutil import set_in_class

Penguin.direction = ppb.Vector(0, 0)

@set_in_class(Penguin)
def on_update(self, update_event, signal):
    self.position += update_event.time_delta * self.direction
```

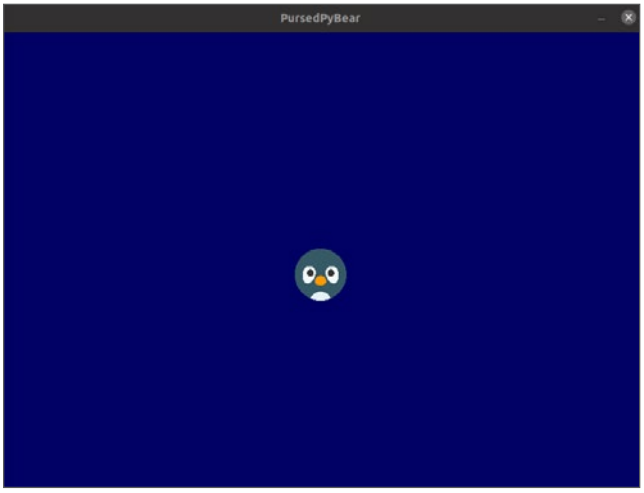
The code for `set_in_class` is not long, but it does use some non-trivial Python tricks. We will put the full utility library at the end of the article for review, and for the sake of flow, we will skip it for now.

Back to the penguin!

Oh, um, well.

The penguin is diligently moving…at zero speed, precisely nowhere. Let's manually set the direction to see what happens.

```
Penguin.direction = DIRECTIONS[keycodes.Up]/4
```



Click to view image animation.

The direction is up, but a little slow. This gives enough time to set the penguin's direction back to zero manually. Let's do that now!

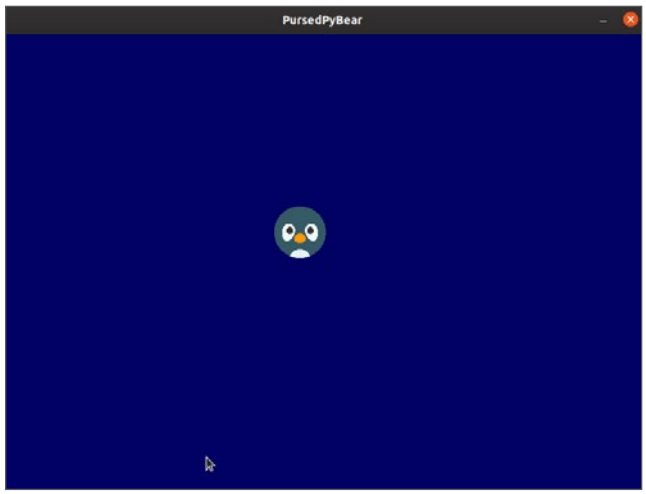
```
Penguin.direction = ppb.Vector(0, 0)
```

Adding interactivity to our penguin game

Phew, that was exciting—but not what we wanted. We want the penguin to respond to keypresses. Controlling it from the code is what gamers refer to as "cheating."

Let's set it to set the direction to the keypress, and back to zero when the key is released.

```
@set_in_class(Penguin)
def on_key_pressed(self, key_event, signal):
    self.direction = DIRECTIONS.get(key_event.key, ppb.
                     Vector(0, 0))

@set_in_class(Penguin)
def on_key_released(self, key_event, signal):
    if key_event.key in DIRECTIONS:
        self.direction = ppb.Vector(0, 0)
```



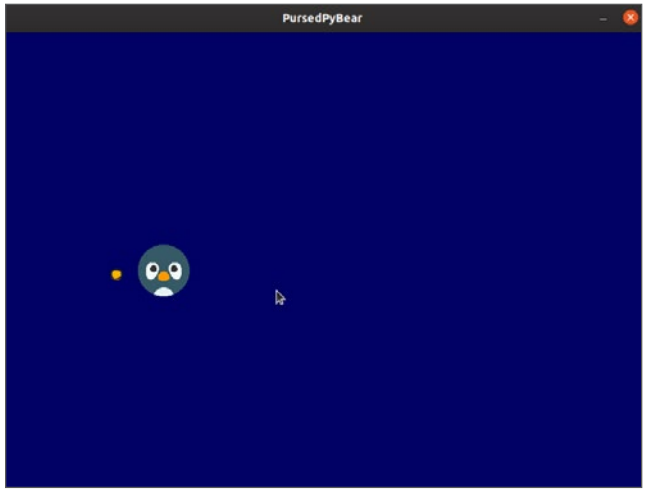
Click to view image animation.

The Penguin is a bit bored, isn't it? Maybe we should give it an orange ball to play with.

```
class OrangeBall(ppb.Sprite):
    pass
```

Again, I made sure to have an image called `orangeball.png`. Now let's put the ball on the left side of the screen.

```
SCENE.add(OrangeBall(pos=(-4, 0)))
```



Click to view image animation.

Try as it might, the penguin cannot kick the ball. Let's have the ball move away from the penguin when it approaches.

First, let's define what it means to "kick" the ball. Kicking the ball means deciding where it is going to be in one second, and then setting its state to "moving."

At first, we will just move it by having the first update move it to the target position.

```
OrangeBall.is_moving = False

@set_in_class(OrangeBall)
def kick(self, direction):
    self.target_position = self.position + direction
    self.original_position = self.position
    self.time_passed = 0
    self.is_moving = True

@set_in_class(OrangeBall)
def on_update(self, update_event, signal):
    if self.is_moving:
        self.position = self.target_position
        self.is_moving = False
```

Now, let's kick it!

```
ball, = SCENE.get(kind=OrangeBall)
ball.kick(ppb.Vector(1, 1))
```

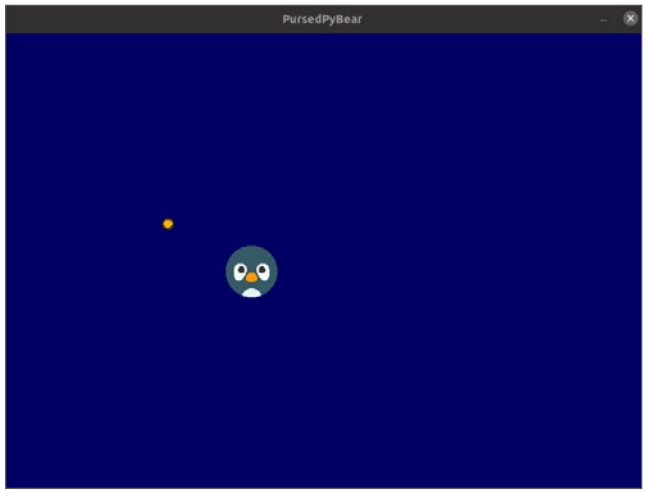
But this just teleports the ball; it immediately changes the position. In real life, the ball goes between the intermediate points. When it's moving, it will interpolate between where it is and where it needs to go.

Naively, we would use linear interpolation [13]. But a cool video game trick is to use an "easing" function. Here, we use the common "smooth step."

```
from mzutil import smooth_step

@set_in_class(OrangeBall)
def maybe_move(self, update_event, signal):
    if not self.is_moving:
        return False
    self.time_passed += update_event.time_delta
    if self.time_passed >= 1:
        self.position = self.target_position
        self.is_moving = False
        return False
    t = smooth_step(self.time_passed)
    self.position = (1-t) * self.original_position + t * self.
                    target_position
    return True

OrangeBall.on_update = OrangeBall.maybe_move
```

Now, let's try kicking it again.

```
ball, = SCENE.get(kind=OrangeBall)
ball.kick(ppb.Vector(1, -1))
```

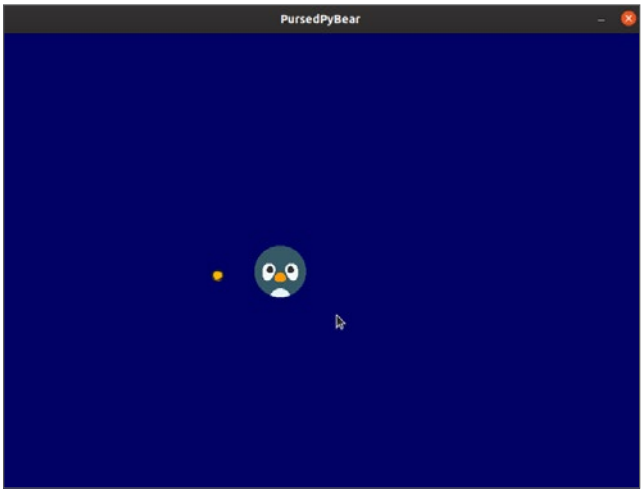


Click to view image animation.

But really, the penguin should be kicking the ball. When the ball sees that it is colliding with the penguin, it will kick itself in the opposite direction. If the penguin has gotten right on top of it, the ball will choose a random direction.

The update function now calls `maybe_move` and will only check collision if we are not moving right now.

```
from mzutil import collide
import random

OrangeBall.x_offset = OrangeBall.y_offset = 0.25

@set_in_class(OrangeBall)
def on_update(self, update_event,signal):
    if self.maybe_move(update_event, signal):
        return
    penguin, = update_event.scene.get(kind=Penguin)
    if not collide(penguin, self):
        return
    try:
        direction = (self.position -
                     penguin.position).normalize()
    except ZeroDivisionError:
        direction = ppb.Vector(random.uniform(-1, 1),
                     random.uniform(-1, 1)).normalize()
    self.kick(direction)
```



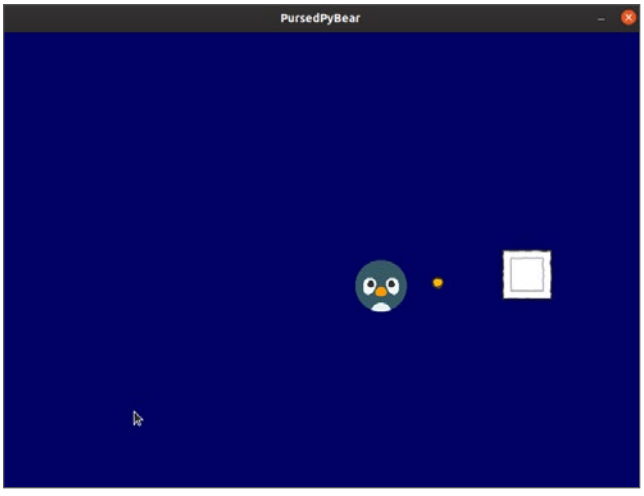
Click to view image animation.

But just kicking a ball around is not that much fun. Let's add a target.

```
class Target(ppb.Sprite):
    pass
```

Let's put the target at the right of the screen.

```
SCENE.add(Target(pos=(4, 0)))
```



Click to view image animation.

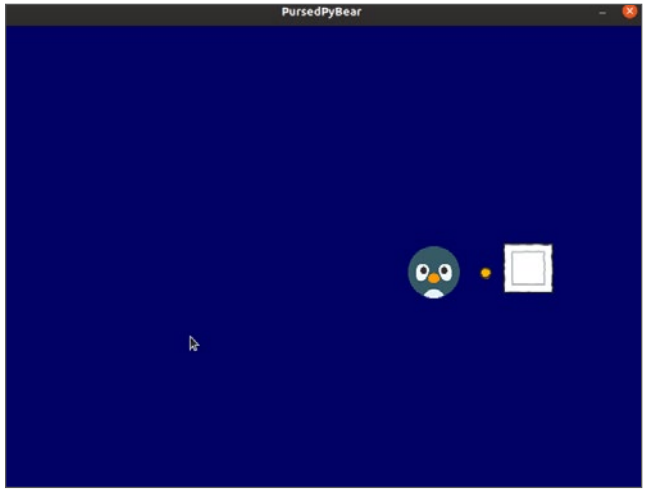
Rewarding our penguin

Now, we will want a reward for the penguin when it kicks the ball into the target. How about a fish?

```
class Fish(ppb.Sprite):
    pass
```

When the target gets the ball, it should remove it and create a new ball at the other end of the screen. Then, it will cause a fish to appear.

```
@set_in_class(Target)
def on_update(self, update_event, signal):
    for ball in update_event.scene.get(kind=OrangeBall):
        if not collide(ball, self):
            continue
        update_event.scene.remove(ball)
        update_event.scene.add(OrangeBall(pos=(-4, random.
                                uniform(-3, 3))))
        update_event.scene.add(Fish(pos=(random.uniform(-4, -3),
                                random.uniform(-3, 3))))
```



Click to view image animation.

We want to have the penguin eat the fish. When the fish sees the penguin, it should vanish.

```
Fish.x_offset = 0.05
Fish.y_offset = 0.2
@set_in_class(Fish)
def on_update(self, update_event,signal):
    penguin, = update_event.scene.get(kind=Penguin)
    if collide(penguin, self):
        update_event.scene.remove(self)
```

It works!

Iterative game design is fun for penguins and people alike!

This has all the makings of a game: the player-controlled penguin kicks the ball into the target, gets a fish, eats the fish, and kicks a new ball. This would work as a "grinding level" part of a game, or we could add obstacles to make the penguin's life harder.

Whether you are an experienced programmer, or just getting started, programming video games is fun. PursuedPyBear with Jupyter brings all the joy of classic 2D games with the interactive programming capabilities of the classic environments like Logo and Smalltalk. Time to enjoy a little retro 80s!

Appendix

Here is the full source code of our utility library. It provides some interesting concepts to make the game board work. For more on how it does that, read about collision detection [14], setattr [15]. and the __name__ attribute [16].

```
def set_in_class(klass):
    def retval(func):
        setattr(klass, func.__name__, func)
        return func
    return retval

def smooth_step(t):
    return t * t * (3 - 2 * t)

_WHICH_OFFSET = dict(
    top='y_offset',
    bottom='y_offset',
    left='x_offset',
    right='x_offset'
)

_WHICH_SIGN = dict(top=1, bottom=-1, left=-1, right=1)

def _effective_side(sprite, direction):
    return (getattr(sprite, direction) -
            _WHICH_SIGN[direction] *
           getattr(sprite, _WHICH_OFFSET[direction], 0))

def _extreme_side(sprite1, sprite2, direction):
    sign = -_WHICH_SIGN[direction]
    return sign * max(sign * _effective_side(sprite1, direction),
                      sign * _effective_side(sprite2, direction))

def collide(sprite1, sprite2):
    return (_extreme_side(sprite1, sprite2, 'bottom') <
            _extreme_side(sprite1, sprite2, 'top')
            and
            _extreme_side(sprite1, sprite2, 'left') <
            _extreme_side(sprite1, sprite2, 'right'))
```

Links

[1] https://ppb.dev/
[2] https://opensource.com/article/19/10/why-love-python
[3] https://opensource.com/article/18/3/getting-started-jupyter-notebooks
[4] https://github.com/moshez/penguin-bit-by-bit/blob/master/dynamic_penguin.ipynb
[5] https://opensource.com/article/20/4/install-python-linux
[6] https://opensource.com/article/19/5/python-3-default-mac
[7] https://opensource.com/article/19/8/how-install-python-windows
[8] https://git-scm.com/download
[9] https://opensource.com/article/19/6/python-virtual-environments-mac
[10] https://docs.python.org/3/library/asyncio-eventloop.html
[11] https://opensource.com/article/20/3/treq-python
[12] https://kenney.nl/
[13] https://en.wikipedia.org/wiki/Linear_interpolation
[14] https://en.wikipedia.org/wiki/Collision_detection
[15] https://docs.python.org/3/library/functions.html#setattr
[16] https://docs.python.org/3/library/stdtypes.html#definition.__name__

Create a slide deck using Jupyter Notebooks

Jupyter may not be the most straightforward way to create presentation slides and handouts, but it affords more control than simpler options.

THERE ARE MANY OPTIONS when it comes to creating slides for a presentation. There are straightforward ways, and generating slides directly from Jupyter [1] is not one of them. But I was never one to do things the easy way. I also have high expectations that no other slide-generation software quite meets.

Why transition from slides to Jupyter?

I want four features in my presentation software:

1. An environment where I can run the source code to check for errors
2. A way to include speaker notes but hide them during the presentation
3. To give attendees a useful handout for reading
4. To give attendees a useful handout for exploratory learning

There is nothing more uncomfortable about giving a talk than having someone in the audience point out that there is a coding mistake on one of my slides. Often, it's misspelling a word, forgetting a return statement, or doing something else that becomes invisible as soon as I leave my development environment, where I have a linter [2] running to catch these mistakes.

After having one too many of these moments, I decided to find a way to run the code directly from my slide editor to make sure it is correct. There are three "gotchas" I needed to consider in my solution:

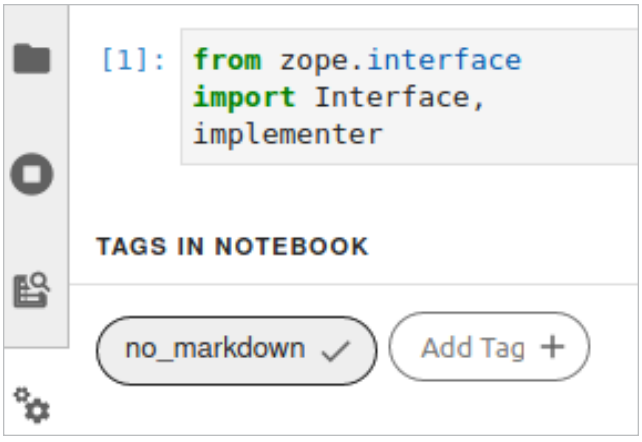
- A lot of code is boring. Nobody cares about three slides worth of `import` statements, and my hacks to mock out the socket module distract from my point. But it's essential that I can test the code without creating a network outage.
- Including boilerplate code is *almost* as boring as hearing me read words directly off of the slide. We have all heard (or even given) talks where there are three bullet points, and the presenter reads them verbatim. I try to avoid this behavior by using speaker notes.
- There is nothing more annoying to the audience when the talk's reference material doesn't have any of the speaker notes. So I want to generate a beautiful handout containing all of my notes and the slides from the same source. Even better, I don't want to have slides on one handout and a separate GitHub repository for the source code.

As is often the case, to solve this issue, I found myself reaching for JupyterLab [3] and its notebook management capabilities.

Using Jupyter Notebooks for presentations

I begin my presentations by using Markdown and code blocks in a Jupyter Notebook, just like I would for anything else in JupyterLab. I write out my presentation using separate Markdown sections for the text I want to show on the slides and for the speaker notes. Code snippets go into their own blocks, as you would expect.

Because you can add a "tag" to cells, I tag any cell that has "boring" code as no_markdown.



(Moshe Zadka, CC BY-SA 4.0)

Then I convert my Notebook to Markdown with:

```
$ jupyter nbconvert presentation.ipynb --to markdown
  --TagRemovePreprocessor.remove_cell_tags='{"no_markdown"}'
  --output build/presentation.md
```

There are ways to convert Markdown to slides [4]—but I have no idea how to use any of them and even less desire to learn. Plus, I already have my favorite presentation-creation tool: Beamer [5].

But Beamer requires custom LaTeX, and that is not usually generated when you convert Markdown to LaTeX. Thankfully, one Markdown implementation–Pandoc Markdown [6]—has a feature that lets me do what I want. Its raw_attribute [7] extension allows including "raw" bits of the target format in the Markdown.

This means if I run `pandoc` on the Markdown export from a notebook that includes `raw_attribute` LaTeX bits, I can have as much control over the LaTeX as I want:

```
$ pandoc --listings -o build/presentation.tex build/presentation.md
```

The `--listings` makes `pandoc` use LaTeX's listings package, which makes code look much prettier. Putting those two pieces together, I can generate LaTeX from the notebook.

Through a series of conversion steps, I was able to hide the parts I wanted to hide by using:

- LaTeX `raw_attribute` bits inside Jupyter Notebook's Markdown cells
- Tagging boring cells as `no_markdown`
- Jupyter's "nbconvert" to convert the notebook to Markdown
- Pandoc to convert the Markdown to LaTeX while interpolating the `raw_attribute` bits
- Beamer to convert the Pandoc output to a PDF slide-deck
- Beamer's beamerarticle mode

All combined with a little bit of duct-tape, in the form of a UNIX shell script, to produce slide-deck creation software. Ultimately, this pipeline works for me. With these tools, or similar, and some light UNIX scripting, you can make your own customized slide created pipeline, optimized to your needs and preferences.

Links

[1] https://jupyter.org/
[2] https://opensource.com/article/19/5/python-flake8
[3] https://jupyterlab.readthedocs.io/en/stable/index.html
[4] https://opensource.com/article/18/5/markdown-slide-generators
[5] https://opensource.com/article/19/1/create-presentations-beamer
[6] https://pandoc.org/MANUAL.html#pandocs-markdown
[7] https://pandoc.org/MANUAL.html#extension-raw_attribute

Build a remote management console using Python and Jupyter Notebooks

Turn Jupyter into a remote administration console.

SECURE SHELL (SSH) is a powerful tool for remote administration, but it lacks some niceties. Writing a full-fledged remote administration console sounds like it would be a lot of work. Surely, someone in the open source community has already written something?

They have, and its name is Jupyter [1]. You might think Jupyter is one of those tools data scientists use to analyze trends in ad clicks over a week or something. This is not wrong—they do, and it is a great tool for that. But that is just scratching its surface.

About SSH port forwarding

Sometimes, there is a server that you can SSH into over port 22. There is no reason to assume you can connect to any other port. Maybe you are SSHing through another "jumpbox" server that has more access or there are host or network firewalls that restrict ports. There are good reasons to restrict IP ranges for access, of course. SSH is a secure protocol for remote management, but allowing anyone to connect to any port is quite unnecessary.

Here is an alternative: Run a simple SSH command with port forwarding to forward a local port to a *remote local* connection. When you run an SSH port-forwarding command like `-L 8111:127.0.0.1:8888`, you are telling SSH to forward your local port `8111` to what the *remote* host thinks `127.0.0.1:8888` is. The remote host thinks `127.0.0.1` is itself.

Just like on *Sesame Street*, "here" is a subtle word.

Click to view video.

The address `127.0.0.1` is how you spell "here" to the network.

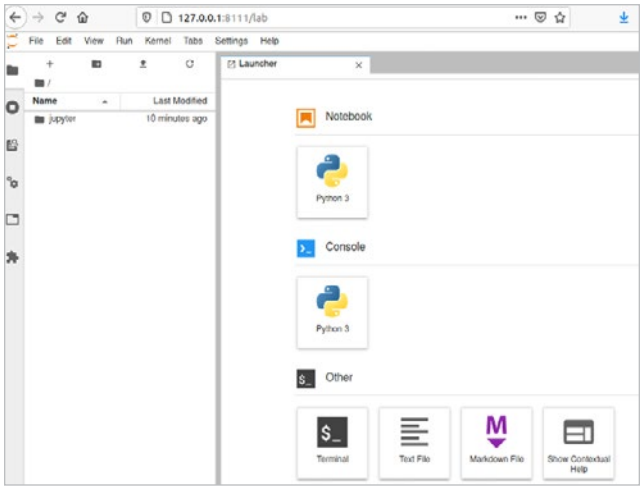
Learn by doing

This might sound confusing, but running this is less complicated than explaining it:

```
$ ssh -L 8111:127.0.0.1:8888 moshez@172.17.0.3
Linux 6ad096502e48 5.4.0-40-generic #44-Ubuntu SMP Tue Jun 23
                                   00:01:04 UTC 2020 x86_64

The programs included with the Debian GNU/Linux system are free
software; the exact distribution terms for each program are
described in the individual files in /usr/share/doc/*/copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the
extent permitted by applicable law.
Last login: Wed Aug  5 22:03:25 2020 from 172.17.0.1
$ jupyter/bin/jupyter lab --ip=127.0.0.1
[I 22:04:29.771 LabApp] JupyterLab application directory is
   /home/moshez/jupyter/share/jupyter/lab
[I 22:04:29.773 LabApp] Serving notebooks from local directory:
   /home/moshez
[I 22:04:29.773 LabApp] Jupyter Notebook 6.1.1 is running at:
[I 22:04:29.773 LabApp] http://127.0.0.1:8888/
   ?token=df91012a36dd26a10b4724d618b2e78cb99013b36bb6a0d1
<MORE STUFF SNIPPED>
```

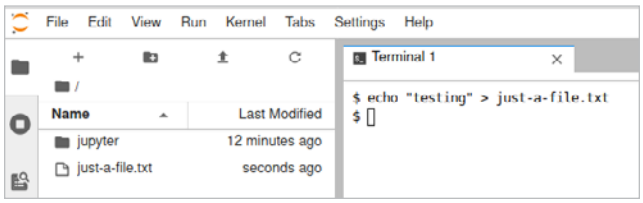
Port-forward `8111` to `127.0.0.1` and start Jupyter on the remote host that's listening on `127.0.0.1:8888`.

Now you need to understand that Jupyter is lying. It thinks you need to connect to port `8888`, but you forwarded that to port `8111`. So, after you copy the URL to your browser, but before clicking Enter, modify the port from `8888` to `8111`:



(Moshe Zadka, CC BY-SA 4.0)

There it is: your remote management console. As you can see, there is a "Terminal" icon at the bottom. Click it to get a terminal:

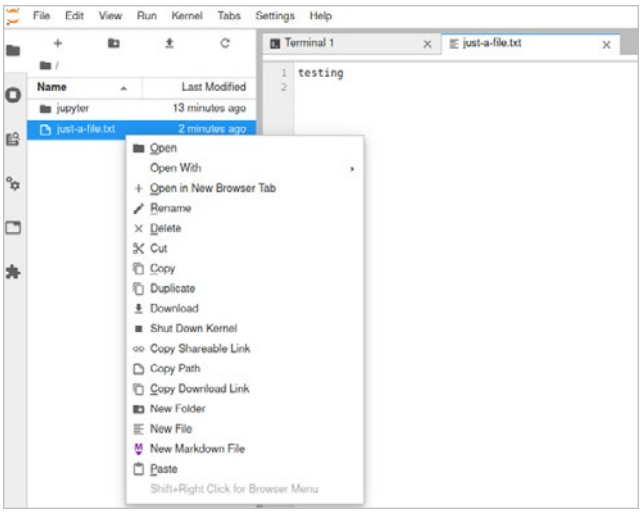


(Moshe Zadka, CC BY-SA 4.0)

You can run a command. Creating a file will show it in the file browser on the side. You can click on that file to open it in an editor that is running locally:

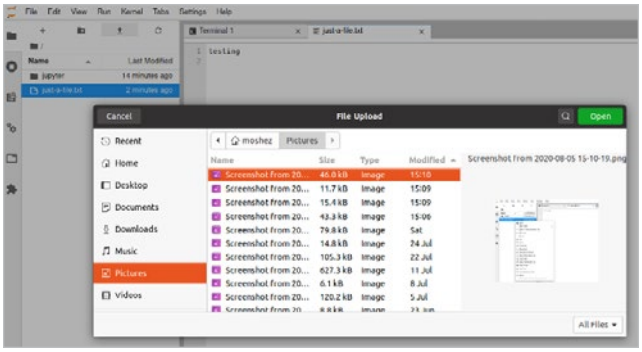
(Moshe Zadka, CC BY-SA 4.0)

You can also download, rename, or delete files:



(Moshe Zadka, CC BY-SA 4.0)

Clicking on the little **Up arrow** will let you upload files. Why not upload the screenshot above?



(Moshe Zadka, CC BY-SA 4.0)

As a nice final tidbit, Jupyter lets you view the remote images directly by double-clicking on them.

Oh, right, and if you want to do systems automation using Python, you can also use Jupyter to open a notebook.

So the next time you need to remotely manage a firewalled environment, why not use Jupyter?

Links

[1] https://jupyter.org/

Edit images with Jupyter and Python

Who needs to learn an image-editing application when you can do the job with open source tools you already know?

RECENTLY, MY KID wanted to make a coloring page from a favorite cartoon. My first thought was to use one of the open source programs on Linux that manipulate images, but then I remembered I have no idea how to use any of them. Luckily, I know how to use Jupyter and Python [1].

How hard can it be, I figured, to use Jupyter for that?

To follow along, you need to have a modern version of Python (if you're a macOS user, you can follow this guide [2]), then install and open Jupyter Labs, which you can learn more about here [3], and Pillow [4], a friendly fork of the Python Imaging Library (PIL), with:

```
$ python -V
Python 3.8.5
$ pip install jupyterlab pillow
# Installation process ommitted
$ jupyter lab
```

Imagine you want to make a coloring page with an image of a deer. The first step is probably to download a picture of a deer and save it locally. Beware of images with dubious copyright status; it's best to use something with a Creative Commons [5] or other open access license. For this example, I used an openly licensed image from Unsplash [6] and named it `deer.jpg`.

Once you're in Jupyter Lab, start by importing `PIL`:

```
from PIL import Image
```

With these preliminaries out of the way, open the image then look at the image size:

```
pic = Image.open("deer.jpg")
pic.size
(3561, 5342)
```

Wow, this is a bit of sticker shock—high-resolution pictures are fine if you want to make a delightful book about deer, but this is probably too big for a homemade coloring book page. Scale it waaaaaaay down. (This kindness is important so that this article loads fast in your browser, too!)

```
x, y = pic.size
x //= 10
y //= 10
smaller = pic.resize((x, y))
```

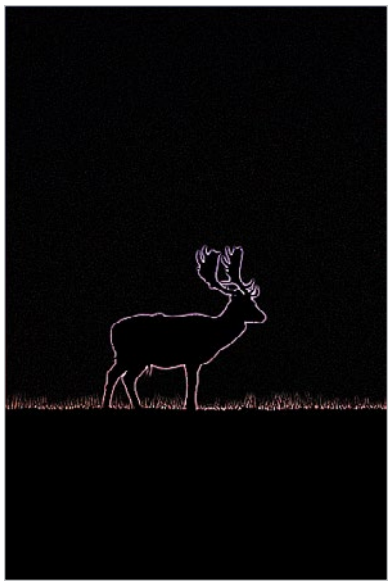
This reduced the scale of the image by 10. See what that looks like:

```
smaller
```

(Max Saeling, Unsplash License)

Beautiful! Majestic and remote, this deer should be a breeze for an edge-detection algorithm. Speaking of which, yes, that's the next step. You want to clean up the image so coloring will be is a breeze, and thankfully there's an algorithm for that. Do some edge detection:

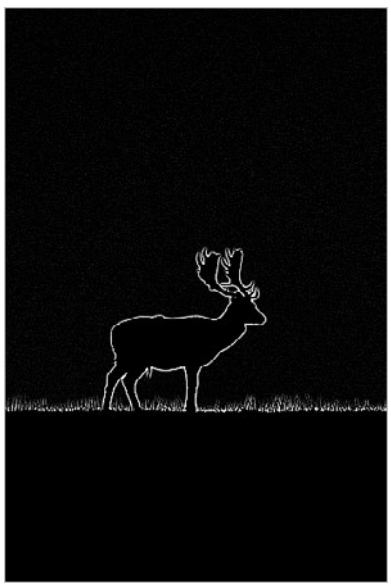
```
from PIL import ImageFilter

edges = smaller.filter(ImageFilter.FIND_EDGES)
edges
```



(Moshe Zadka, CC BY-SA 4.0)

This is probably the most important step. It removes all the extraneous details and leaves clear lines. The color is a little weird, but this is not a hard problem to solve. Split the image into its color bands, and choose the one where the lines are crispest:

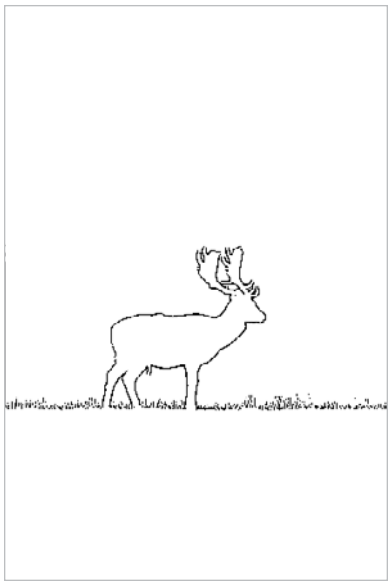
```
bands = edges.split()
bands[0]
```



(Moshe Zadka, CC BY-SA 4.0)

The lines are clear now, but this is not a good image to print because your printer will run out of ink, and your kid will not be happy coloring on black. So invert the colors. While you're at it, snap the colors to max-black or max-white to make the lines even crisper by using a lambda function call [7]:

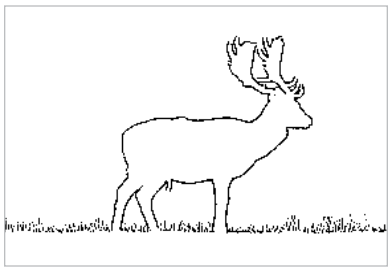
```
outline = bands[0].point(lambda x: 255 if x<100 else 0)
outline
```



(Moshe Zadka, CC BY-SA 4.0)

The original image had a lot of nature that I mercilessly cleared. Now there's a lot of empty space, so crop the picture to the essentials:

```
outline.crop((10, 200, 300, 400))
```



(Moshe Zadka, CC BY-SA 4.0)

All that's left is to save the picture as something easy to print, like a PDF:

```
outline.save("deer.pdf")
```

I'll let you figure out how to print from Linux [8].

Have fun making homemade coloring books for your kids!

Links

[1] https://opensource.com/resources/python
[2] https://opensource.com/article/19/5/python-3-default-mac
[3] https://opensource.com/article/19/5/jupyterlab-python-developers-magic
[4] https://pillow.readthedocs.io/en/stable/installation.html
[5] https://creativecommons.org/
[6] https://unsplash.com/photos/Xu1xO3mpr1I
[7] https://opensource.com/article/19/10/python-programming-paradigms
[8] https://opensource.com/article/18/11/choosing-printer-linux

Teach Python with Jupyter Notebooks

With Jupyter, PyHamcrest, and a little duct tape of a testing harness, you can teach any Python topic that is amenable to unit testing.

SOME THINGS ABOUT THE RUBY community have always impressed me. Two examples are the commitment to testing and the emphasis on making it easy to get started. The best example of both is Ruby Koans [1], where you learn Ruby by fixing tests.

With the amazing tools we have for Python, we should be able to do something even better. We can. Using Jupyter Notebook [2], PyHamcrest [3], and just a little bit of duct tape-like code, we can make a tutorial that includes teaching, code that works, and code that needs fixing.

First, some duct tape. Usually, you do your tests using some nice command-line test runner, like pytest [4] or virtue [5]. Usually, you do not even run it directly. You use a tool like tox [6] or nox [7] to run it. However, for Jupyter, you need to write a little harness that can run the tests directly in the cells.

Luckily, the harness is short, if not simple:

```
import unittest

def run_test(klass):
    suite = unittest.TestLoader().loadTestsFromTestCase(klass)
    unittest.TextTestRunner(verbosity=2).run(suite)
    return klass
```

Now that the harness is done, it's time for the first exercise.

In teaching, it is always a good idea to start small with an easy exercise to build confidence.

So why not fix a really simple test?

```
@run_test
class TestNumbers(unittest.TestCase):

    def test_equality(self):
        expected_value = 3 # Only change this line
        self.assertEqual(1+1, expected_value)
    test_equality (__main__.TestNumbers) ... FAIL

    ======================================================================
    FAIL: test_equality (__main__.TestNumbers)
    ----------------------------------------------------------------------
    Traceback (most recent call last):
      File "<ipython-input-7-5ebe25bc00f3>", line 6, in test_
        equality
        self.assertEqual(1+1, expected_value)
    AssertionError: 2 != 3

    ----------------------------------------------------------------------
    Ran 1 test in 0.002s

    FAILED (failures=1)
```

`Only change this line` is a useful marker for students. It shows exactly what needs to be changed. Otherwise, students could fix the test by changing the first line to `return`.

In this case, the fix is easy:

```
@run_test
class TestNumbers(unittest.TestCase):

    def test_equality(self):
        expected_value = 2 # Fixed this line
        self.assertEqual(1+1, expected_value)
    test_equality (__main__.TestNumbers) ... ok

    ----------------------------------------------------------------------
    Ran 1 test in 0.002s

    OK
```

Quickly, however, the `unittest` library's native assertions will prove lacking. In `pytest`, this is fixed with rewriting the bytecode in `assert` to have magical properties and all kinds of heuristics. This would not work easily in a Jupyter notebook. Time to dig out a good assertion library: PyHamcrest:

```
from hamcrest import *
@run_test
class TestList(unittest.TestCase):

    def test_equality(self):
        things = [1,
                  5, # Only change this line
                  3]
        assert_that(things, has_items(1, 2, 3))
    test_equality (__main__.TestList) ... FAIL
```

```
======================================================================
FAIL: test_equality (__main__.TestList)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "<ipython-input-11-96c91225ee7d>", line 8, in test_
    equality
    assert_that(things, has_items(1, 2, 3))
AssertionError:
Expected: (a sequence containing <1> and a sequence
  containing <2> and a sequence containing <3>)
     but: a sequence containing <2> was <[1, 5, 3]>


----------------------------------------------------------------------
Ran 1 test in 0.004s

FAILED (failures=1)
```

PyHamcrest is not just good at flexible assertions; it is also good at clear error messages. Because of that, the problem is plain to see: [1, 5, 3] does not contain 2, and it looks ugly besides:

```
@run_test
class TestList(unittest.TestCase):

    def test_equality(self):
        things = [1,
                  2, # Fixed this line
                  3]
        assert_that(things, has_items(1, 2, 3))
```

```
test_equality (__main__.TestList) ... ok

----------------------------------------------------------------------
Ran 1 test in 0.001s

OK
```

With Jupyter, PyHamcrest, and a little duct tape of a testing harness, you can teach any Python topic that is amenable to unit testing.

For example, the following can help show the differences between the different ways Python can strip whitespace from a string:

```
source_string = "  hello world  "

@run_test
class TestList(unittest.TestCase):

    # This one is a freebie: it already works!
    def test_complete_strip(self):
        result = source_string.strip()
        assert_that(result,
                    all_of(starts_with("hello"), ends_
                    with("world")))

    def test_start_strip(self):
        result = source_string # Only change this line
        assert_that(result,
                    all_of(starts_with("hello"), ends_
                    with("world  ")))

    def test_end_strip(self):
        result = source_string # Only change this line
        assert_that(result,
                    all_of(starts_with("  hello"), ends_
                    with("world")))
```

```
test_complete_strip (__main__.TestList) ... ok
test_end_strip (__main__.TestList) ... FAIL
test_start_strip (__main__.TestList) ... FAIL

======================================================================
FAIL: test_end_strip (__main__.TestList)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "<ipython-input-16-3db7465bd5bf>", line 19, in
    test_end_strip
    assert_that(result,
AssertionError:
Expected: (a string starting with '  hello' and a string
          ending with 'world')
     but: a string ending with 'world' was '  hello world  '


======================================================================
FAIL: test_start_strip (__main__.TestList)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "<ipython-input-16-3db7465bd5bf>", line 14, in test_
    start_strip
    assert_that(result,
AssertionError:
Expected: (a string starting with 'hello' and a string
  ending with 'world  ')
     but: a string starting with 'hello' was '  hello world  '


----------------------------------------------------------------------
Ran 3 tests in 0.006s

FAILED (failures=2)
```

Ideally, students would realize that the methods `.lstrip()` and `.rstrip()` will do what they need. But if they do not and instead try to use `.strip()` everywhere:

```
source_string = "  hello world  "
@run_test
class TestList(unittest.TestCase):

    # This one is a freebie: it already works!
    def test_complete_strip(self):
        result = source_string.strip()
        assert_that(result,
                    all_of(starts_with("hello"), ends_
                    with("world")))

    def test_start_strip(self):
        result = source_string.strip() # Changed this line
        assert_that(result,
                    all_of(starts_with("hello"), ends_
                    with("world  ")))

    def test_end_strip(self):
        result = source_string.strip() # Changed this line
        assert_that(result,
                    all_of(starts_with("  hello"), ends_
                    with("world")))
    test_complete_strip (__main__.TestList) ... ok
    test_end_strip (__main__.TestList) ... FAIL
    test_start_strip (__main__.TestList) ... FAIL

    ======================================================================
    FAIL: test_end_strip (__main__.TestList)
    ----------------------------------------------------------------------
    Traceback (most recent call last):
      File "<ipython-input-17-6f9cfa1a997f>", line 19, in
        test_end_strip
        assert_that(result,
    AssertionError:
    Expected: (a string starting with '  hello' and a string
              ending with 'world')
         but: a string starting with '  hello' was 'hello world'

    ======================================================================
    FAIL: test_start_strip (__main__.TestList)
    ----------------------------------------------------------------------
    Traceback (most recent call last):
      File "<ipython-input-17-6f9cfa1a997f>", line 14, in test_
        start_strip
        assert_that(result,
    AssertionError:
    Expected: (a string starting with 'hello' and a string
              ending with 'world  ')
         but: a string ending with 'world  ' was 'hello world'

    ----------------------------------------------------------------------
    Ran 3 tests in 0.007s

    FAILED (failures=2)
```

They would get a different error message that shows too much space has been stripped:

```
source_string = "  hello world  "

@run_test
class TestList(unittest.TestCase):

    # This one is a freebie: it already works!
    def test_complete_strip(self):
        result = source_string.strip()
        assert_that(result,
                    all_of(starts_with("hello"), ends_
                    with("world")))

    def test_start_strip(self):
        result = source_string.lstrip() # Fixed this line
        assert_that(result,
                    all_of(starts_with("hello"), ends_
                    with("world  ")))

    def test_end_strip(self):
        result = source_string.rstrip() # Fixed this line
        assert_that(result,
                    all_of(starts_with("  hello"), ends_
                    with("world")))
    test_complete_strip (__main__.TestList) ... ok
    test_end_strip (__main__.TestList) ... ok
    test_start_strip (__main__.TestList) ... ok

    ----------------------------------------------------------------------
    Ran 3 tests in 0.005s

    OK
```

In a more realistic tutorial, there would be more examples and more explanations. This technique using a notebook with some examples that work and some that need fixing can work for real-time teaching, a video-based class, or even, with a lot more prose, a tutorial the student can complete on their own.

Now go out there and share your knowledge!

Links

[1] https://github.com/edgecase/ruby_koans
[2] https://jupyter.org/
[3] https://github.com/hamcrest/PyHamcrest
[4] https://docs.pytest.org/en/stable/
[5] https://github.com/Julian/Virtue
[6] https://tox.readthedocs.io/en/latest/
[7] https://nox.thea.codes/en/stable/

Improve your time management with Jupyter

Discover how you are spending time by parsing your calendar with Python in Jupyter.

PYTHON [1] has incredibly scalable options for exploring data. With Pandas [2] or Dask [3], you can scale Jupyter [4] up to big data. But what about small data? Personal data? Private data?

JupyterLab and Jupyter Notebook provide a great environment to scrutinize my laptop-based life.

My exploration is powered by the fact that almost every service I use has a web application programming interface (API). I use many such services: a to-do list, a time tracker, a habit tracker, and more. But there is one that almost everyone uses: a calendar. The same ideas can be applied to other services, but calendars have one cool feature: an open standard that almost all web calendars support: CalDAV.

Parsing your calendar with Python in Jupyter

Most calendars provide a way to export into the `CalDAV` format. You may need some authentication for accessing this private data. Following your service's instructions should do the trick. How you get the credentials depends on your service, but eventually, you should be able to store them in a file. I store mine in my root directory in a file called `.caldav`:

```
import os
with open(os.path.expanduser("~/.caldav")) as fpin:
    username, password = fpin.read().split()
```

Never put usernames and passwords directly in notebooks! They could easily leak with a stray `git push`.

The next step is to use the convenient PyPI caldav [5] library. I looked up the CalDAV server for my email service (yours may be different):

```
import caldav
client = caldav.DAVClient(url="https://caldav.fastmail.com/dav/",
        username=username, password=password)
```

CalDAV has a concept called the `principal`. It is not important to get into right now, except to know it's the thing you use to access the calendars:

```
principal = client.principal()
calendars = principal.calendars()
```

Calendars are, literally, all about time. Before accessing events, you need to decide on a time range. One week should be a good default:

```
from dateutil import tz
import datetime
now = datetime.datetime.now(tz.tzutc())
since = now - datetime.timedelta(days=7)
```

Most people use more than one calendar, and most people want all their events together. The `itertools.chain.from_iterable` makes this straightforward:

```
import itertools

raw_events = list(
    itertools.chain.from_iterable(
        calendar.date_search(start=since, end=now, expand=True)
        for calendar in calendars
    )
)
```

Reading all the events into memory is important, and doing so in the API's raw, native format is an important practice. This means that when fine-tuning the parsing, analyzing, and displaying code, there is no need to go back to the API service to refresh the data.

But "raw" is not an understatement. The events come through as strings in a specific format:

```
print(raw_events[12].data)
    BEGIN:VCALENDAR
    VERSION:2.0
    PRODID:-//CyrusIMAP.org/Cyrus
     3.3.0-232-g4bdb081-fm-20200825.002-g4bdb081a//EN
    BEGIN:VEVENT
    DTEND:20200825T230000Z
```

```
    DTSTAMP:20200825T181915Z
    DTSTART:20200825T220000Z
    SUMMARY:Busy
    UID:
     1302728i-040000008200E00074C5B7101A82E00800000000D939773 \
       EA578D601000000000
     000000010000000CD71CC3393651B419E9458134FE840F5
    END:VEVENT
    END:VCALENDAR
```

Luckily, PyPI comes to the rescue again with another helper library, vobject [6]:

```
import io
import vobject

def parse_event(raw_event):
    data = raw_event.data
    parsed = vobject.readOne(io.StringIO(data))
    contents = parsed.vevent.contents
    return contents
parse_event(raw_events[12])
    {'dtend': [<DTEND{}2020-08-25 23:00:00+00:00>],
     'dtstamp': [<DTSTAMP{}2020-08-25 18:19:15+00:00>],
     'dtstart': [<DTSTART{}2020-08-25 22:00:00+00:00>],
     'summary': [<SUMMARY{}Busy>],
     'uid': [<UID{}1302728i-040000008200E00074C5B7101A82E0080 \
            0000000D939773EA578D601000000000000000010000000CD \
            71CC3393651B419E9458134FE840F5>]}
```

Well, at least it's a little better.

There is still some work to do to convert it to a reasonable Python object. The first step is to have a reasonable Python object. The attrs [7] library provides a nice start:

```
import attr
from __future__ import annotations
@attr.s(auto_attribs=True, frozen=True)
class Event:
    start: datetime.datetime
    end: datetime.datetime
    timezone: Any
    summary: str
```

Time to write the conversion code!

The first abstraction gets the value from the parsed dictionary without all the decorations:

```
def get_piece(contents, name):
    return contents[name][0].value
get_piece(_, "dtstart")
    datetime.datetime(2020, 8, 25, 22, 0, tzinfo=tzutc())
```

Calendar events always have a start, but they sometimes have an "end" and sometimes a "duration." Some careful parsing logic can harmonize both into the same Python objects:

```
def from_calendar_event_and_timezone(event, timezone):
    contents = parse_event(event)
    start = get_piece(contents, "dtstart")
    summary = get_piece(contents, "summary")
    try:
        end = get_piece(contents, "dtend")
    except KeyError:
        end = start + get_piece(contents, "duration")
    return Event(start=start, end=end, summary=summary,
                 timezone=timezone)
```

Since it is useful to have the events in your *local* time zone rather than UTC, this uses the local timezone:

```
my_timezone = tz.gettz()
from_calendar_event_and_timezone(raw_events[12], my_timezone)
    Event(start=datetime.datetime(2020, 8, 25, 22, 0,
tzinfo=tzutc()), end=datetime.datetime(2020, 8, 25, 23, 0,
    tzinfo=tzutc()), timezone=tzfile('/etc/localtime'),
    summary='Busy')
```

Now that the events are real Python objects, they really should have some additional information. Luckily, it is possible to add methods retroactively to classes.

But figuring which *day* an event happens is not that obvious. You need the day in the *local* timezone:

```
def day(self):
    offset = self.timezone.utcoffset(self.start)
    fixed = self.start + offset
    return fixed.date()
Event.day = property(day)
print(_.day)
    2020-08-25
```

Events are always represented internally as start/end, but knowing the duration is a useful property. Duration can also be added to the existing class:

```
def duration(self):
    return self.end - self.start
Event.duration = property(duration)
print(_.duration)
    1:00:00
```

Now it is time to convert all events into useful Python objects:

```
all_events = [from_calendar_event_and_timezone(raw_event,
              my_timezone)
              for raw_event in raw_events]
```

All-day events are a special case and probably less useful for analyzing life. For now, you can ignore them:

```
# ignore all-day events
all_events = [event for event in all_events
              if not type(event.start) == datetime.date]
```

Events have a natural order—knowing which one happened first is probably useful for analysis:

```
all_events.sort(key=lambda ev: ev.start)
```

Now that the events are sorted, they can be broken into days:

```
import collections
events_by_day = collections.defaultdict(list)
for event in all_events:
    events_by_day[event.day].append(event)
```

And with that, you have calendar events with dates, duration, and sequence as Python objects.

Reporting on your life in Python

Now it is time to write reporting code! It is fun to have eye-popping formatting with proper headers, lists, important things in bold, etc.

This means HTML and some HTML templating. I like to use Chameleon [8]:

```
template_content = """
<html><body>
<div tal:repeat="item items">
<h2 tal:content="item[0]">Day</h2>
<ul>
    <li tal:repeat="event item[1]"><span tal:replace="event">
     Thing</span></li>
</ul>
</div>
</body></html>"""
```

One cool feature of Chameleon is that it will render objects using its `html` method. I will use it in two ways:

- The summary will be in **bold**
- For most events, I will remove the summary (since this is my personal information)

```
def __html__(self):
    offset = my_timezone.utcoffset(self.start)
    fixed = self.start + offset
    start_str = str(fixed).split("+")[0]
    summary = self.summary
    if summary != "Busy":
        summary = "&lt;REDACTED&gt;"
    return f"<b>{summary[:30]}</b> -- {start_str} ({self.duration})"
Event.__html__ = __html__
```

In the interest of brevity, the report will be sliced into one day's worth.

```
import chameleon
from IPython.display import HTML
template = chameleon.PageTemplate(template_content)
html = template(items=itertools.islice(events_by_day.items(), 3, 4))
HTML(html)
```

When rendered, it will look something like this:

2020-08-25

- **<REDACTED>** -- 2020-08-25 08:30:00 (0:45:00)
- **<REDACTED>** -- 2020-08-25 10:00:00 (1:00:00)
- **<REDACTED>** -- 2020-08-25 11:30:00 (0:30:00)
- **<REDACTED>** -- 2020-08-25 13:00:00 (0:25:00)
- **Busy** -- 2020-08-25 15:00:00 (1:00:00)
- **<REDACTED>** -- 2020-08-25 15:00:00 (1:00:00)
- **<REDACTED>** -- 2020-08-25 19:00:00 (1:00:00)
- **<REDACTED>** -- 2020-08-25 19:00:12 (1:00:00)

Endless options with Python and Jupyter

This only scratches the surface of what you can do by parsing, analyzing, and reporting on the data that various web services have on you.

Why not try it with your favorite service?

Links

[1] https://opensource.com/resources/python
[2] https://pandas.pydata.org/
[3] https://dask.org/
[4] https://jupyter.org/
[5] https://pypi.org/project/caldav/
[6] https://pypi.org/project/vobject/
[7] https://opensource.com/article/19/5/python-attrs
[8] https://chameleon.readthedocs.io/en/latest/

Explore the world of programming with Jupyter

JUPYTERLAB is the next-generation web-based Jupyter [1] user interface. It allows you to work with Jupyter Notebooks [2], as well as editors, terminals, and more, to produce interactive documents for data science, statistical modeling, data visualization, and more.

It has native viewers for PDF, CSV, JSON, images, and more. It is also extensible to support other formats.

JupyterLab's left sidebar has tabs for using it as a file manager, a Jupyter kernel manager, or a Jupyter Notebook metadata editor.

Writing code in Jupyter Notebooks enables an interactive development experience. You can write code, see the results, and modify the code—all without restarting your process or losing your in-memory data. This is a great fit for exploratory programming when you are not sure what your end result will look like.

Exploration is common in data science; after all, science is the process of finding out answers not known before. But exploration is not limited to data science. Jupyter works well for system diagnostics and automation where you don't know the answer or solution in advance. Whenever feedback is useful for the next step, whether it is image manipulation, analyzing your exercise data, or writing games, Jupyter's bias toward exploration can be helpful.

Jupyter and JupyterLab are great tools, so this JupyterLab cheat sheet will make it easier for you to get started.

Links

[1] https://jupyter.org/
[2] https://opensource.com/article/18/3/getting-started-jupyter-notebooks

JupyterLab Cheat Sheet

By Moshe Zadka

JupyterLab is a web-based Jupyter user interface.

Install

Install JupyterLab with pip	`python -m pip install jupyterlab`
Install JupyterLab with conda	`conda install -c conda-forge jupyterlab`

Run

```
$ jupyter lab
```

`--ip=<IP>`	Bind to the given IP port
`--port=<PORT>`	Use a different port
`--LabApp.token=<TOKEN>`	Use specific token (do not auto-generate)

```
$ jupyter notebook list
```

`--json`	One JSON per line
`--jsonlist`	A JSON list

Transforming notebooks

Install **nbconvert**

```
$ python -m pip install nbconvert
```

Convert to {html, markdown, latex, script}

```
$ jupyter nbconvert \
  -to <FORMAT> my.ipynb
```

Keyboard shortcuts

`Ctrl+Shift+]`	Next tab
`Ctrl+Shift+[`	Previous tab
`Ctrl+B`	Toggle left bar
`Shift+Enter`	Execute cell
`Ctrl+S`	Save notebook
`Esc`	Enter command mode
`Ctrl+Shift+-`	Split cell at cursor

Command mode

`Up-arrow`	Move one cell up
`Down-arrow`	Move one cell down
`A`	Add cell above
`B`	Add cell below
`C`	Copy cell
`V`	Paste cell
`X`	Cut cell
`Z`	Undo cell operation
`Shift+L`	Toggle cell line numbers

