Published in Towards Data Science

Louis Chan Follow

Nov 10, 2020 · 7 min read · · Listen

Efficient Root Searching Algorithms in Python

Implementing efficient searching algorithms for finding roots, and optimisation in Python

Photo by Matteo Grando on Unsplash

As Data Scientists/Computer Scientists, it is quite often that we deal with root-finding algorithms in our daily routines whether or not we realise it. These are algorithms designed for efficiently **locating proximity to a particular value, local/global maxima or minima**.

> We use root-finding algorithm to search for the proximity of a particular value, local / global maxima or minima.

In mathematics, when we say finding a root, it usually means that we are trying to solve a system of equation(s) such that `f(X) = 0`. This makes root-finding algorithms very efficient searching algorithm as well. All we need to do is to define `g(X) = f(X) – Y` where `Y` is our search target and instead solve for `X` such that `g(X) = f(X) – Y = 0`.

There are mainly two different families of approaches: **bracketing approaches** (e.g. bisection algorithm) and **iterative approaches** (e.g. Newton's method, secant method, Steffensen's method, etc.). In this blog, we will be looking into implementing some of these algorithms in python, and compare them against each other (feel free click the algorithm names to jump to different section):

1. Bisection Algorithm
2. Regula-Falsi Algorithm
3. Illinois Algorithm
4. Secant Algorithm
5. Steffensen's Algorithm

Before we start, let's assume that we have a **continuous function** `f` and that we would like to search for a value `y`. i.e. we are solving for `f(x) – y = 0`.

1. Bisection Algorithm

Bisection algorithm, or more famously known for its discrete version (Binary search) or tree variant (Binary search tree), is an efficient algorithm for searching

for a target value within a bound. Because of that, this algorithm is also known as a **bracketing approach to finding a root of an algorithm.**

Key Strength:

- Robust algorithm that guarantees reasonable rate of convergence to the target value

Key Weakness:

- Requires knowledge of the approximate area of root e.g. $3 \leq \pi \leq 4$
- Works well there is only **one root** in the approximate area

Assuming that we know `x` is between `f(a)` and `f(b)`, which forms the bracket of the search. The algorithm will check if `x` is greater than or less than `f((a + b) / 2)`, which is the mid-point of the bracket.

A **margin of error** is required for checking against the mid-point of the bracket. When searching a continuous function, it is possible that we will never be able to locate the exact value (e.g. locating the end of π). A margin of error can be treated as an early stopping when the computed value is close enough to the target value. E.g. if margin of error is 0.001%, 3.141624 is close enough to π, which is approximately 3.1415926...

If the computed value is close enough to the target value, the search is done, otherwise, we search for the value in the bottom half if `x < f((a + b) / 2)` and vice versa.

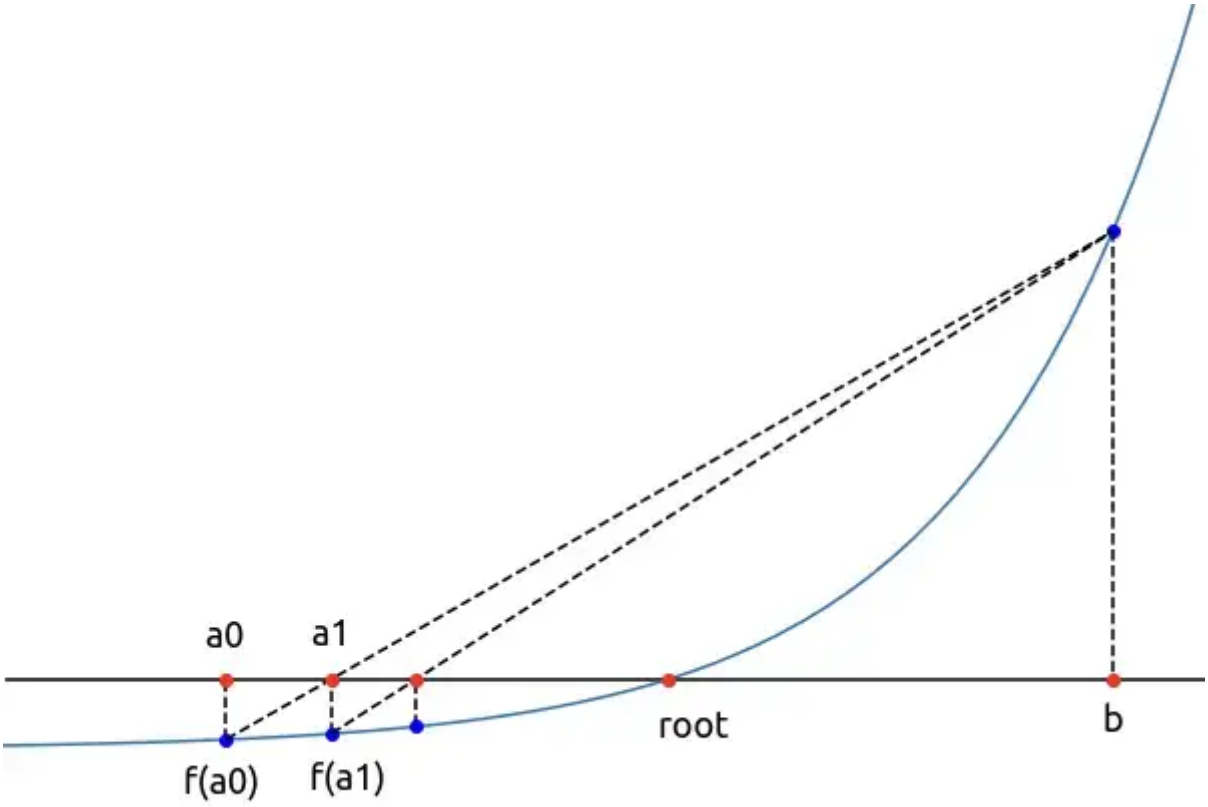
```
def bisection_algorithm(f, a, b, y, margin=.00_001):
    ''' Bracketed approach of Root-finding with bisection method

    Parameters
    ----------
    f: callable, continuous function
    a: float, lower bound to be searched
    b: float, upper bound to be searched
    y: float, target value
    margin: float, margin of error in absolute term

    Returns
    -------
```

```
    A float c, where f(c) is within the margin of y
    '''

    if (lower := f(a)) > (upper := f(b)):
        a, b = b, a
        lower, upper = upper, lower

    assert y >= lower, f"y is smaller than the lower bound. {y} < {lower}"
    assert y <= upper, f"y is larger than the upper bound. {y} > {upper}"

    while 1:
        c = (a + b) / 2
        if abs((y_c := f(c)) - y) < margin:
            # found!
            return c
        elif y < y_c:
            b, upper = c, y_c
        else:
            a, lower = c, y_c
```

bisection_algorithm.py hosted with ❤ by GitHub view raw

2. False Position Algorithm (Regula Falsi)

Just like Bisection algorithm, Regula Falsi also uses a bracketing approach. However, unlike Bisection algorithm, it does not use a brute-force approach of dividing the problem space in half for every iteration. Instead, Regula Falsi iteratively draws a straight line from `f(a)` to `f(b)` and **compares the intercept with the target value.** It is however not guaranteed that the searching efficiency is always improved. For example, the diagram below shows how only the lower bound has been increased in a reducing rate, while the upper bound remains a **stagnant bound.**



Stagnant bound slows down convergence. Image by Author.

Key Strength:

- Often faster convergence than Bisection algorithm

> Regula Falsi takes advantage of that as bracket gets smaller, the continuous function will converge to a straight line.

Key Weakness:

- Slower convergence when algorithm hits a stagnant bound
- Requires knowledge of the approximate area of root e.g. $3 \leq \pi \leq 4$

The key difference in the implementation between Regula Falsi and Bisection is that `c` is no longer the mid-point between `a` and `b`, but instead is being calculated as:

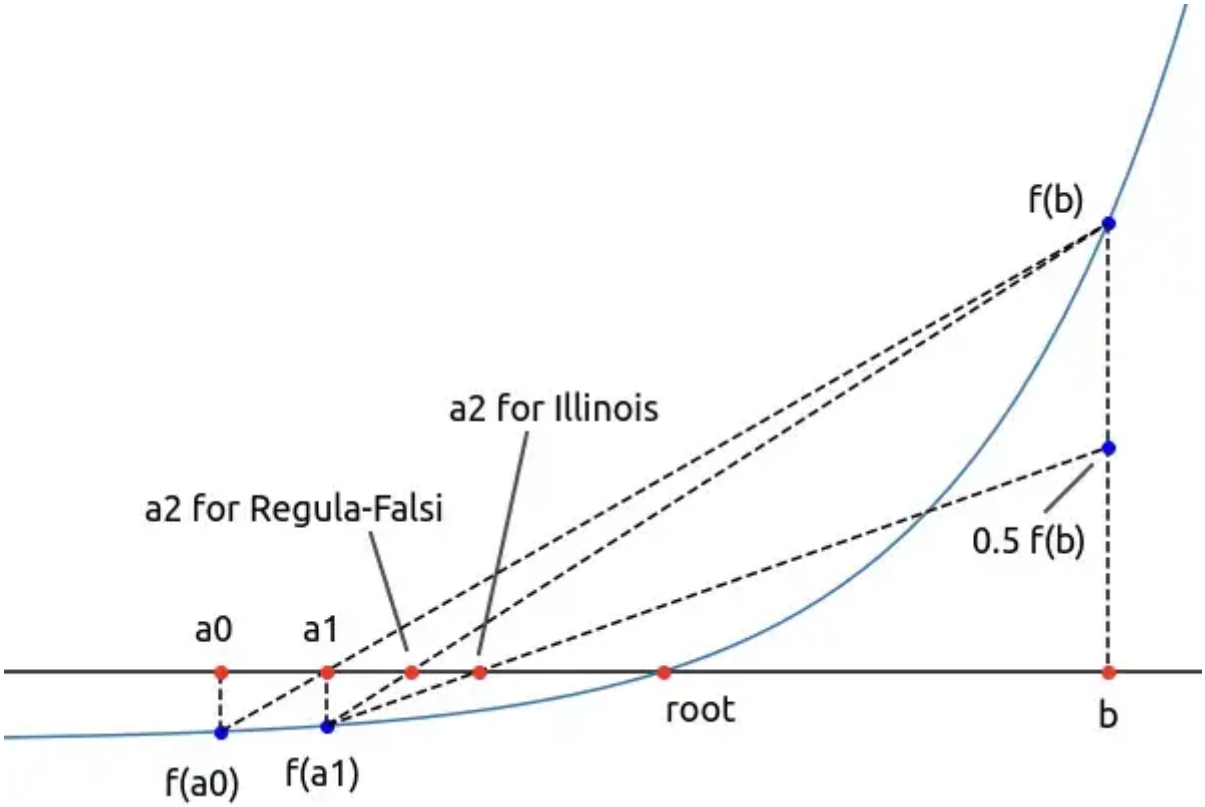
$$c = \frac{af(b) - bf(a)}{f(b) - f(a)}$$

```python
def regula_falsi_algorithm(f, a, b, y, margin=.00_001):
    ''' Bracketed approach of Root-finding with regula-falsi method

    Parameters
    ----------
    f: callable, continuous function
    a: float, lower bound to be searched
    b: float, upper bound to be searched
    y: float, target value
    margin: float, margin of error in absolute term

    Returns
    -------
    A float c, where f(c) is within the margin of y
    '''

    assert y >= (lower := f(a)), f"y is smaller than the lower bound. {y} < {lower}"
    assert y <= (upper := f(b)), f"y is larger than the upper bound. {y} > {upper}"

    while 1:
        c = ((a * (upper - y)) - (b * (lower - y))) / (upper - lower)
        if abs((y_c := f(c)) - y) < margin:
            # found!
            return c
        elif y < y_c:
            b, upper = c, y_c
        else:
            a, lower = c, y_c
```

regula_falsi.py hosted with ❤ by GitHub view raw

3. Illinois Algorithm (Modified Regula-Falsi)

In order to get pass the stagnant bound, we can add in a conditional rule for when a bound remain stagnant for two rounds. Taking the previous example, as `b` has not moved for two round, and that `c` is not close to the root `x` yet, in the next round, the line will be drawn to `f(b) / 2` instead of `f(b)`. Same applies for the lower bound if instead lower bound is the stagnant bound.



Illinois algorithm avoid prolonged stagnant bound for faster convergence. Image by Author.

Key Strength:

- Often faster convergence than both Bisection and Regula-Falsi
- Avoid stagnant bound by halving the stagnant bound's distance to the target value

Key Weakness:

- Slower convergence when algorithm hits a stagnant bound
- Requires knowledge of the approximate area of root e.g. $3 \leq \pi \leq 4$

```
def illinois_algorithm(f, a, b, y, margin=.00_001):
    ''' Bracketed approach of Root-finding with illinois method
    Parameters
    ----------
    f: callable, continuous function
    a: float, lower bound to be searched
    b: float, upper bound to be searched
    y: float, target value
    margin: float, margin of error in absolute term
    Returns
    -------
    A float c, where f(c) is within the margin of y
```

```
    '''

    assert y >= (lower := f(a)), f"y is smaller than the lower bound. {y} < {lower}"
    assert y <= (upper := f(b)), f"y is larger than the upper bound. {y} > {upper}"

    stagnant = 0

    while 1:
        c = ((a * (upper - y)) - (b * (lower - y))) / (upper - lower)
        if abs((y_c := f(c)) - y) < margin:
            # found!
            return c
        elif y < y_c:
            b, upper = c, y_c
            if stagnant == -1:
                # Lower bound is stagnant!
                lower += (y - lower) / 2
            stagnant = -1
        else:
            a, lower = c, y_c
            if stagnant == 1:
                # Upper bound is stagnant!
                upper -= (upper - y) / 2
            stagnant = 1
```

illinois.py hosted with ❤ by GitHub view raw

4. Secant Method (Quasi-Newton's Method)

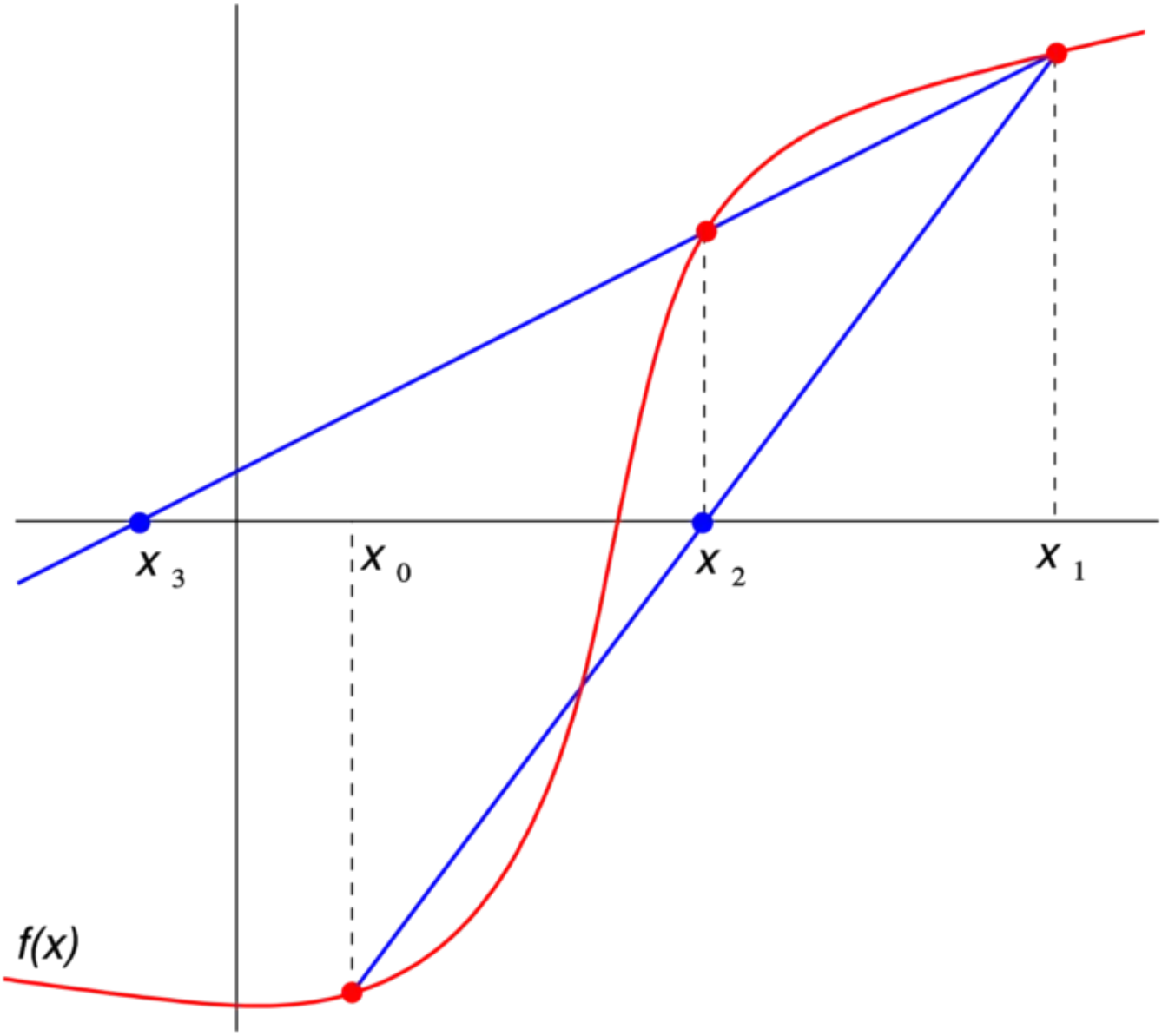
Till now, we have been implementing bracket approaches. What if we **do not know where the brackets are**? Well, then secant method can be very useful. Secant method is an iterative algorithm that starts with two estimates and attempts to converge towards the target value. While we can get a better performance when the algorithm convergence and also that we do not need knowledge of the rough location of root, we may run into **risk of divergence if the two initial estimates are too far away from the actual root.**

Key Strength:

- Does not require a bracket that contains the root
- Does not require knowledge of the approximate area of root

Key Weakness:

- Unlike all the previous methods, Secant does not guarantee convergence



Secant's method of locating x_3 based on x_1 and x_2. Credit: Wikipedia

This method starts by checking two user-defined seeds, say we want to search for a root for `x² - math.pi=0` starting with `x_0=4` and `x_1=5`, then our seeds are 4 and 5. (note that this is the same as searching for `x` such that `x²=math.pi`)

We then locate the intercept with the target value `x_2` by drawing a straight line through `f(x_0)` and `f(x_1)` like what we have done in Regula-Falsi. If `f(x_2)` is not close enough to the target value, we repeat the step and locate `x_3`. In general, the next `x` can be calculated as:

$$x_{n+1} = x_n - f(x_n)\frac{x_n - x_{n-1}}{f(x_n) - f(x_{n-1})}$$

```python
def secant_algorithm(f, x_0, x_1, y, iterations, margin=.00_001):
    ''' Iterative approach of Root-finding with secant method
    Parameters
    ----------
    f: callable, continuous function
    x_0: float, initial seed
    x_1: float, initial seed
    y: float, target value
    iterations: int, maximum number of iterations to avoid indefinite divergence
    margin: float, margin of error in absolute term
    Returns
    -------
    A float x_2, where f(x_2) is within the margin of y
    '''

    assert x_0 != x_1, "Two different initial seeds are required."

    if abs((y_0 := f(x_0) - y)) < margin:
        # found!
        return x_0
    if abs((y_1 := f(x_1) - y)) < margin:
        # found!
        return x_1

    for i in range(iterations):
        x_2 = x_1 - y_1 * (x_1 - x_0) / (y_1 - y_0)
        if abs((y_2 := f(x_2) - y)) < margin:
            # found!
            return x_2
        x_0, x_1 = x_1, x_2
        y_0, y_1 = y_1, y_2
    return x_2
```

secant.py hosted with ❤ by GitHub view raw

5. Steffensen's Method

Secant's method further improves the Regula-Falsi algorithm by removing the requirement of a bracket which contains a root. Recall that the straight line is in fact just a naive estimate of the tangent line (i.e. the derivative) of the of two `x` values (or upper and lower bound in Regula-Falsi and Illinois algorithm). This estimate will be more accurate as the search converges. In Steffensen's algorithm, we will attempt to replace the straight line with a **better estimate of the derivative** for further improving the Secant's method.

Key Strength:

- Does not require a bracket that contains the root
- Does not require knowledge of the approximate area of root
- Faster convergence than Secant's method if possible

Key Weakness:

- Does not guarantee convergence if the initial seed is too far away from the actual root
- The continuous function will be evaluated twice that of Secant's method in order to better estimate the derivative

In order to estimate the derivative better, Steffenson's algorithm will compute the following based on the user-defined initial seed `x_0`.

$$g(x) = \frac{f(x + f(x))}{f(x)} - 1$$

which is equivalent to the following where $h = f(x)$:

$$g(x) = \frac{f(x + h) - f(x)}{h} - 1$$

Take a limit of h to 0 and you will get a derivative of f(x)

The generalised estimated slope function will then be used to locate the next step following the similar step as Secant's method:

$$x_{n+1} = x_n - \frac{f(x_n)}{x_n \times g(x_n)}$$

```
def steffensen_algorithm(f, x, y, iterations, margin=.00_001):
    ''' Iterative approach of Root-finding with steffensen's method
    Parameters
    ----------
    f: callable, continuous function
    x: float, initial seed
```

```
    y: float, target value
    iterations: int, maximum number of iterations to avoid indefinite divergence
    margin: float, margin of error in absolute term
    Returns
    -------
    A float x_2, where f(x_2) is within the margin of y
    '''

    assert x != 0, "Initial seed cannot be zero."

    if abs((y_x := f(x) - y)) < margin:
        # found!
        return x

    for i in range(iterations):
        g = (f(x + y_x) - y) / y_x - 1
        if g * x == 0:
            # Division by zero, search stops
            return x
        x -= (f(x) - y) / (g * x)
        if abs((y_x := f(x) - y)) < margin:
            # found!
            return x
    return x
```

steffensen.py hosted with ❤ by GitHub view raw

Conclusion

In the blog, we have gone through the strength, weakness, and the implementation of the following 5 root-finding algorithms:

1. Bisection algorithm
2. Regula-Falsi algorithm
3. Illinois algorithm
4. Secant's algorithm
5. Steffensen's algorithm

	Bisection	Regula-Falsi	Illinois	Secant	Steffensen
Approach	Bracket	Bracket	Bracket	Iterative	Iterative
Convergence	Guaranteed	Guaranteed	Guaranteed	Not Guaranteed	Not Guaranteed
Knowledge of Approximate Location of Root	Required	Required	Required	Not Required	Not Required
Number of Initial Seeds	2	2	2	2	1
Risk of Slow Convergence	N/A	Stagnant Bound	N/A	Initial Seed Not Close Enough to Root	Initial Seed Not Close Enough to Root
Method of Reducing Problem Space	Brute-force Halving	Approximate Derivate with Finite Difference	Approximate Derivate with Finite Difference	Approximate Derivate with Finite Difference	Approximate Derivative with Polynomial Fit
Speed of Convergence	Linear	Linear	Super-linear	Super-linear	Quadratic

Comparison of algorithms we have implemented

Once you get comfortable with these algorithms, don't stop there. There are actually more root-finding algorithms that we have not covered in this blog, for instance, Newton-Raphson's method, Inverse quadratic interpolation, Brent's method etc. Keep exploring and add these algorithms to your arsenal of tools.

Note that the snippets above are all written by myself, and have not been thoroughly tested. I would strongly recommend you to give it try, and use that as a starting point to modify to your own use case. Remember! Test it before you deploy it! Just that it work in my environment does not mean it would work in every setup!

Adios!

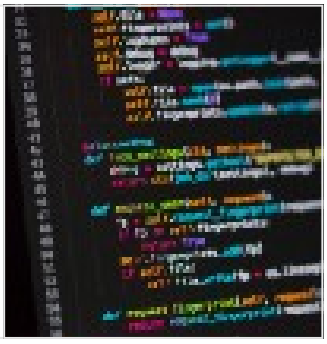
Before you go

You may also want to check these medium blogs out on how to improve your data science game:

Efficient Conditional Logic on Pandas DataFrames

Time to stop being too dependent on .iterrows() and .apply()

towardsdatascience.com



towardsdatascience.com

Let me know if you have learnt something new from this! And please also let me know if there are other neat tricks that I have missed!

Louis Chan - Director, Data Science - FTI Consulting | LinkedIn

Ambitious, curious and creative individual with a strong belief in inter-connectivity between branches knowledge and a...

www.linkedin.com

Data Science | Machine Learning | Python | Software Development | Programming

65

Enjoy the read? Reward the writer.[Beta]

Your tip will go to Louis Chan through a third-party platform of their choice, letting them know you appreciate their story.

Give a tip

Sign up for The Variable

By Towards Data Science

Every Thursday, the Variable delivers the very best of Towards Data Science: from hands-on tutorials and cutting-edge research to original features you don't want to miss. Take a look.

Get this newsletter