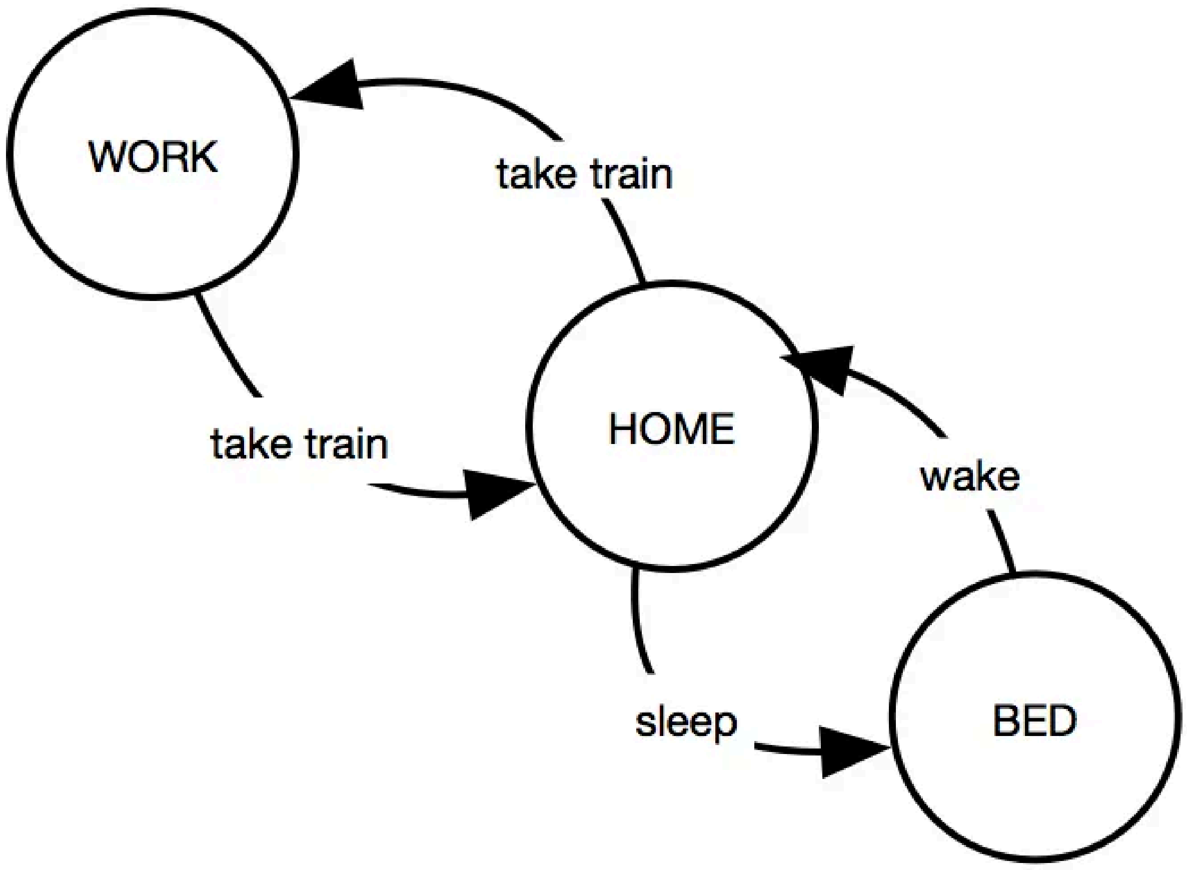
Tutorial: Write a Finite State Machine to parse a custom language in pure Python

Brian Ray Follow 6 min read · Oct 25, 2017

644 5

I was once a huge fan of FSMs (Finite State Machines) as a mechanism to keep track of states. **Automata theory** is the basis of class of computational problems solvable by discrete math. I had used fysom in the past but this time I wanted something home grown. I was able to write complex language parse in a couple hours using only 200 lines of code.



Work, Home, and Bed state machine example

First what is a FSM: This *work, home, and bed* example shows how the transitions work. Each transition changes state. However not every transition is available from every state. For example, you can't sleep at work. WAKE UP!

My Problem is simple: Write a language parser for the syntax I invented call POSH Syntax. You can read all about this syntax and its purpose in a future article yet to be written. Just know it has something with to do with writing rules for detecting parts of speech. Why didn't I use the traditional methods of parsing a language using lex/yacc or even Beazly's PLY? Well the reasons are simple:

1. More straightforward
2. Zero dependencies
3. Ability to have cleaner control over program flow
4. I wanted others to have the ability to read my code

Last time I tried to write a parser was probably 20 years ago and it started looking like a mess of reg-ex and obfuscated logic. This time I wanted it to be clean, I wanted it to be in Python, and I wanted it to be entirely under 200 lines of code. Long story short, this tutorial will outline the steps taken to get me there.

1. Analyze the structure

First a couple simple examples of the POSH Syntax one per line (3 examples):

```
VB(noise+3)
NNS(acoustics) & RB(not)
(NNS(acoustics) & RB(not)) | JJ(muted)
```

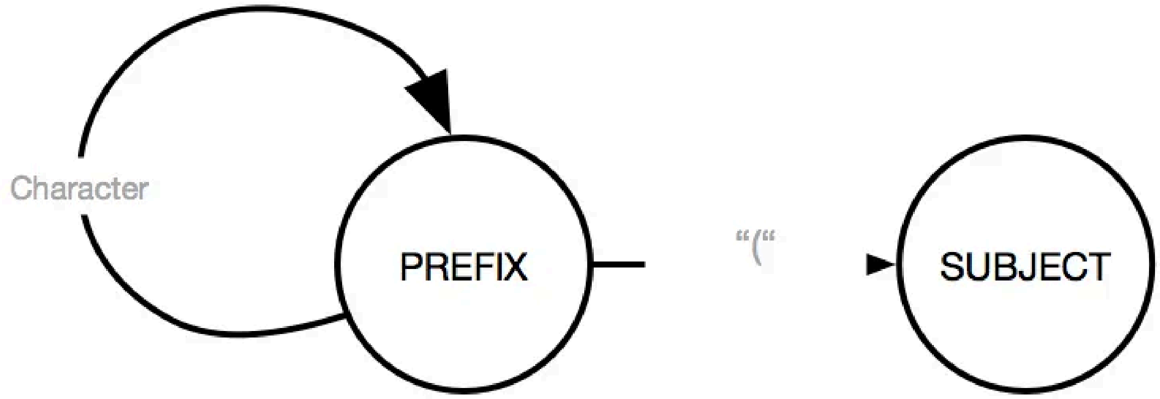
Let's not dive too deep into the meaning of my syntax. Instead we are going to focus on the stream of characters and how they may be parsed. A couple quick observations:

- There is a prefix like "VB" then a parenthesis set.

- Inside the parenthesis is a subject
- each of those prefix + (+ subject +) we can call a Rule
- then there may be an operator that strings rules together like '&' or '|'
- sometimes there are groups of things also encapsulated by a parenthesis.
- if I look left to right, character per character, I observe what options from that spot to the next will modify state

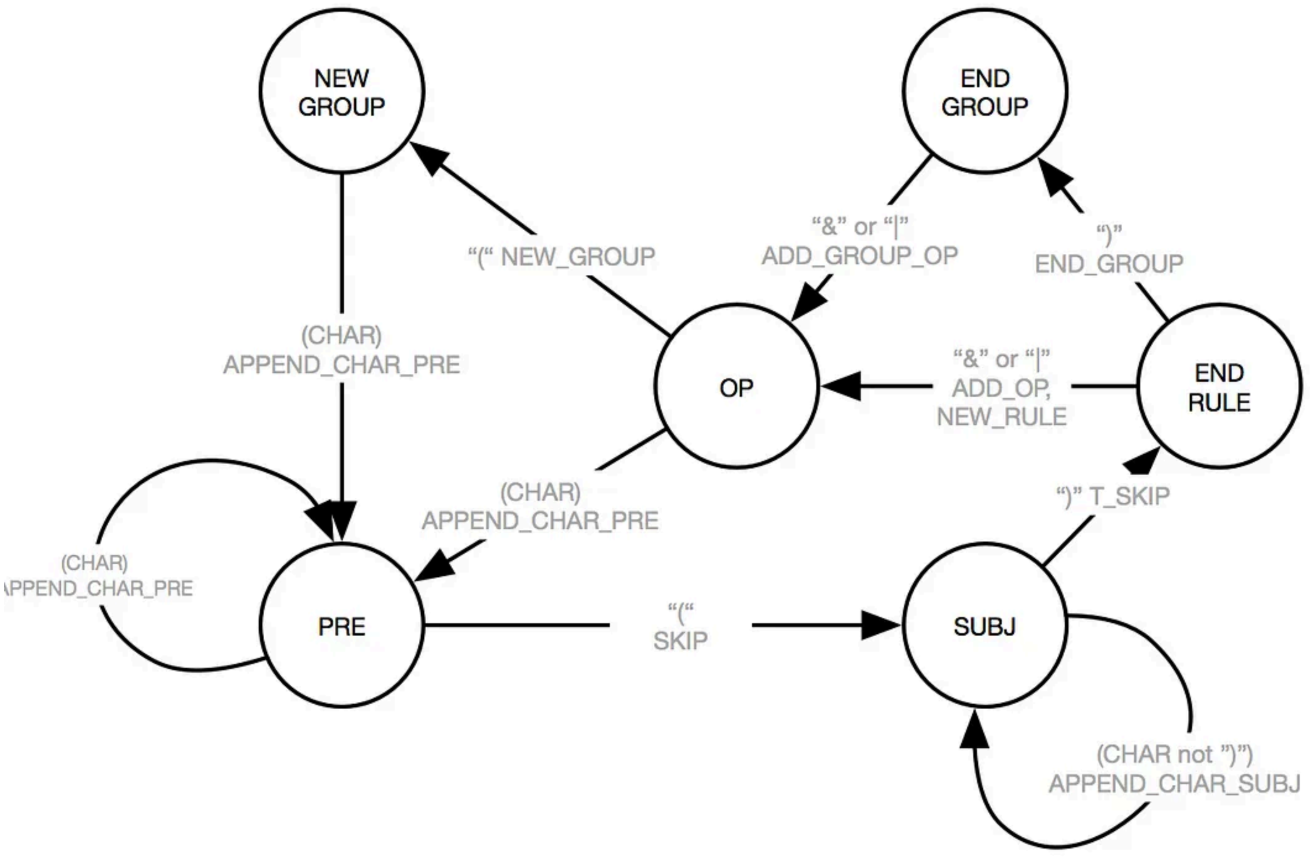
2. Draw a state diagram

It's easiest for me if I simply start with one state and I ask myself, "self, where can I go from here?" For example if start with the state PREFIX while looking at the very first character of the first example the 'V' (in " `VB(noise+3)` ") I see I can only go two places, 1) to another character, in this case the 'B' or 2) I can go to the first '(' parenthesis. If I graph this out it looks like this:



Here I am saying when I go to the next character I am either still in the prefix or I move to the subject state if I ever to see the "(" character.

We need to do this for every single state. We just keep going character per character and ask ourselves what are the possible options and what are the results of those options. Then end diagram looks like this:



POSH Syntax State Diagram with transitions

Notice I have given each transition a name, each state as well.

> *A note on coverage: This is where TDD (test driven development) comes it. A later goal will be to try to cover (run a "coverage" report) every transition and every state.*

3. List our transitions and states and convert them to code

For the transitions I will give each a short all caps name then a lower case longer name (so I can implement those in Python later):

```
T_SKIP = transition_skip
T_NEW_GROUP = transition_new_group
T_APPEND_CHAR_PRE = transition_append_pre
T_ADD_OP = transition_add_op
T_ADD_OP_NEW_RULE = transition_add_op_new_rule
T_END_GROUP = transition_end_group
T_END_RULE = transition_end_rule
T_APPEND_CHAR_SUBJ = transition_append_subj
T_ADD_GROUP_OP = transition_add_op_new_group
```

Now for the states, give each one a short name and then a longer full text name to help identify them.

```
S_NEW_GROUP = “STATE: NEW_GROUP”
S_END_GROUP = “STATE: END_GROUP”
S_PRE = “STATE: PREFIX”
S_OP = “STATE: OPERATOR”
S_END_RULE = “STATE: END_RULE”
S_SUBJ = “STATE: SUBJECT”
```

4. Create a transition table of state changes

Our transition table must contain one entry for each:

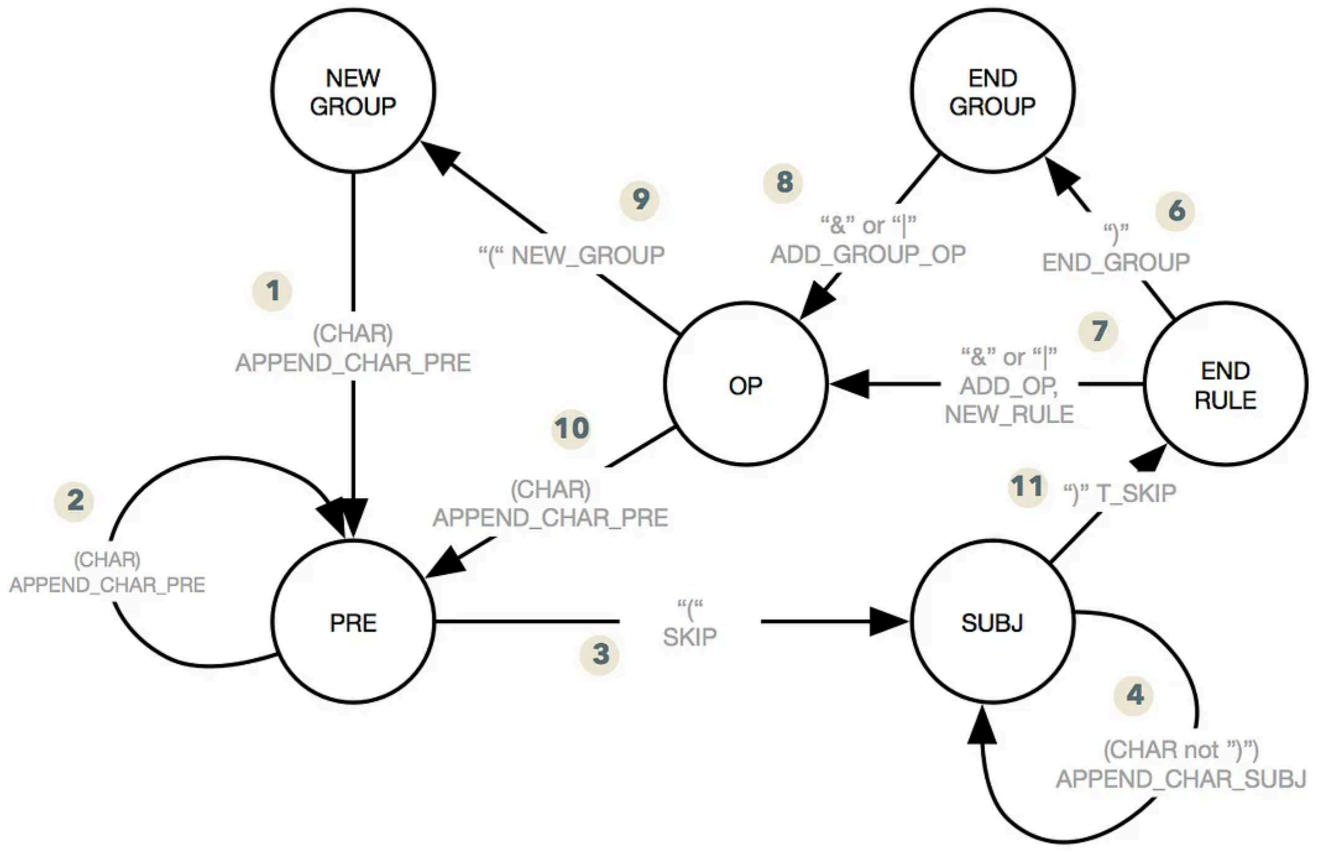
- start state (src)
- end state (dst)
- rule for transition (condition)

- callback for the transition (callback)

In Python it will look something like this:

```
# For transition 1
{‘src’: S_NEW_GROUP,
 ‘dst’: S_PRE,
 ‘condition’: “[A-Za-z|+|-|\d]”,
 ‘callback’: T_APPEND_CHAR_PRE}
```

Now we number all of our possible stage changes so we are sure we don’t miss one.



POSH Syntax State Diagram with transitions

The end table looks something like this:

```
FSM_MAP = (
    #  {'src':, 'dst':, 'condition':, 'callback': },
    {'src': S_NEW_GROUP,
        'dst': S_PRE,
        'condition': "[A-Za-z|+|-|\d]",
        'callback': T_APPEND_CHAR_PRE},  # 1
    {'src': S_PRE,
        'dst': S_PRE,
        'condition': "[A-Za-z|+|-|\d]",
        'callback': T_APPEND_CHAR_PRE},  # 2
    {'src': S_PRE,
        'dst': S_SUBJ,
        'condition': "\(",
        'callback': T_SKIP},  # 3
    {'src': S_SUBJ,
        'dst': S_SUBJ,
        'condition': "[^\)]",
        'callback': T_APPEND_CHAR_SUBJ},  # 4
    {'src': S_SUBJ,
        'dst': S_END_RULE,
        'condition': "\)",
        'callback': T_END_RULE},  # 5
    {'src': S_END_RULE,
        'dst': S_END_GROUP,
        'condition': "\)",
        'callback': T_END_GROUP},  # 6
    {'src': S_END_RULE,
        'dst': S_OP,
        'condition': "[\&|\|]",
        'callback': T_ADD_OP_NEW_RULE},  # 7
    {'src': S_END_GROUP,
        'dst': S_OP,
        'condition': "[\&|\|]",
        'callback': T_ADD_GROUP_OP},  # 8
```

```
        'dst': S_PRE,
        'condition': "[A-Za-z|+|-|\d]",
        'callback': T_APPEND_CHAR_PRE},  # 10
    {'src': S_SUBJ,
        'dst': S_END_RULE,
        'condition': "\)",
           'callback': T_END_RULE})  # 11
```

trans_table.py hosted with ❤ by GitHub view raw

5. Complete our program class design

We start with three class:

1. A Rule class: It will hold our Prefix, Suffix, and operator (left hand side)
2. A RuleGroup: This will have a parent RuleGroup — poor man's pushdown automata
3. And the Rule_Parse_FSM class that main goal is to iterate over the input string and handle the transitions while callling the callbacks.

Rule Class:

RuleGroup Class:

And finally the Rule_Parse_FSM class:

A quick overview of the parse logic:

- Run take the input string character by character and sends it to process next.
- process takes only the relevant transitions (those with matching states to current state) and sends the character to an evaluator

- The evaluator (iterate_re_evaluators) takes the cmd logic (which happens to be a very short regex statement) I stress Very short and not confusing, I hope. If it matches it fires a state change
- The state changer stores a new stage and calls the call backs

Now we need callbacks that do something different depending on the transition from state to to state. We already outlined them but we need to now write each. Here is what we end up with:

It’s very convenient to pass in the fsm_obj which is an instance of Rule_Parse_FSM class. Each transition does something different. Let’s take transition_add_op for example. That is when we hit a operator like the ’&’ or the ‘|’ we find the Rule it belongs to (which will eventually be the complete left hand side of the rule) and assign that operator to the instance.

6. Test our Program

The complete program may be located here.

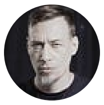
Running this files gives us:

```
N -> STATE: NEW_GROUP : STATE: PREFIX
N -> STATE: PREFIX : STATE: PREFIX
( -> STATE: PREFIX : STATE: SUBJECT
h -> STATE: SUBJECT : STATE: SUBJECT
e -> STATE: SUBJECT : STATE: SUBJECT
l -> STATE: SUBJECT : STATE: SUBJECT
l -> STATE: SUBJECT : STATE: SUBJECT
o -> STATE: SUBJECT : STATE: SUBJECT
) -> STATE: SUBJECT : STATE: END_RULE
skip ' ' in STATE: END_RULE
& -> STATE: END_RULE : STATE: OPERATOR
skip ' ' in STATE: OPERATOR
N -> STATE: OPERATOR : STATE: PREFIX
N -> STATE: PREFIX : STATE: PREFIX
( -> STATE: PREFIX : STATE: SUBJECT
w -> STATE: SUBJECT : STATE: SUBJECT
o -> STATE: SUBJECT : STATE: SUBJECT
r -> STATE: SUBJECT : STATE: SUBJECT
l -> STATE: SUBJECT : STATE: SUBJECT
d -> STATE: SUBJECT : STATE: SUBJECT
) -> STATE: SUBJECT : STATE: END_RULE

<RuleGroup: {'rules': [<Rule:  NN(hello)>, <Rule: & NN(world)>],
'level': 0, 'rule_count': 2, 'parent': None, 'op': None}>
```

Now we have completely implemented our state machine in 200 lines of Python code. Yay! 🍺

Again the final code may be located HERE

Programming Finite State Machine Python Computer Science Mathematics



Written by Brian Ray

1K followers · 354 following

Follow

Long time Python-isto, Inquisitor, Solver, Data Science in Cognitive/AI/Machine Learning Frequent Flyer

Responses (5)

Write a response

What are your thoughts?

Nok Chan
Apr 19, 2018 (edited)

Where is Stage 5 in the diagram? Did I missed something?

One question, if all the transition functions are expecting a Rule_Parse_FSM class, what is the reason not include it in the class as a method?

1 1 reply Reply

Ben Siewert
Aug 21, 2019

Nice article man! I am new to Finite State Machines.

I was looking into how to manage the application state of a Python program and just stumbled across this page.

So, I have more experience in the Front-End world where developers often use Redux to... more

Reply

Yves Cloutier
Nov 6, 2018

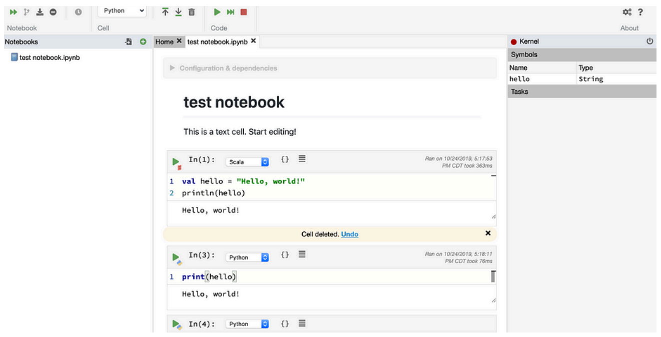
Brian,

Which program did you use for your state diagrams? Looks great!

Reply

See all responses

More from Brian Ray



test notebook

This is a text cell. Start editing!

Name	Type
hello	String

```
In(1):
Ran on 10/24/2019, 5:17:53 PM CDT took 363ms
val hello = "Hello, world!"
println(hello)

Hello, world!
```

Cell deleted. Undo

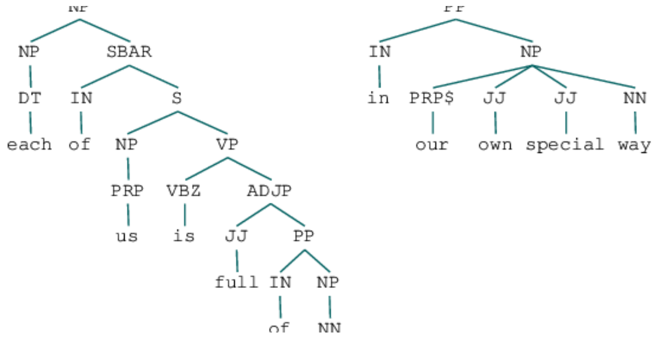
```
In(3):
Ran on 10/24/2019, 5:18:11 PM CDT took 76ms
print(hello)

Hello, world!

In(4):
```

Brian Ray

Did Netflix just kill Databricks with releasing Polynote for free?

Databricks announces $400M round on $6.2B valuation as analytics platform…

Oct 25, 2019 72 1

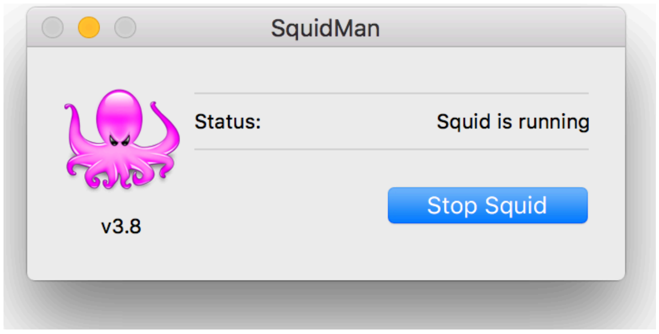


Brian Ray

AI in Practice: Identifying Parts of Speech in Python

In short: computers can at most times correctly identify the context of each word in…

Sep 17, 2017 128 2



Brian Ray

Google Cloud's AutoML first look

A first look into GCP AutoML Tables [beta] and some comparisons between DataRobot,...

Apr 23, 2019 290 8

Brian Ray

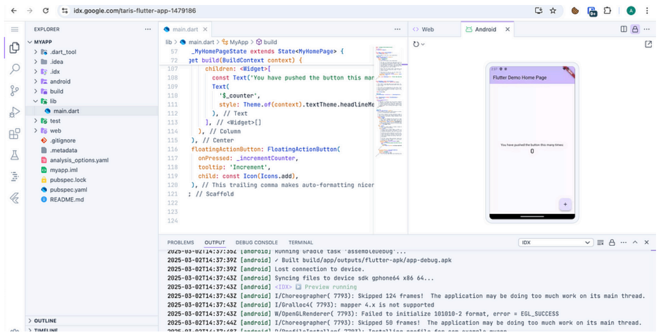
Securely install Jupyter / Python without internet connection in 4...

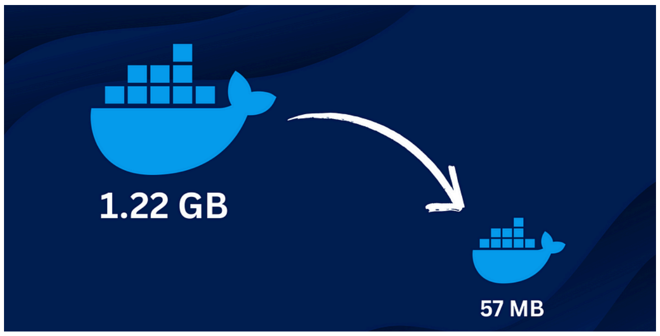
There is often times a reason you don't want a remote server to have internet connection....

Mar 13, 2018 32

See all from Brian Ray

Recommended from Medium





In Coding Beauty by Tari Ibaba

This new IDE from Google is an absolute game changer

This new IDE from Google is seriously revolutionary.

Mar 11 5.3K 311

Prateek Jain

A Step-by-Step Guide to Docker Image Optimisation: Reduce Size...

When I first deployed a Node.js app to production using Docker, I was stunned, The...

6d ago 725 18

Devlink Tips

Why I switched from obsidian: A real developer's story and what I'...

Obsidian was great until it wasn't. Here's the real story, tools I replaced it with, and why yo...

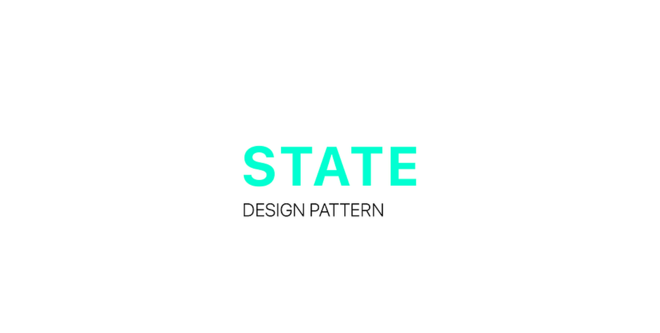
Apr 28 1K 44

Wes O'Donnell

How a Secret Japanese Satellite Deal Just Supercharged Ukraine's...

On April 21, 2025, Japan did something it had never done before: it handed over its most...

4d ago 5.4K 29



Maxim Gorin

Stop Writing If-Else Trees: Use the State Pattern Instead

Tired of messy conditionals? Learn how the State pattern makes your code cleaner,...

LORY

Math Explained to Programmers: Inverse Matrix

Adjugate matrix reminds me of the dark days before Git.

Apr 10 2.7K 57

Jan 12 15

See more recommendations