Welcome to Programming for Data Science

Welcome to the course manual for CSC310 at URI with Professor Brown.

This class meets MWF 3-3:50pm in Chafee Social Sci Center 235.

This website will contain the syllabus, class notes and other reference material for the class.

Course Calendar on BrightSpace

> **Tip**
>
> subscribe to that calendar in your favorite calendar application

Basic Facts

About this course

Data science exists at the intersection of computer science, statistics, and machine learning. That means writing programs to access and manipulate data so that it becomes available for analysis using statistical and machine learning techniques is at the core of data science. Data scientists use their data and analytical ability to find and interpret rich data sources; manage large amounts of data despite hardware, software, and bandwidth constraints; merge data sources; ensure consistency of datasets; create visualizations to aid in understanding data; build mathematical models using the data; and present and communicate the data insights/findings.

This course provides a survey of data science. Topics include data driven programming in Python; data sets, file formats and meta-data; descriptive statistics, data visualization, and foundations of predictive data modeling and machine learning; accessing web data and databases; distributed data management. You will work on weekly substantial programming problems such as accessing data in database and visualize it or build machine learning models of a given data set.

Basic programming skills (CSC201 or CSC211) are a prerequisite to this course. This course is a prerequisite course to machine learning, where you learn how machine learning algorithms work. In this course, we will start with a very fast review of basic programming ideas, since you've already done that before. We will learn how to *use* machine learning algorithms to do data science, but not how to *build* machine learning algorithms, we'll use packages that implement the algorithms for us.

About this syllabus

This syllabus is a *living* document and accessible from BrightSpace, as a pdf for download directly online at rhodyprog4ds.github.io/BrownFall21/syllabus. If you choose to download a copy of it, note that it is only a copy. You can get notification of changes from GitHub by "watching" the You can view the date of changes and exactly what changes were made on the Github commit history page.

Creating an issue is also a good way to ask questions about anything in the course it will prompt additions and expand the FAQ section.

About your instructor

Name: Dr. Sarah M Brown Office hours: TBA via zoom, link on BrightSpace

Dr. Sarah M Brown is a second year Assistant Professor of Computer Science, who does research on how social context changes machine learning. Dr. Brown earned a PhD in Electrical Engineering from Northeastern University, completed a postdoctoral fellowship at University of California Berkeley, and worked as a postdoctoral research associate at Brown University before joining URI. At Brown University, Dr. Brown taught the Data and Society course for the Master's in Data Science Program. You can learn more about me at my website or my research on my lab site.

You can call me Professor Brown or Dr. Brown, I use she/her pronouns.

The best way to contact me is e-mail or an issue on an assignment repo. For more details, see the Communication Section

Tools and Resources

We will use a variety of tools to conduct class and to facilitate your programming. You will need a computer with Linux, MacOS, or Windows. It is unlikely that a tablet will be able to do all of the things required in this course. A Chromebook may work, especially with developer tools turned on. Ask Dr. Brown if you need help getting access to an adequate computer.

All of the tools and resources below are either:

- paid for by URI **OR**
- freely available online.

Important

TL;DR [1]

- check Brightspace
- Log in to Prismia Chat
- Make a GitHub Account
- Install Python
- Install Git

Note

Seeing the BrightSpace site requires loging in with your URI SSO and being enrolled in the course

BrightSpace

This will be the central location from which you can access all other materials. Any links that are for private discussion among those enrolled in the course will be available only from our course Brightspace site.

This is also where your grades will appear and how I will post announcements.

For announcements, you can customize how you receive them.

Prismia chat

Our class link for Prismia chat is available on Brightspace. We will use this for chatting and in-class understanding checks.

On Prismia, all students see the instructor’s messages, but only the Instructor and TA see student responses.

Course Manual

The course manual will have content including the class policies, scheduling, class notes, assignment information, and additional resources. This will be linked from Brightspace and available publicly online at rhodyprog4ds.github.io/BrownFall21/. Links to the course reference text and code documentation will also be included here in the assignments and class notes.

GitHub Classroom

You will need a GitHub Account. If you do not already have one, please create one by the first day of class. If you have one, but have not used it recently, you may need to update your password and login credentials as the Authentication rules changed over the summer. In order to use the command line with https, you will need to create a Personal Access Token for each device you use. In order to use the command line with SSH, set up your public key.

Programming Environment

This a programming course, so you will need a programming environment. In order to complete assignments you need the items listed in the requirements list. The easiest way to meet these requirements is to follow the recommendations below. I will provide instruction assuming that you have followed the recommendations.

Requirements:

- Python with scientific computing packages (numpy, scipy, jupyter, pandas, seaborn, sklearn)
- Git
- A web browser compatible with Jupyter Notebooks

Warning

Everything in this class will be tested with the up to date (or otherwise specified) version of Jupyter Notebooks. Google Colab is similar, but not the same, and some things may not work there. It is an okay backup, but should not be your primary work environment.

Note

all Git instructions will be given as instructions for the command line interface and GitHub specific instructions via the web interface. You may choose to use GitHub desktop or built in IDE tools, but the instructional team may not be able to help.

Recommendation:

- Install python via Anaconda
- if you use Windows, install Git with GitBash (video instructions).
- if you use MacOS, install Git with the Xcode Command Line Tools. On Mavericks (10.9) or above you can do this by trying to run git from the Terminal the very first time.`git --version`

Optional:

- Text Editor: you may want a text editor outside of the Jupyter environment. Jupyter can edit markdown files (that you'll need for your portfolio), in browser, but it is more common to use a text editor like Atom or Sublime for this purpose.

Video install instructions for Anaconda:

- Windows
- Mac
- I don't have a video for linux, but it's a little more straight forward.

A tip from Dr. Brown

I use atom, but I decided to use it by downloading both Atom and Sublime and trying different things in each for a week. I liked Atom better after that and I've stuck with it since. I used Atom to write all of the content in this syllabus. VScode will also work, if needed

Textbook

The text for this class is a reference book and will not be a source of assignments. It will be a helpful reference and you may be directed there for answers to questions or alternate explanations ot topics.

Python for Data Science is available free online:

Zoom (backup only, Fall 2021 is in person)

This is where we will meet if for any reason we cannot be in person. You will find the link to class zoom sessions on Brightspace.

URI provides all faculty, staff, and students with a paid Zoom account. It *can* run in your browser or on a mobile device, but you will be able to participate in class best if you download the Zoom client on your computer. Please log in and configure your account. Please add a photo of yourself to your account so that we can still see your likeness in some form when your camera is off. You may also wish to use a virtual background and you are welcome to do so.

Class will be interactive, so if you cannot be in a quiet place at class time, headphones with a built in microphone are strongly recommended.

For help, you can access the instructions provided by IT.

[1] Too long; didn't read.

Data Science Achievements

In this course there are 5 learning outcomes that I expect you to achieve by the end of the semester. To get there, you'll focus on 15 smaller achievements that will be the basis of your grade. This section will describe how the topics covered, the learning outcomes, and the achievements are covered over time. In the next section, you'll see how these achievements turn into grades.

Learning Outcomes

By the end of the semester

1. (process) Describe the process of data science, define each phase, and identify standard tools
2. (data) Access and combine data in multiple formats for analysis
3. (exploratory) Perform exploratory data analyses including descriptive statistics and visualization
4. (modeling) Select models for data by applying and evaluating mutiple models to a single dataset
5. (communicate) Communicate solutions to problems with data in common industry formats

We will build your skill in the `process` and `communicate` outcomes over the whole semester. The middle three skills will correspond roughly to the content taught for each of the first three portfolio checks.

Schedule

The course will meet MWF 3-3:50pm in Chafee Social Sci Center 235. Every class will include participatory live coding (instructor types code while explaining, students follow along)) instruction and small exercises for you to progress toward level 1 achievements of the new skills introduced in class that day.

Note

On the Course Calendar on BrightSpace page you can get a feed link to add to the calendar of your choice by clicking on the subscribe (star) button on the top right of the page. Class is for 1 hour there because of Brightspace/zoom integration limitations, but that calendar includes the zoom link.

Programming assignments that will be due each week Tuesday by 11:59pm.

	topics	**skills**
week		
1	[admin, python review]	process
2	Loading data, Python review	[access, prepare, summarize]
3	Exploratory Data Analysis	[summarize, visualize]
4	Data Cleaning	[prepare, summarize, visualize]
5	Databases, Merging DataFrames	[access, construct, summarize]
6	Modeling, Naive Bayes, classification performance metrics	[classification, evaluate]
7	decision trees, cross validation	[classification, evaluate]
8	Regression	[regression, evaluate]
9	Clustering	[clustering, evaluate]
10	SVM, parameter tuning	[optimize, tools]
11	KNN, Model comparison	[compare, tools]
12	Text Analysis	[unstructured]
13	Images Analysis	[unstructured, tools]
14	Deep Learning	[tools, compare]

Achievement Definitions

The table below describes how your participation, assignments, and portfolios will be assessed to earn each achievement. The keyword for each skill is a short name that will be used to refer to skills throughout the course materials; the full description of the skill is in this table.

	skill	Level 1	Level 2	Level 3
keyword				
python	pythonic code writing	python code that mostly runs, occasional pep8 adherance	python code that reliably runs, frequent pep8 adherance	reliable, efficient, pythonic code that consistently adheres to pep8
process	describe data science as a process	Identify basic components of data science	Describe and define each stage of the data science process	Compare different ways that data science can facilitate decision making
access	access data in multiple formats	load data from at least one format; identify the most common data formats	Load data for processing from the most common formats; Compare and constrast most common formats	access data from both common and uncommon formats and identify best practices for formats in different contexts
construct	construct datasets from multiple sources	identify what should happen to merge datasets or when they can be merged	apply basic merges	merge data that is not automatically aligned
summarize	Summarize and describe data	Describe the shape and structure of a dataset in basic terms	compute summary statndard statistics of a whole dataset and grouped data	Compute and interpret various summary statistics of subsets of data
visualize	Visualize data	identify plot types, generate basic plots from pandas	generate multiple plot types with complete labeling with pandas and seaborn	generate complex plots with pandas and plotting libraries and customize with matplotlib or additional parameters
prepare	prepare data for analysis	identify if data is or is not ready for analysis, potential problems with data	apply data reshaping, cleaning, and filtering as directed	apply data reshaping, cleaning, and filtering manipulations reliably and correctly by assessing data as received
classification	Apply classification	identify and describe what classification is, apply pre-fit classification models	fit preselected classification model to a dataset	fit and apply classification models and select appropriate classification models for different contexts
regression	Apply Regression	identify what data that can be used for regression looks like	can fit linear regression models	can fit and explain regrularized or nonlinear regression
clustering	Clustering	describe what clustering is	apply basic clustering	apply multiple clustering techniques, and interpret results
evaluate	Evaluate model performance	Explain basic performance metrics for different data science tasks	Apply basic model evaluation metrics to a held out test set	Evaluate a model with multiple metrics and cross validation
optimize	Optimize model parameters	Identify when model parameters need to be optimized	Manually optimize basic model parameters such as model order	Select optimal parameters based of mutiple quanttiateve criteria and automate parameter tuning
compare	compare models	Qualitatively compare model classes	Compare model classes in specific terms and fit models in terms of traditional model performance metrics	Evaluate tradeoffs between different model comparison types

	skill	Level 1	Level 2	Level 3
keyword				
unstructured	model unstructured data	Identify options for representing text data and use them once data is tranformed	Apply at least one representation to transform unstructured data for model fitting or summarizing	apply multiple representations and compare and contrast them for different end results
workflow	use industry standard data science tools and workflows to solve data science problems	Solve well strucutred problems with a single tool pipeline	Solve semi-strucutred, completely specified problems, apply common structure to learn new features of standard tools	Scope, choose an appropriate tool pipeline and solve data science problems, describe strengths and weakensses of common tools

Assignments and Skills

Using the keywords from the table above, this table shows which assignments you will be able to demonstrate which skills and the total number of assignments that assess each skill. This is the number of opportunities you have to earn Level 2 and still preserve 2 chances to earn Level 3 for each skill.

	A1	A2	A3	A4	A5	A6	A7	A8	A9	A10	A11	A12	A13	Assignment
keyword														
python	1	1	1	1	0	0	0	0	0	0	0	0	0	
process	1	1	0	0	0	0	0	0	0	0	0	0	0	
access	0	1	1	1	0	0	0	0	0	0	0	0	0	
construct	0	0	0	0	1	1	0	0	0	0	0	0	0	
summarize	0	0	1	1	1	1	1	1	1	1	1	1	1	1
visualize	0	0	1	1	0	1	1	1	1	1	1	1	1	1
prepare	0	0	0	1	1	0	0	0	0	0	0	0	0	
classification	0	0	0	0	0	1	1	0	0	1	0	0	0	
regression	0	0	0	0	0	0	0	1	0	0	1	0	0	
clustering	0	0	0	0	0	0	0	0	1	0	1	0	0	
evaluate	0	0	0	0	0	0	0	0	0	1	1	0	0	
optimize	0	0	0	0	0	0	0	0	0	1	1	0	0	
compare	0	0	0	0	0	0	0	0	0	0	1	0	1	
unstructured	0	0	0	0	0	0	0	0	0	0	0	1	1	
workflow	0	0	0	0	0	0	0	0	0	1	1	1	1	

Portfolios and Skills

The objective of your portfolio submissions is to earn Level 3 achievements. The following table shows what Level 3 looks like for each skill and identifies which portfolio submissions you can earn that Level 3 in that skill.

	Level 3	P1	P2	P3	P4
keyword					
python	reliable, efficient, pythonic code that consistently adheres to pep8	1	1	0	0
process	Compare different ways that data science can facilitate decision making	0	1	1	0
access	access data from both common and uncommon formats and identify best practices for formats in different contexts	1	1	0	0
construct	merge data that is not automatically aligned	1	1	0	0
summarize	Compute and interpret various summary statistics of subsets of data	1	1	0	0
visualize	generate complex plots with pandas and plotting libraries and customize with matplotlib or additional parameters	1	1	0	0
prepare	apply data reshaping, cleaning, and filtering manipulations reliably and correctly by assessing data as received	1	1	0	0
classification	fit and apply classification models and select appropriate classification models for different contexts	0	1	1	0
regression	can fit and explain regrularized or nonlinear regression	0	1	1	0
clustering	apply multiple clustering techniques, and interpret results	0	1	1	0
evaluate	Evaluate a model with multiple metrics and cross validation	0	1	1	0
optimize	Select optimal parameters based of mutiple quanttiateve criteria and automate parameter tuning	0	0	1	1
compare	Evaluate tradeoffs between different model comparison types	0	0	1	1
unstructured	apply multiple representations and compare and contrast them for different end results	0	0	1	1
workflow	Scope, choose an appropriate tool pipeline and solve data science problems, describe strengths and weakensses of common tools	0	0	1	1

Grading

This section of the syllabus describes the principles and mechanics of the grading for the course. This course will be graded on a basis of a set of *skills* (described in detail the next section of the syllabus). This is in contrast to more common grading on a basis of points earned through assignments.

Principles of Grading

Learning happens through practice and feedback. My goal as a teacher is for you to learn. The grading in this course is based on your learning of the material, rather than your completion of the activities that are assigned.

This course is designed to encourage you to work steadily at learning the material and demonstrating your new knowledge. There are no single points of failure, where you lose points that cannot be recovered. Also, you cannot cram anything one time and then forget it. The material will build and you have to demonstrate that you retained things.

- Earning a C in this class means you have a general understanding of Data Science and could participate in a basic conversation about all of the topics we cover. I expect everyone to reach this level.
- Earning a B means that you could solve simple data science problems on your own and complete parts of more complex problems as instructed by, for example, a supervisor in an internship or entry level job. This is a very accessible goal, it does not require you to get anything on the first try or to explore topics on your own. I expect most students to reach this level.
- Earning an A means that you could solve moderately complex problems independently and discus the quality of others' data science solutions. This class will be challenging, it requires you to explore topics a little deeper than we cover them in class, but unlike typical grading it does not require all of your assignments to be near perfect.

Grading this way also is more amenable to the fact that there are correct and incorrect ways to do things, but there is not always a single correct answer to a realistic data science problem. Your work will be assessed on whether or not it demonstrates your learning of the targeted skills. You will also receive feedback on how to improve.

How it works

There are 15 skills that you will be graded on in this course. While learning these skills, you will work through a progression of learning. Your grade will be based on earning 45 achievements that are organized into 15 skill groups with 3 levels for each.

These map onto letter grades roughly as follows:

- If you achieve level 1 in all of the skills, you will earn at least a C in the course.
- To earn a B, you must earn all of the level 1 and level 2 achievements.
- To earn an A, you must earn all of the achievements.

You will have at least three opportunities to earn every level 2 achievement. You will have at least two opportunities to earn every level 3 achievement. You will have three *types* of opportunities to demonstrate your current skill level: participation, assignments, and a portfolio.

Each level of achievement corresponds to a phase in your learning of the skill:

- To earn level 1 achievements, you will need to demonstrate basic awareness of the required concepts and know approximately what to do, but you may need specific instructions of which things to do or to look up examples to modify every step of the way. You can earn level 1 achievements in class, assignments, or portfolio submissions.
- To earn level 2 achievements you will need to demonstrate understanding of the concepts and the ability to apply them with instruction after earning the level 1 achievement for that skill. You can earn level 2 achievements in assignments or portfolio submissions.
- To earn level 3 achievements you will be required to consistently execute each skill and demonstrate deep understanding of the course material, after achieving level 2 in that skill. You can earn level 3 achievements only through your portfolio submissions.

For each skill these are defined in the Achievement Definition Table

Participation

While attending synchronous class sessions, there will be understanding checks and in class exercises. Completing in class exercises and correctly answering questions in class can earn level 1 achievements. In class questions will be administered through the classroom chat platform Prismia.chat; these records will be used to update your skill progression. You can also earn level 1 achievements from adding annotation to a section of the class notes.

Assignments

For your learning to progress and earn level 2 achievements, you must practice with the skills outside of class time.

Assignments will each evaluate certain skills. After your assignment is reviewed, you will get qualitative feedback on your work, and an assessment of your demonstration of the targeted skills.

Warning

If you will skip an assignment, please accept the GitHub assignment and then close the Feedback pull request with a comment. This way we can make sure that you have support you need.

Portfolio Checks

To earn level 3 achievements, you will build a portfolio consisting of reflections, challenge problems, and longer analyses over the course of the semester. You will submit your portfolio for review 4 times. The first two will cover the skills taught up until 1 week before the submission deadline.

The third and fourth portfolio checks will cover all of the skills. The fourth will be due during finals. This means that, if you have achieved mastery of all of the skills by the 3rd portfolio check, you do not need to submit the fourth one.

Portfolio prompts will be given throughout the class, some will be strucutred questions, others may be questions that arise in class, for which there is not time to answer.

TLDR

You *could* earn a C through in class participation alone, if you make nearly zero mistakes. To earn a B, you must complete assignments and participate in class. To earn an A you must participate, complete assignments, and build a portfolio.

Detailed mechanics

On Brightspace there are 45 Grade items that you will get a 0 or a 1 grade for. These will be revealed, so that you can view them as you have an opportunity to demonstrate each one. The table below shows the minimum number of skills at each level to earn each letter grade.

	Level 3	Level 2	Level 1
letter grade			
A	15	15	15
A-	10	15	15
B+	5	15	15
B	0	15	15
B-	0	10	15
C+	0	5	15
C	0	0	15
C-	0	0	10
D+	0	0	5
D	0	0	3

For example, if you achieve level 2 on all of the skills and level 3 on 7 skills, that will be a B+.

If you achieve level 3 on 14 of the skills, but only level 1 on one of the skills, that will be a B-, because the minimum number of level 2 achievements for a B is 15. In this scenario the total number of achievements is 14 at level 3, 14 at level 2 and 15 at level 3, because you have to earn achievements within a skill in sequence.

ℹ **Note**

In this example, you will have also achieved level 1 on all of the skills, because it is a prerequisite to level 2.

The letter grade can be computed as follows

```
def compute_grade(num_level1,num_level2,num_level3):
    '''
    Computes a grade for CSC/DSP310 from numbers of achievements at each level

    Parameters:
    ------------
    num_level1 : int
      number of level 1 achievements earned
    num_level2 : int
      number of level 2 achievements earned
    num_level3 : int
      number of level 3 achievements earned

    Returns:
    --------
    letter_grade : string
      letter grade with modifier (+/-)
    '''
    if num_level1 == 15:
        if num_level2 == 15:
            if num_level3 == 15:
                grade = 'A'
            elif num_level3 >= 10:
                grade = 'A-'
            elif num_level3 >=5:
                grade = 'B+'
            else:
                grade = 'B'
        elif num_level2 >=10:
            grade = 'B-'
        elif num_level2 >=5:
            grade = 'C+'
        else:
            grade = 'C'
    elif num_level1 >= 10:
        grade = 'C-'
    elif num_level1 >= 5:
        grade = 'D+'
    elif num_level1 >=3:
        grade = 'D'
    else:
        grade = 'F'


    return grade
```

For example you can run the code like this in a cell to see the output

```
compute_grade(15,15,15)
```

```
'A'
```

```
compute_grade(14,14,14)
```

```
'C-'
```

Or use `assert` to test it formally

```
assert compute_grade(14,14,14) == 'C-'
```

```
assert compute_grade(15,15,15) == 'A'
```

```
assert compute_grade(15,15,11) == 'A-'
```

Late work

Late assignments will not be graded. Every skill will be assessed through more than one assignment, so missing assignments occasionally not necessarily hurt your grade. If you do not submit any assignments that cover a given skill, you may earn the level 2 achievement in that skill through a portfolio check, but you will not be able to earn the level 3 achievement in that skill. If you submit work that is not complete, however, it will be assessed and receive feedback. Submitting pseudocode or code with errors and comments about what you have tried could earn a level 1 achievement. Additionally, most assignments cover multiple skills, so partially completing the assignment may earn level 2 for one, but not all. Submitting *something* even if it is not perfect is important to keeping conversation open and getting feedback and help continuously.

> **Note**
>
> You may visit office hours to discuss assignments that you did not complete on time to get feedback and check your own understanding, but they will not count toward skill demonstration.

Building your Data Science Portfolio should be an ongoing process, where you commit work to your portfolio frequently. If something comes up and you cannot finish all that you would like assessed by the deadline, open an `Extension Request` issue on your repository.

In this issue, include:

1. A new deadline proposal
2. What additional work you plan to add
3. Why the extension is important to your learning
4. Why the extension will not hinder your ability to complete the next assignment on time.

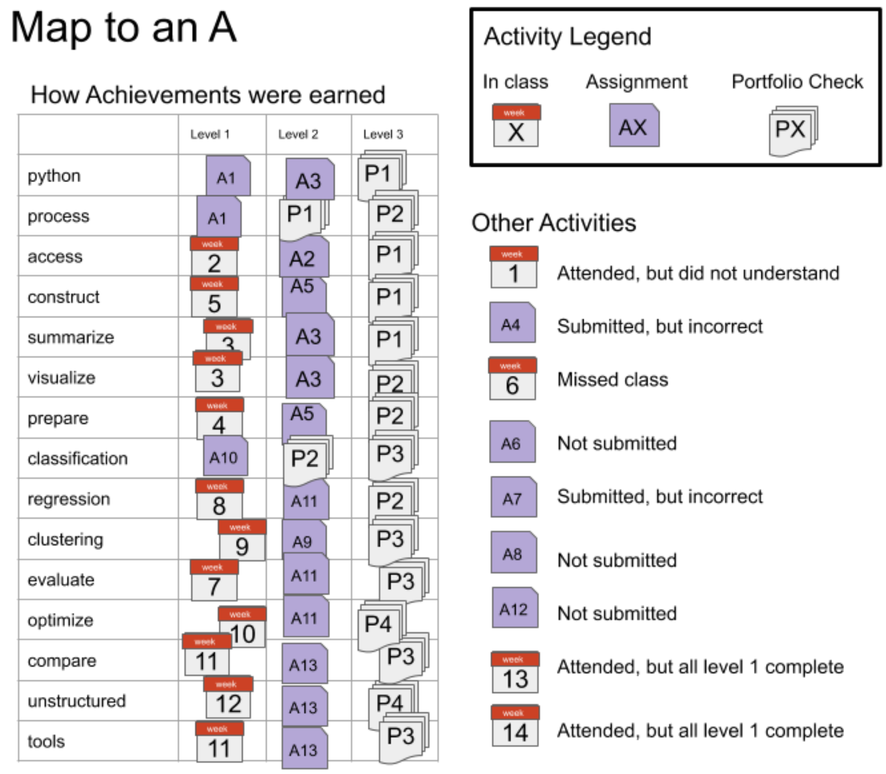
This request should be no more than 7 sentences.

Portfolio due dates will be announced well in advance and prompts for it will be released weekly. You should spend some time working on it each week, applying what you've learned so far, from the feedback on previous assignments.

Examples

If you always attend and get everything correct, you will earn and A and you won't need to submit the 4th portfolio check or assignment 13.

Getting A Without Perfection



How Achievements were earned

	Level 1	Level 2	Level 3
python	A1	A3	P1
process	A1	P1	P2
access	week 2	A2	P1
construct	week 5	A5	P1
summarize	week 3	A3	P1
visualize	week 3	A3	P2
prepare	week 4	A5	P2
classification	A10	P2	P3
regression	week 8	A11	P2
clustering	week 9	A9	P3
evaluate	week 7	A11	P3
optimize	week 10	A11	P4
compare	week 11	A13	P3
unstructured	week 12	A13	P4
tools	week 11	A13	P3

Activity Legend

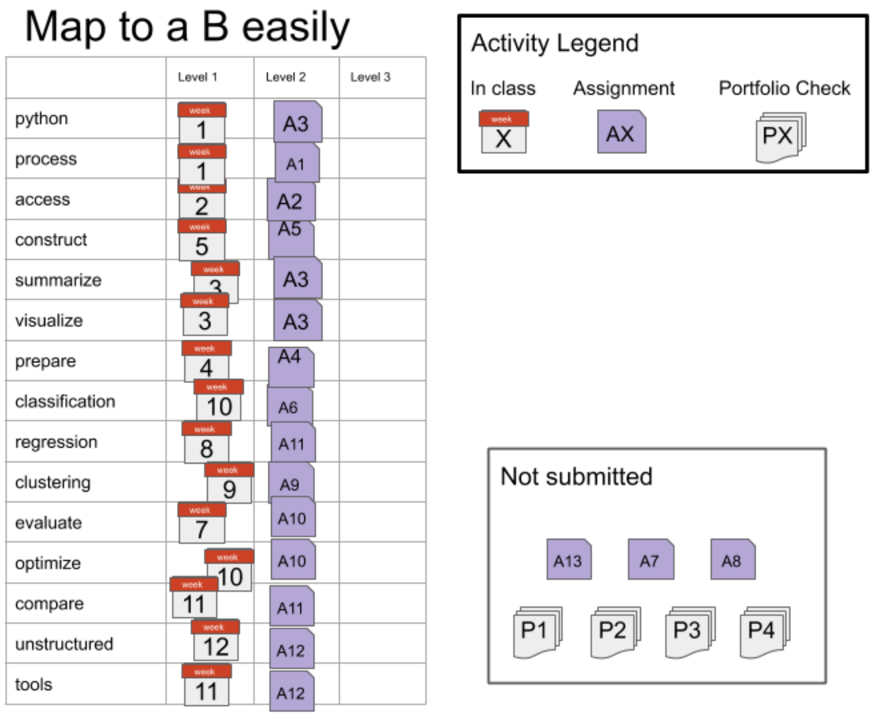
In class	Assignment	Portfolio Check
week X	AX	PX

Other Activities

week 1	Attended, but did not understand
A4	Submitted, but incorrect
week 6	Missed class
A6	Not submitted
A7	Submitted, but incorrect
A8	Not submitted
A12	Not submitted
week 13	Attended, but all level 1 complete
week 14	Attended, but all level 1 complete

In this example the student made several mistakes, but still earned an A. This is the advantage to this grading scheme. For the `python`, `process`, and `classification` skills, the level 1 achievements were earned on assignments, not in class. For the `process` and `classification` skills, the level 2 achievements were not earned on assignments, only on portfolio checks, but they were earned on the first portfolio of those skills, so the level 3 achievements were earned on the second portfolio check for that skill. This student's fourth portfolio only demonstrated two skills: `optimize` and `unstructured`. It included only 1 analysis, a text analysis with optimizing the parameters of the model. Assignments 4 and 7 were both submitted, but didn't earn any achievements, the student got feedback though, that they were able to apply in later assignments to earn the achievements. The student missed class week 6 and chose to not submit assignment 6 and use week 7 to catch up. The student had too much work in another class and chose to skip assignment 8. The student tried assignment 12, but didn't finish it on time, so it was not graded, but the student visited office hours to understand and be sure to earn the level 2 `unstructured` achievement on assignment 13.

Getting a B with minimal work


Map to a B easily

	Level 1	Level 2	Level 3
python	week 1	A3	
process	week 1	A1	
access	week 2	A2	
construct	week 5	A5	
summarize	week 3	A3	
visualize	week 3	A3	
prepare	week 4	A4	
classification	week 10	A6	
regression	week 8	A11	
clustering	week 9	A9	
evaluate	week 7	A10	
optimize	week 10	A10	
compare	week 11	A11	
unstructured	week 12	A12	
tools	week 11	A12	

Activity Legend

In class	Assignment	Portfolio Check
week X	AX	PX

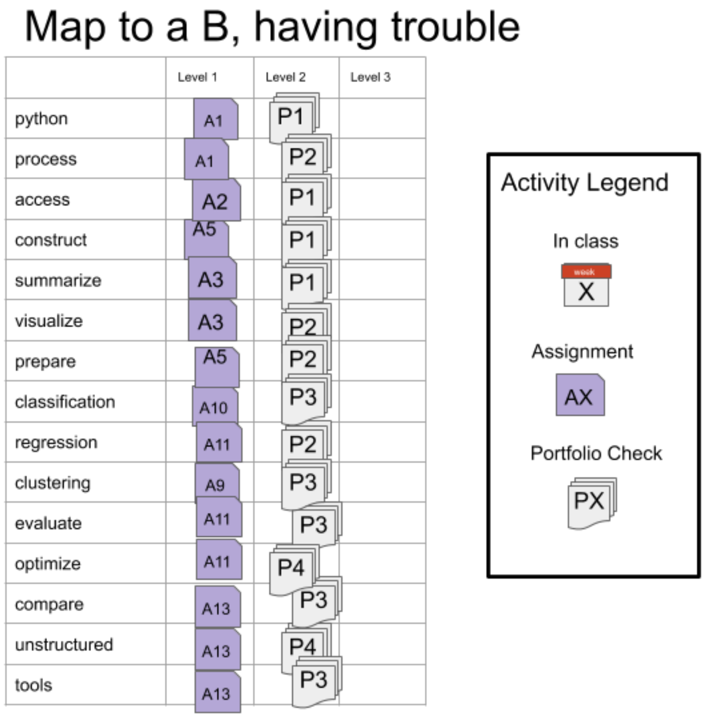
Not submitted

A13 A7 A8

P1 P2 P3 P4

In this example, the student earned all level 1 achievements in class and all level 2 on assignments. This student was content with getting a B and chose to not submit a portfolio.

Getting a B while having trouble


Map to a B, having trouble

	Level 1	Level 2	Level 3
python	A1	P1	
process	A1	P2	
access	A2	P1	
construct	A5	P1	
summarize	A3	P1	
visualize	A3	P2	
prepare	A5	P2	
classification	A10	P3	
regression	A11	P2	
clustering	A9	P3	
evaluate	A11	P3	
optimize	A11	P4	
compare	A13	P3	
unstructured	A13	P4	
tools	A13	P3	

Activity Legend

In class: week X

Assignment: AX

Portfolio Check: PX

In this example, the student struggled to understand in class and on assignments. Assignments were submitted that showed some understanding, but all had some serious mistakes, so only level 1 achievements were earned from assignments. The student wanted to get a B and worked hard to get the level 2 achievements on the portfolio checks.

Ram Tokens

Ram Tokens in this course will be used as a currency for extra effort. You can earn Ram Tokens by doing work that supports your learning or class activities, but do not directly demonstrate achievements. You can spend Ram Tokens to get extra grading. This will be mostly applicable to Portfolio Checks. In Checks 3 & 4, some achievements will not be eligible for grading as per the table. However, you can exchange Ram Tokens to make more achievements eligible for assessment. This system rewards you for putting in consistent effort, even if it takes you many tries to understand a concept.

To accumulate Ram Tokens, you submit a ‘Deposit’ to the Ram Token Bank: http://drsmb.co/ramtoken with a link to what you did to earn a token. To apply Ram tokens for extra grading, submit the same form, with a link to the assignment and add the

Support

Academic Enhancement Center

Academic Enhancement Center (for undergraduate courses): Located in Roosevelt Hall, the AEC offers free face-to-face and web-based services to undergraduate students seeking academic support. Peer tutoring is available for STEM-related courses by appointment online and in-person. The Writing Center offers peer tutoring focused on supporting undergraduate writers at any stage of a writing assignment. The UCS160 course and academic skills consultations offer students strategies and activities aimed at improving their studying and test-taking skills. Complete details about each of these programs, up-to-date schedules, contact information and self-service study resources are all available on the AEC website.

- **STEM Tutoring** helps students navigate 100 and 200 level math, chemistry, physics, biology, and other select STEM courses. The STEM Tutoring program offers free online and limited in-person peer-tutoring this fall. Undergraduates in introductory STEM courses have a variety of small group times to choose from and can select occasional or weekly appointments. Appointments and locations will be visible in the TutorTrac system on September 14th, 2020. The TutorTrac application is available through URI Microsoft 365 single sign-on and by visiting aec.uri.edu. More detailed information and instructions can be found on the AEC tutoring page.
- **Academic Skills Development** resources helps students plan work, manage time, and study more effectively. In Fall 2020, all Academic Skills and Strategies programming are offered both online and in-person. UCS160: Success in Higher Education is a one-credit course on developing a more effective approach to studying. Academic Consultations are 30-minute, 1 to 1 appointments that students can schedule on Starfish with Dr. David Hayes to address individual academic issues. Study Your Way to Success is a self-guided web portal connecting students to tips and strategies on studying and time management related topics. For more information on these programs, visit the Academic Skills Page or contact Dr. Hayes directly at davidhayes@uri.edu.
- The **Undergraduate Writing Center** provides free writing support to students in any class, at any stage of the writing process: from understanding an assignment and brainstorming ideas, to developing, organizing, and revising a draft. Fall 2020 services are offered through two online options: 1) real-time synchronous appointments with a peer consultant (25- and 50-minute slots, available Sunday - Friday), and 2) written asynchronous consultations with a 24-hour turn-around response time (available Monday - Friday). Synchronous appointments are video-based, with audio, chat, document-sharing, and live captioning capabilities, to meet a range of accessibility needs. View the synchronous and asynchronous schedules and book online, visit uri.mywconline.com.

Policies

Anti-Bias Statement:

We respect the rights and dignity of each individual and group. We reject prejudice and intolerance, and we work to understand differences. We believe that equity and inclusion are critical components for campus community members to thrive. If you are a target or a witness of a bias incident, you are encouraged to submit a report to the URI Bias Response Team at www.uri.edu/brt. There you will also find people and resources to help.

Disability Services for Students Statement:

Your access in this course is important. Please send me your Disability Services for Students (DSS) accommodation letter early in the semester so that we have adequate time to discuss and arrange your approved academic accommodations. If you have not yet established services through DSS, please contact them to engage in a confidential conversation about the process for requesting reasonable accommodations in the classroom. DSS can be reached by calling: 401-874-2098, visiting: web.uri.edu/disability, or emailing: dss@etal.uri.edu. We are available to meet with students enrolled in Kingston as well as Providence courses.

Academic Honesty

Students are expected to be honest in all academic work. A student's name on any written work, quiz or exam shall be regarded as assurance that the work is the result of the student's own independent thought and study. Work should be stated in the student's own words, properly attributed to its source. Students have an obligation to know how to quote, paraphrase, summarize, cite and reference the work of others with integrity. The following are examples of academic dishonesty.

- Using material, directly or paraphrasing, from published sources (print or electronic) without appropriate citation
- Claiming disproportionate credit for work not done independently
- Unauthorized possession or access to exams
- Unauthorized communication during exams
- Unauthorized use of another's work or preparing work for another student
- Taking an exam for another student
- Altering or attempting to alter grades
- The use of notes or electronic devices to gain an unauthorized advantage during exams
- Fabricating or falsifying facts, data or references
- Facilitating or aiding another's academic dishonesty
- Submitting the same paper for more than one course without prior approval from the instructors

URI COVID-19 Statement

The University is committed to delivering its educational mission while protecting the health and safety of our community. While the university has worked to create a healthy learning environment for all, it is up to all of us to ensure our campus stays that way.

As members of the URI community, students are required to comply with standards of conduct and take precautions to keep themselves and others safe. Visit web.uri.edu/coronavirus/ for the latest information about the URI COVID-19 response.

- Universal indoor masking is required by all community members, on all campuses, regardless of vaccination status. If the universal mask mandate is discontinued during the semester, students who have an approved exemption and are not fully vaccinated will need to continue to wear a mask indoors and maintain physical distance.
- Students who are experiencing symptoms of illness should not come to class. Please stay in your home/room and notify URI Health Services via phone at 401-874-2246.
- If you are already on campus and start to feel ill, go home/back to your room and self-isolate. Notify URI Health Services via phone immediately at 401-874-2246.

If you are unable to attend class, please notify me at brownsarahm@uri.edu. We will work together to ensure that course instruction and work is completed for the semester.

Course Communications

Help Hours

Day	Time	Location	Host
Monday	1:00:00 PM-2:30	inperson roomtbd	Chamundi
Wednesday	4:00:00 PM	inperson roomtbd	Chamundi
Wedneday	2:00:00 PM-3	inperson roomtbd	Chamundi
Wednesday	7:00:00 PM-8:30	gather.town	Sarah
Friday	5:00:00 PM-6:30pm	gather.town	Chamundi
By appointment	TBD	in person Tyler 134	Sarah

We have several different ways to communicate in this course. This section summarizes them

To reach out, By usage

usage	platform	area	note
in class	prismia	chat	outside of class time this is not monitored closely
any time	prismia	message board	for discussion with peers
any time	prismia	download transcript	use after class to get preliminary notes eg if you miss a class
private questions to your assignment	github	issue on assignment repo	eg bugs in your code"
for general questions that can help others	github	issue on course website	eg what the instructions of an assignment mean or questions about the syllabus
to share resources	github	pull request on website	remember to request ram tokens if applicable
matters that don't fit into another category	e-mail	to brownsarahm@uri.edu	remember to include `[CSC310]` or `[DSP310]` (note `verbatim` no space)

> ℹ **Note**
>
> e-mail is last because it's not collaborative; other platforms allow us (Proessor + TA) to collaborate on who responds to things more easily.

By Platform

Use e-mail for

usage	area	note
matters that don't fit into another category	to brownsarahm@uri.edu	remember to include `[CSC310]` or `[DSP310]` (note `verbatim` no space)

Use github for

usage	area	note
private questions to your assignment	issue on assignment repo	eg bugs in your code"
for general questions that can help others	issue on course website	eg what the instructions of an assignment mean or questions about the syllabus
to share resources	pull request on website	remember to request ram tokens if applicable

Use prismia for

usage	area	note
in class	chat	outside of class time this is not monitored closely
any time	message board	for discussion with peers
any time	download transcript	use after class to get preliminary notes eg if you miss a class

Tips

For assignment help

- **send in advance, leave time for a response** I check e-mail/github a small number of times per day, during work hours, almost exclusively. You might see me post to this site, post to BrightSpace, or comment on your assignments outside of my normal working hours, but I will not reliably see emails that arrive during those hours. This means that it is important to start assignments early.

Using issues

- use issues for content directly related to assignments. If you push your code to the repository and then open an issue, I can see your code and your question at the same time and download it to run it if I need to debug it
- use issues for questions about this syllabus or class notes. At the top right there's a GitHub logo that allows you to open a issue (for a question) or suggest an edit (eg if you think there's a typo or you find an additional helpful resource related to something)

For E-email

- use e-mail for general inquiries or notifications
- Please include `[CSC310]` or `[DSP310]` in the subject line of your email along with the topic of your message. This is important, because your messages are important, but I also get a lot of e-mail. Consider these a cheat code to my inbox: I have setup a filter that will flag your e-mail if you use one of those in the subject to ensure that I see it.

> **Note**
>
> Whether you use CSC or DSP does not matter.

1. Welcome to Programming to Data Science

Today's goals:

1. Operate tools for in-class participation
2. Understand what Data Science is, in broad terms
3. Understand the syllabus (grading, topics covered, schedule, etc)
4. Understand how to learn in this course

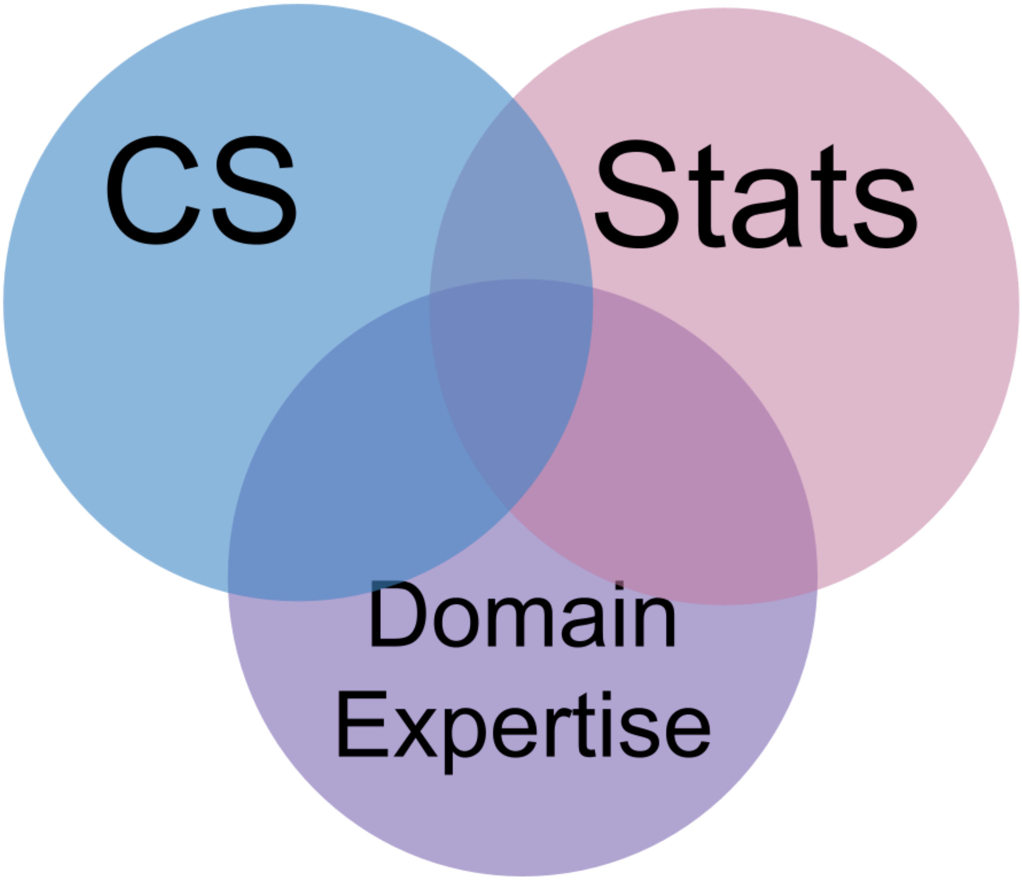
1.1. Prismia Chat

We will use these to monitor your participation in class and to gather information. Features:

- instructor only
- reply to you directly
- share responses for all

1.2. What is Data Science

In general:

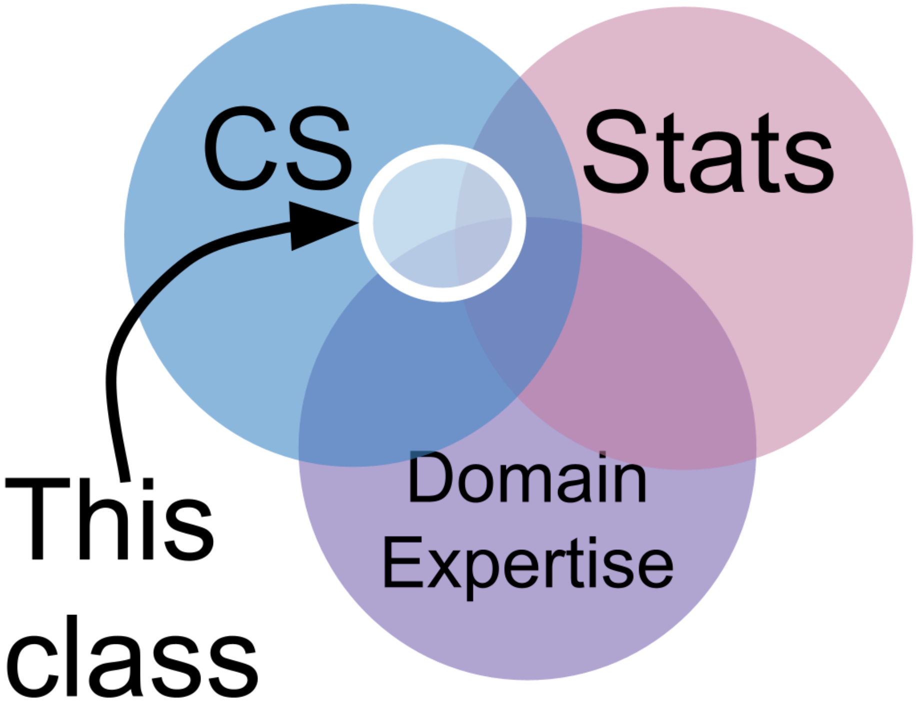


statistics is the type of math we use to make sense of data. Formally, a statistic is just a function of data.

computer science is so that we can manipulate visualize and automate the inferences we make.

domain expertise helps us have the intuition to know if what we did worked right. A statistic must be interpreted in context; the relevant context determines what they mean and which are valid. The context will say whether automating something is safe or not, it can help us tell whether our code actually worked right or not.

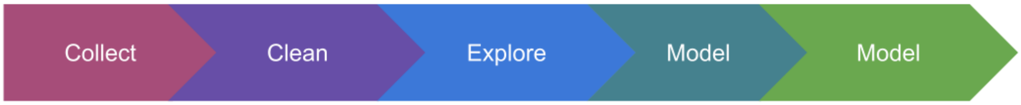
For this class



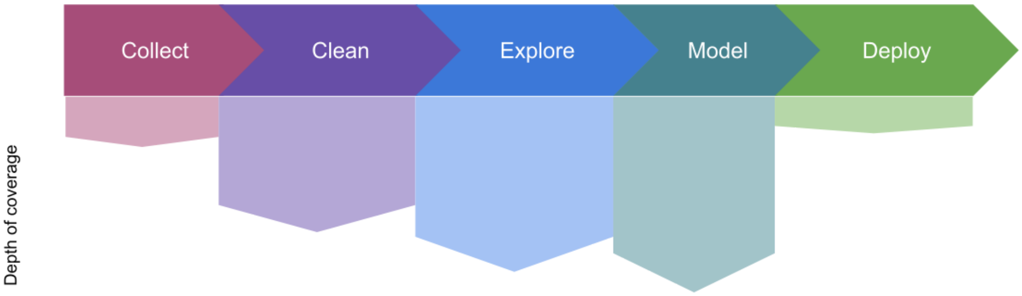
We'll focus on the programming as our main means of studying data science, but we will use bits of the other parts. In particular, you're encouraged to choose datasets that you have domain expertise about, or that you want to learn about.

But there are many definitions. We'll use this one, but you may come across others.

1.2.1. How does data science happen?

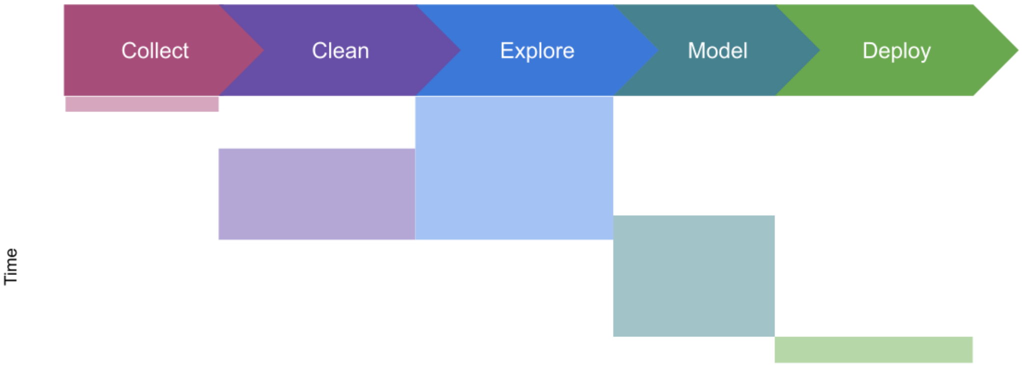


1.2.2. how we'll cover it, in depth



- *collect*: Discuss only a little; Minimal programming involved
- *clean*: Cover the main programming techniques; Some requires domain knowledge beyond scope of course
- *explore*: Cover the main programming techniques; Some requires domain knowledge beyond scope of course
- *model*:Cover the main programming, basic idea of models; How to use models, not how learning algorithms work
- *deploy*: A little bit at the end, but a lot of preparation for decision making around deployment

1.2.2.1. how we'll cover it in, time



We'll cover exploratory data analysis before cleaning because those tools will help us check how we've cleaneed the data.

1.3. How this class will work

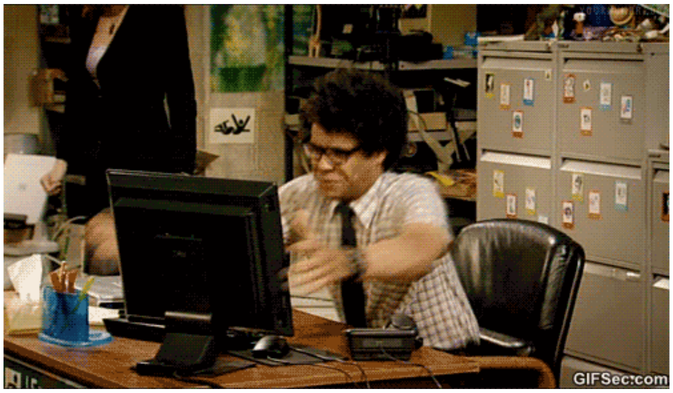
- today is an exception
- in general we'll be live coding

Let's look at the syllabus

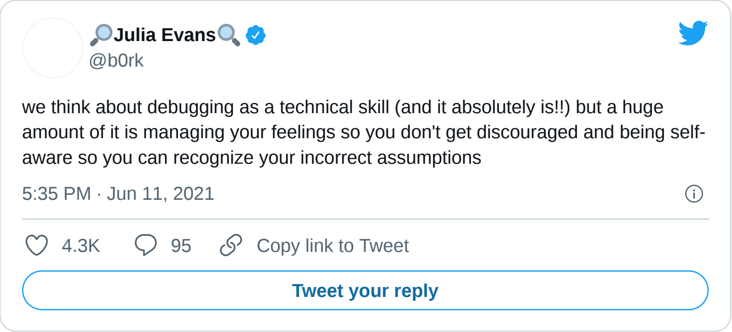
Read carefully to make sure you understand the grading; it’s not typical points and an average.

Class is designed to avoid this:

1.4.



1.5. Learning Cycle



🔎Julia Evans🔍
@b0rk

we think about debugging as a technical skill (and it absolutely is!!) but a huge amount of it is managing your feelings so you don't get discouraged and being self-aware so you can recognize your incorrect assumptions

5:35 PM · Jun 11, 2021

Read more about how I’m designing this course to help you learn on the how to learn page.

1.6. Check your understaning of the syllabus

It’s easy when reading something long to lose track of it. Your eyes can go over each word, without actually retaining the information, but it’s important to understand the syllabus for the course.

You can find the answers to the following questions on the syllabus. If you’ve already read it, try answering them to check your understanding. If you haven’t read it yet, use these to guide you to get familiar with finding key facts about the course on the syllabus.

1. What do you need to bring to class each day?
2. What is the basis of grading for this course?
3. How do you reference the course text?
4. What is the penalty for missing an assignment?

More information about the course is available throughout the site, the next few questions will help you self-check that you’ve found the important things. Remember, the goal is not necessarily to memorize all of this, but to be able to find it.

1. When & what are you expected to read for this class?

- [ ] read the text book before class
- [ ] review notes & documentation after class
- [ ] preview the notes & documentation before class
- [ ] read documentation and text book after class

1. Your assignment says to find a dataset that has variables of a specific type, which website can you use?
2. Your assignment says to find a dataset of any type about something you’re interested in, which resource would you use?

2. Jupyter Notebook Tour & Python Review

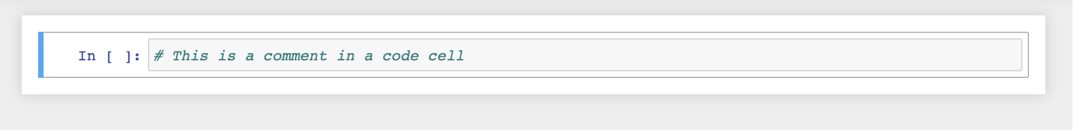
2.1. A jupyter notebook tour

Launch a `jupyter notebook`:

- on Windows, use anaconda terminal
- on Mac/Linux, use terminal

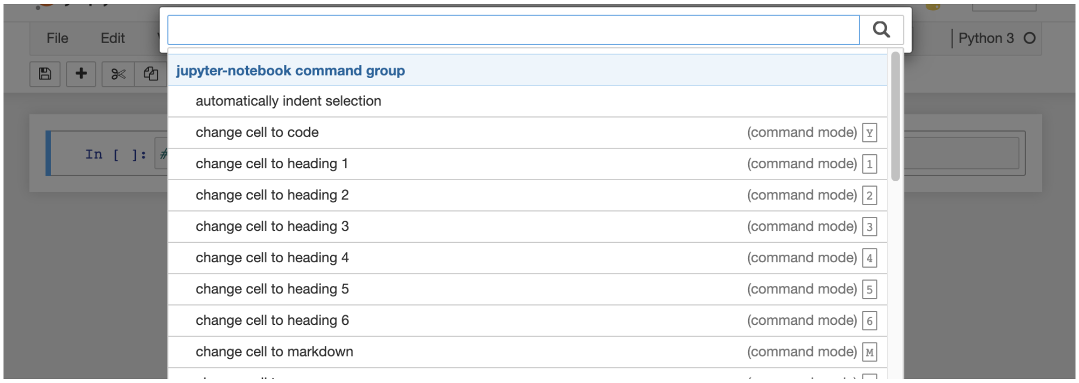
```
cd path/to/where/you/save/notes
jupyter notebook
```

A Jupyter notebook has two modes. When you first open, it is in command mode. The border is blue in command mode.

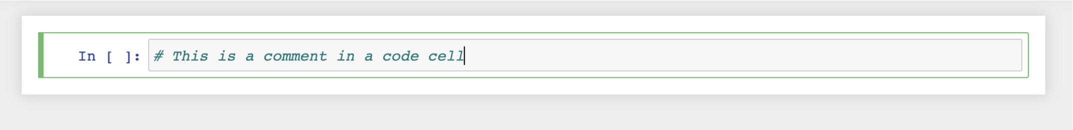


```
In [ ]: # This is a comment in a code cell
```

When you press a key in command mode it works like a shortcut. For example `p` shows the command search menu.

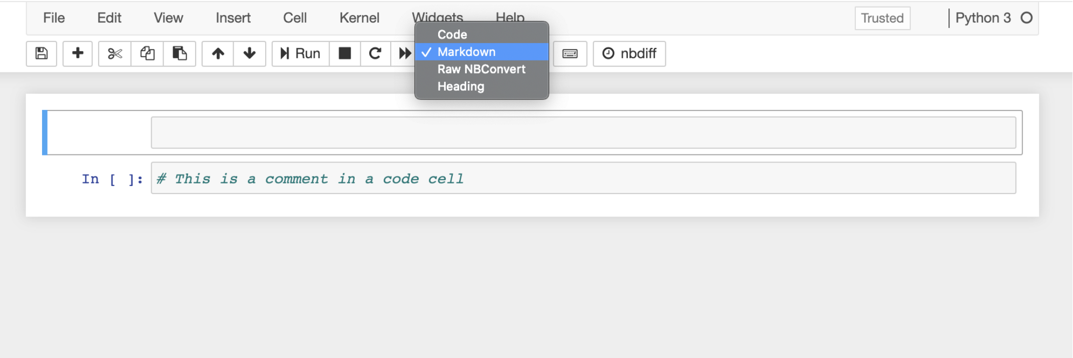

File
Edit
Python 3
In [ ]:
jupyter-notebook command group
automatically indent selection
change cell to code (command mode) Y
change cell to heading 1 (command mode) 1
change cell to heading 2 (command mode) 2
change cell to heading 3 (command mode) 3
change cell to heading 4 (command mode) 4
change cell to heading 5 (command mode) 5
change cell to heading 6 (command mode) 6
change cell to markdown (command mode) M

If you press `enter` (or `return`) or click on the highlighted cell, which is the boxes we can type in, it changes to edit mode. The border is green in edit mode



```
In [ ]: # This is a comment in a code cell
```

There are two type of cells that we will used: code and markdown. You can change that in command mode with `y` for code and `m` for markdown or on the cell type menu at the top of the notebook.



```
In [ ]: # This is a comment in a code cell
```

++

This is a markdown cell

- we can make
- itemized lists of
- bullet points

1. and we can make nubmered
2. lists, and not have to worry
3. about renumbering them
4. if we add a step in the middle later

2.1.1. Notebook Reminders

Blue border is command mode, green border is edit mode

use Escape to get to command mode

Common command mode actions:

- m: switch cell to markdown
- y: switch cell to code
- a: add a cell above
- b: add a cell below
- c: copy cell
- v: paste the cell
- 0 + 0: restart kernel
- p: command menu

use enter/return to get to edit mode

In code cells, we can use a python interpreter, for example as a calculator.

```
4+6
```

```
10
```

It prints out the last line of code that it ran, even though it executes all of them

```
name = 'sarah'
4+5
name *3
```

```
'sarahsarahsarah'
```

> **Note**
>
> For a little more python review, see my 2020 CSC310 notes this is just enough for this assignment.

2.2. Just enough Git for Assignment 1

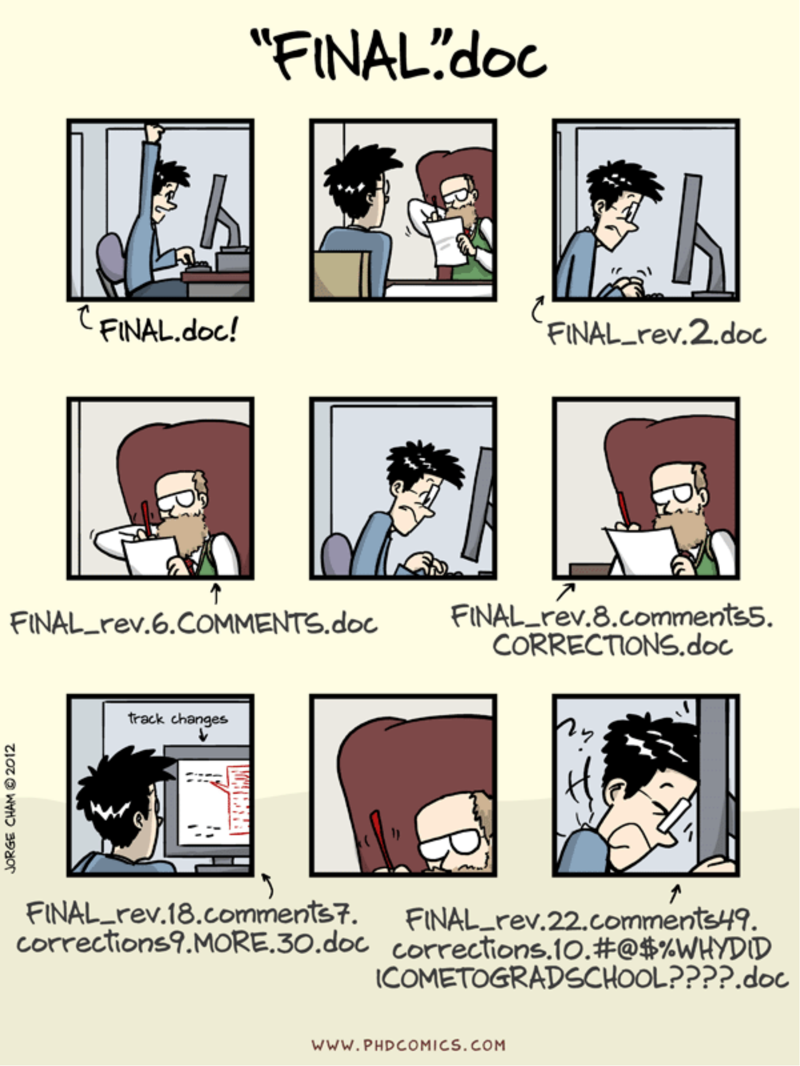
2.2.1. Assignment 1:

Goals for this assignment

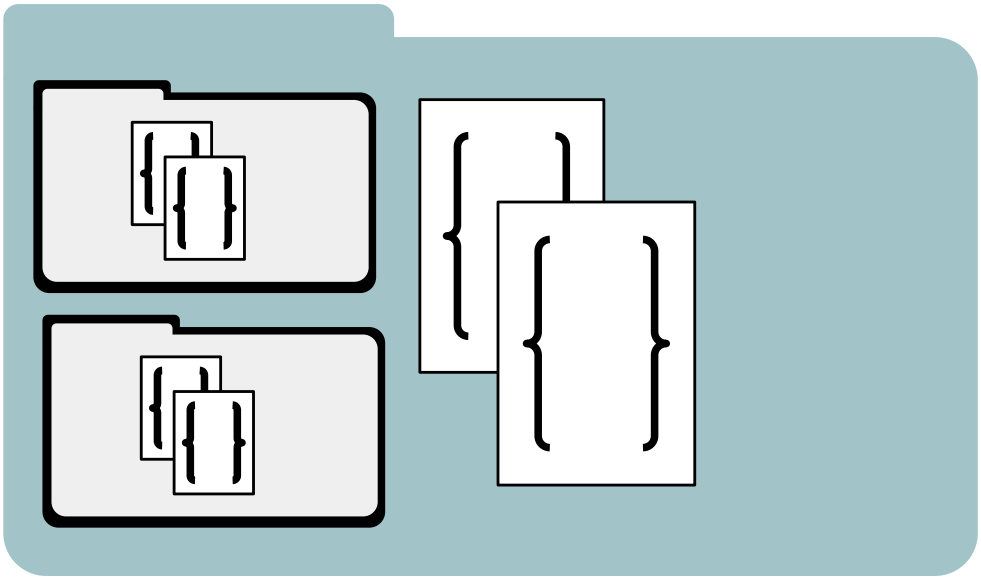
- setup your portfolio
- check that you understand the grading
- review Python basics
- practice with git and GitHub

2.2.2. Why Version control

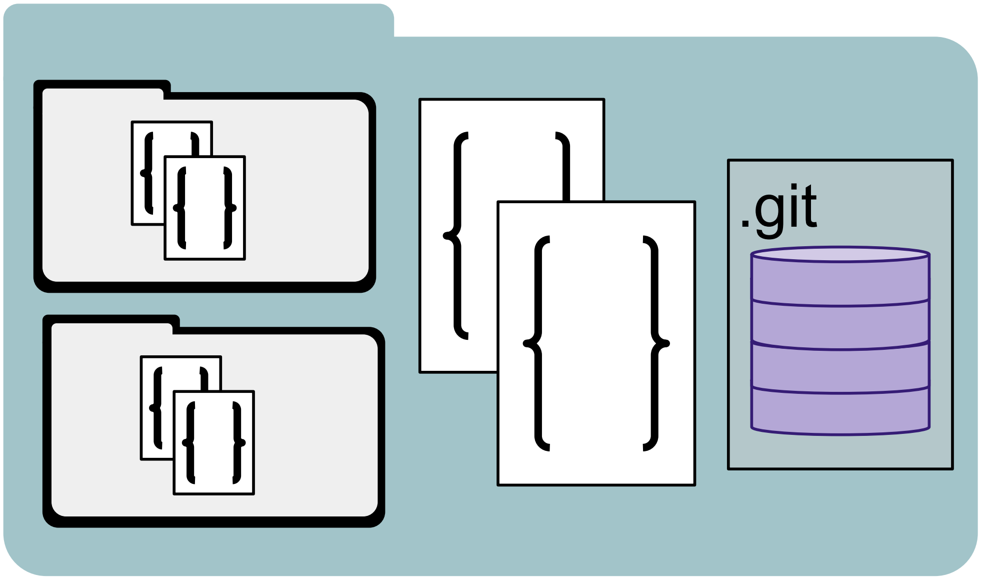
We often want to keep track of the different versions in case we want to go back, but this can be painful:



We typically organize projects in folder

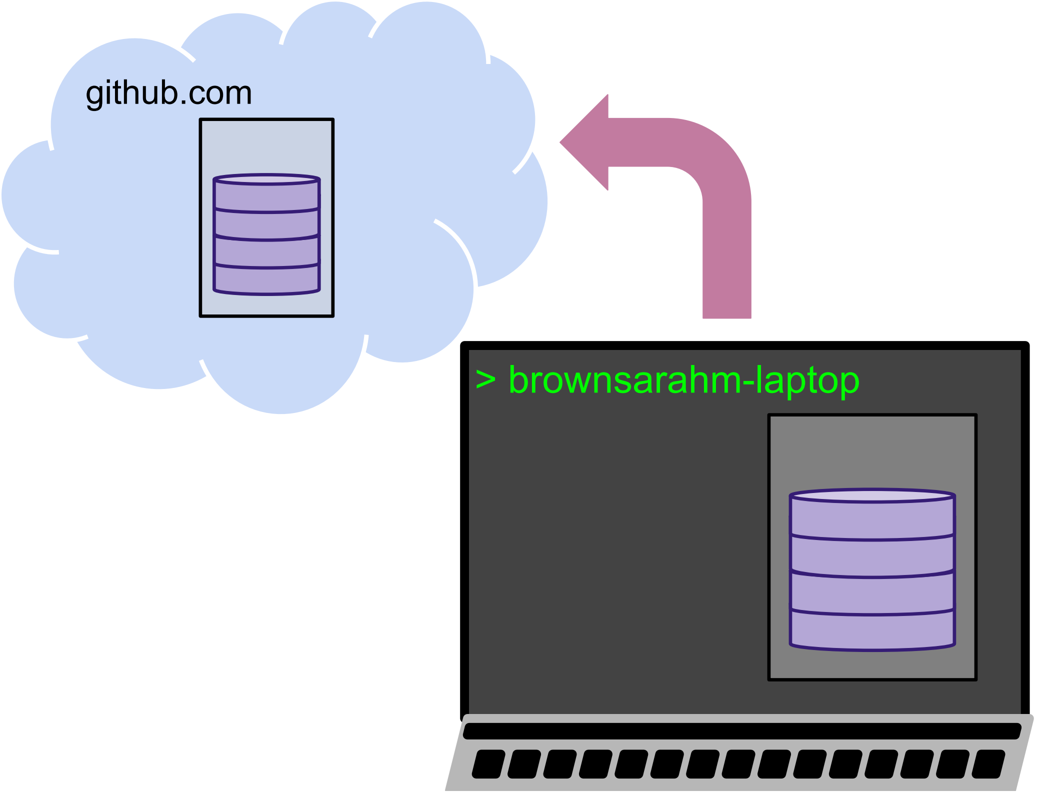


A repository is a folder with a hidden directory named `.git`

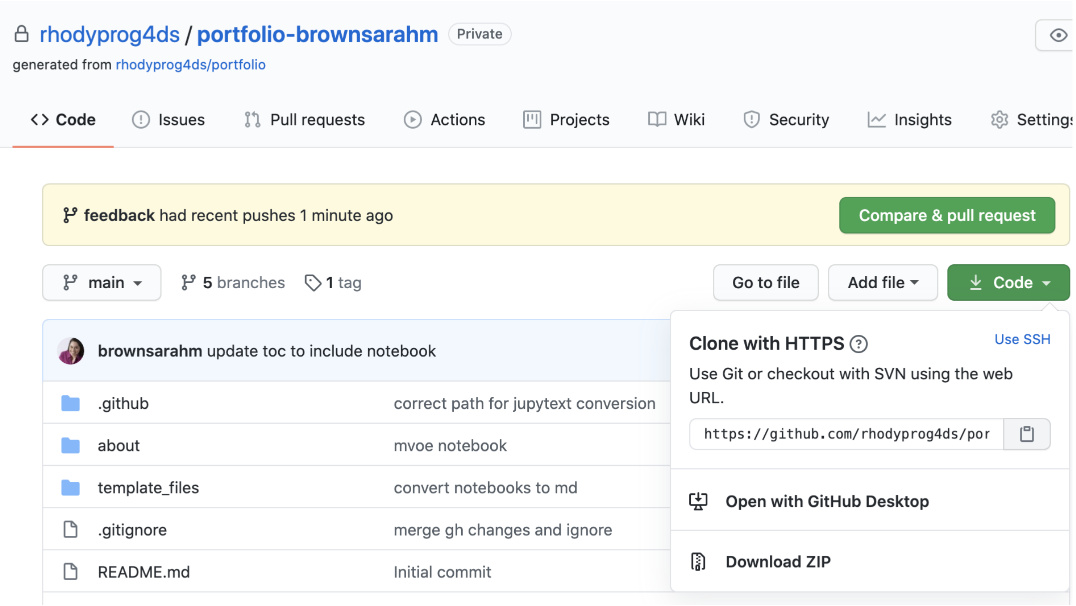


The `git` applicaiton manages that hidden directory, we don't write to it directly, which is why we keep it hidden.

Git is a distributed system, you have a local version and a remote version.



Once a repository exists on GitHub, we get a local copy by cloning it after we get its address from the GitHub interface, by clicking on the green code butto that is below the menu area to the right. It's at the top right corner of the list of files in the repository.



rhodyprog4ds / portfolio-brownsarahm

generated from rhodyprog4ds/portfolio

feedback had recent pushes 1 minute ago

5 branches **1** tag

brownsarahm update toc to include notebook

.github	correct path for jupytext conversion
about	mvoe notebook
template_files	convert notebooks to md
.gitignore	merge gh changes and ignore
README.md	Initial commit

Clone with HTTPS

Use Git or checkout with SVN using the web URL.

```
https://github.com/rhodyprog4ds/por
```

For this part, use GitBash on windows or terminal otherwise: If you set up a Personal Access Token you can use the https version

After `cd/to/where/you/want/your/repo/locally`:

```
git clone https://github.com/rhodyprog4ds/portfolio-example
```

Note

These notes can be downloaded as an actual notebook, click the GitHub logo at the top of the page and choose .ipynb. The following is not runnabel in the notebook as is.

If you set up ssh keys you use that instead

```
git clone git@github.com:rhodyprog4ds/portfolio-example.git
```

Once it's cloned, then you can navigate into the new folder:

```
cd portfolio-example
```

Then you can change files, for example adding to the intro.

Some common actions in Git, you'll want.

Check on the status of your repository:

```
git status
```

Add files to the staging area:

```
git add filename
```

Add all changes to the staging area:

```
git add .
```

Commit your changes to the repository:

```
git commit -m 'a message that will help your future self know what this part is'
```

Push your changes to GitHub

```
git push
```

Pull changes from GitHub

```
git pull
```

You can also go through these same basic steps: add, commit, push

2.3. More on git

- GitHub Hello World
- Software Carpentry Git Novice Lesson

Also, in Spring 2022, I’m teaching a section of CSC392: Topics in Computing, Introduction to Computer Systems, that will cover tools of the trade (git, bash, etc) and how they all work in great detail.

2.4. More on Python

Read Pep 8 to see what good style in Python is.

3. Getting help, object inspection, loading data

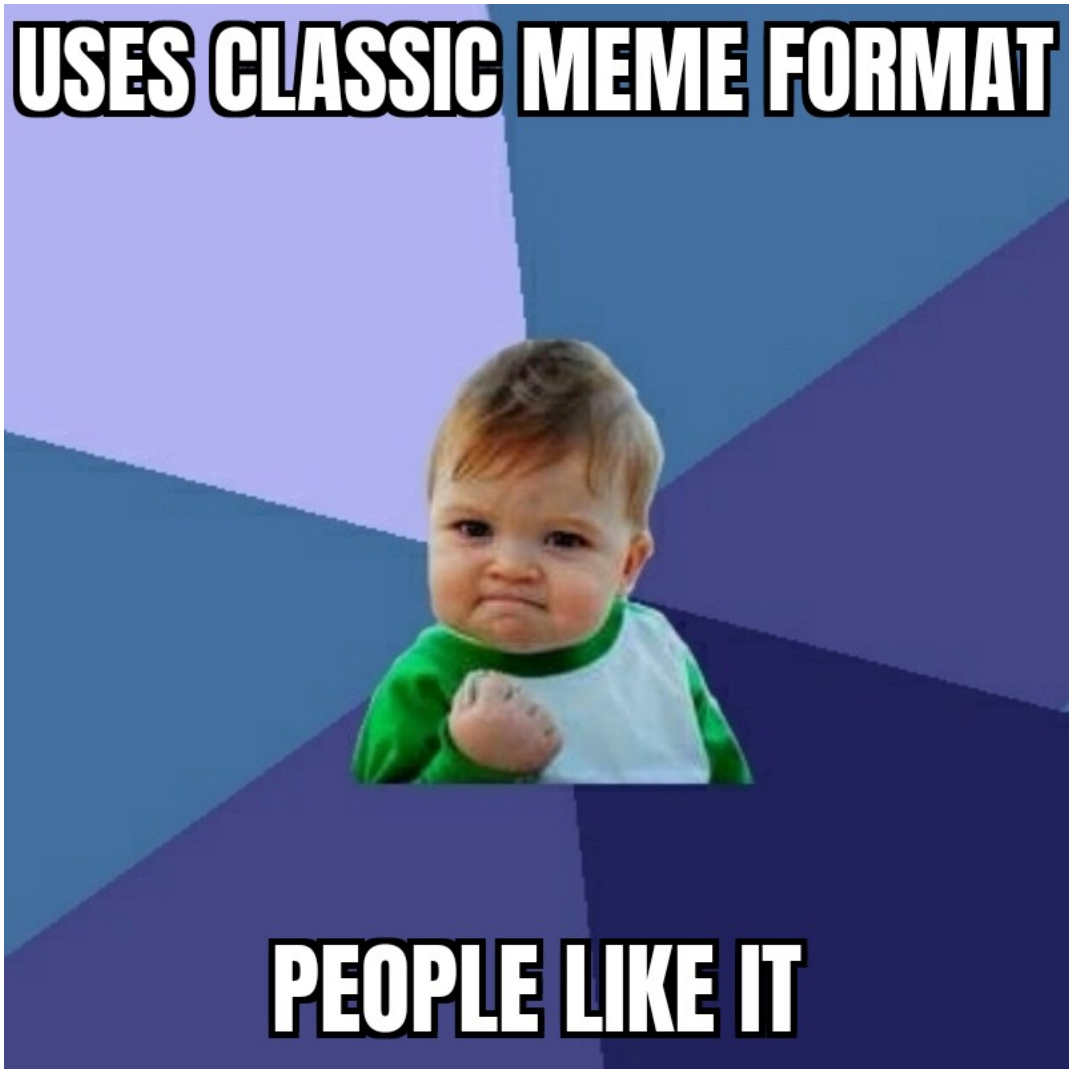
3.1. First, Don’t Worry members

Class Response Summary:

funny CS memes

David Steele
@David_C_Steele

"We just left a rest stop! Why didn't you go then?"



3.2. Getting Help in Jupyter

Python has a `print` function and we can use the help in jupyter to learn about how to use it in different ways.

Given this code excerpt, how could you print out “Sarah_Brown”?

```
first = 'Sarah'
last = 'Brown'
```

We can use jupyter popup help wiht shift +tab or `?`

```
print?
```

Or the base python `help` function

```
help(print)
```

```
Help on built-in function print in module builtins:

print(...)
    print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)

    Prints the values to a stream, or to sys.stdout by default.
    Optional keyword arguments:
    file:  a file-like object (stream); defaults to the current sys.stdout.
    sep:   string inserted between values, default a space.
    end:   string appended after the last value, default a newline.
    flush: whether to forcibly flush the stream.
```

Notice that function can take multiple arguments and has a keyword argument (must be used like `argument=value`) described as `sep=' '`. This means that by default it adds a space

```
print(first,last)
```

```
Sarah Brown
```

But we can change the separator.

```
Sarah_Brown
```

Note that it also defaults to end to use `\n`

```
print(first,last)
print('hello')
```

```
Sarah Brown
hello
```

Where does this help information come from?

Note

You can copy code from the notes, try hovering over this

```python
def compute_grade(num_level1,num_level2,num_level3):
    '''
    Computes a grade for CSC/DSP310 from numbers of achievements at each level

    Parameters:
    ------------
    num_level1 : int
      number of level 1 achievements earned
    num_level2 : int
      number of level 2 achievements earned
    num_level3 : int
      number of level 3 achievements earned

    Returns:
    --------
    letter_grade : string
      letter grade with modifier (+/-)
    '''
    if num_level1 == 15:
        if num_level2 == 15:
            if num_level3 == 15:
                grade = 'A'
            elif num_level3 >= 10:
                grade = 'A-'
            elif num_level3 >=5:
                grade = 'B+'
            else:
                grade = 'B'
        elif num_level2 >=10:
            grade = 'B-'
        elif num_level2 >=5:
            grade = 'C+'
        else:
            grade = 'C'
    elif num_level1 >= 10:
        grade = 'C-'
    elif num_level1 >= 5:
        grade = 'D+'
    elif num_level1 >=3:
        grade = 'D'
    else:
        grade = 'F'


    return grade
```

We can apply `help` on the function we wrote

```python
help(compute_grade)
```

```
Help on function compute_grade in module __main__:

compute_grade(num_level1, num_level2, num_level3)
    Computes a grade for CSC/DSP310 from numbers of achievements at each level

    Parameters:
    ------------
    num_level1 : int
      number of level 1 achievements earned
    num_level2 : int
      number of level 2 achievements earned
    num_level3 : int
      number of level 3 achievements earned

    Returns:
    --------
    letter_grade : string
      letter grade with modifier (+/-)
```

It gets the docstring

3.3. Everything is an Object in Python

we can use the builtin function `type` to inspect them, and get attributes with `.`

```python
type(compute_grade)
```

```
function
```

```
compute_grade.__name__
```

```
'compute_grade'
```

```
c = 4.5
```

```
type(c)
```

```
float
```

```
c= 'hello'
```

```
type(c)
```

```
str
```

When do we use single vs double quotes?

- You can use either, unless you need to put one inside the string then use the other.

```
my_sentence = "The professor's name is Dr. Brown"
```

```
my_sentence = 'The professor's name is Dr. Brown'
```

```
  File "/tmp/ipykernel_1651/607286316.py", line 1
    my_sentence = 'The professor's name is Dr. Brown'
                                 ^
SyntaxError: invalid syntax
```

Yes we can escape special characters:

```
my_sentence = 'The professor\'s name is Dr. Brown'
```

but, it’s less readable and not recommended.

3.4. Good Code is always relative

In programming for data science, we are often trying to tell a story.

> **Try it yourself**
>
> How might this goal change your code for this class relative to other code you have written or could imagine writing?

Python is a fully open source project and as such is governed by community standards and conventions.

> **Try it yourself**
>
> Find PEP8 (note that following it is part of earning python achievements)

The documentation for the full language is online too.

Guido van Rossum was the first main developer and wrote essays about python too.

it’s pretty popular

3.5. Coffee Data

We’re going to use a dataset about coffee quality today.

How was this dataset collected?

- reviewrs added to DB
- then scraped

Where did it come from?

- offee Quality Institute’s trained reviewers.

what format is it provided in?

- csv (Comma Separated Values)

what other information is in this repository?

- the code to scrape

Get raw url for the dataset click on the raw button on the csv page, then copy the url.
a screenshot from github of the data file page with the raw button circled in pink

We’ll save that url as a variable to work with it.

```
data_url = 'https://raw.githubusercontent.com/jldbc/coffee-quality-database/master/data/robusta_data_cleaned.csv'
```

We will use a library called Pandas

```
import pandas as pd
# import library and give it an alias (nickname) pd
```

```
pd.read_csv(data_url)
```

	Unnamed: 0	Species	Owner	Country.of.Origin	Farm.Name	Lot.Number	
0	1	Robusta	ankole coffee producers coop	Uganda	kyangundu cooperative society	NaN	a c produ
1	2	Robusta	nishant gurjer	India	sethuraman estate kaapi royale	25	sethura e
2	3	Robusta	andrew hetzel	India	sethuraman estate	NaN	
3	4	Robusta	ugacof	Uganda	ugacof project area	NaN	ug
4	5	Robusta	katuka development trust ltd	Uganda	katikamu capca farmers association	NaN	ka develop
5	6	Robusta	andrew hetzel	India	NaN	NaN	
6	7	Robusta	andrew hetzel	India	sethuraman estates	NaN	
7	8	Robusta	nishant gurjer	India	sethuraman estate kaapi royale	7	sethura e
8	9	Robusta	nishant gurjer	India	sethuraman estate	RKR	sethura e
9	10	Robusta	ugacof	Uganda	ishaka	NaN	nsu
10	11	Robusta	ugacof	Uganda	ugacof project area	NaN	ug
11	12	Robusta	nishant gurjer	India	sethuraman estate kaapi royale	RC AB	sethura e
12	13	Robusta	andrew hetzel	India	sethuraman estates	NaN	
13	14	Robusta	kasozi coffee farmers association	Uganda	kasozi coffee farmers	NaN	
14	15	Robusta	ankole coffee producers coop	Uganda	kyangundu coop society	NaN	a c produ coop u
15	16	Robusta	andrew hetzel	India	sethuraman estate	NaN	
16	17	Robusta	andrew hetzel	India	sethuraman estates	NaN	sethura es
17	18	Robusta	kawacom uganda ltd	Uganda	bushenyi	NaN	kawa
18	19	Robusta	nitubaasa ltd	Uganda	kigezi coffee farmers association	NaN	nitub
19	20	Robusta	mannya coffee project	Uganda	mannya coffee project	NaN	ma c pr
20	21	Robusta	andrew hetzel	India	sethuraman estates	NaN	
21	22	Robusta	andrew hetzel	India	sethuraman estates	NaN	sethura es
22	23	Robusta	andrew hetzel	United States	sethuraman estates	NaN	sethura es

	Unnamed: 0	Species	Owner	Country.of.Origin	Farm.Name	Lot.Number	
23	24	Robusta	luis robles	Ecuador	robustasa	Lavado 1	our ow
24	25	Robusta	luis robles	Ecuador	robustasa	Lavado 3	labor
25	26	Robusta	james moore	United States	fazenda cazengo	NaN	caz
26	27	Robusta	cafe politico	India	NaN	NaN	
27	28	Robusta	cafe politico	Vietnam	NaN	NaN	

28 rows × 44 columns

> **Try it yourself**
>
> Read the data in again, but with the index correct and save it to a variable.

Once we read it in, we van view the first 5 rows with the `head` method.

```
coffee_df.head()
```

	Species	Owner	Country.of.Origin	Farm.Name	Lot.Number	Mill	ICC
1	Robusta	ankole coffee producers coop	Uganda	kyangundu cooperative society	NaN	ankole coffee producers	
2	Robusta	nishant gurjer	India	sethuraman estate kaapi royale	25	sethuraman estate	14/114
3	Robusta	andrew hetzel	India	sethuraman estate	NaN	NaN	
4	Robusta	ugacof	Uganda	ugacof project area	NaN	ugacof	
5	Robusta	katuka development trust ltd	Uganda	katikamu capca farmers association	NaN	katuka development trust	

5 rows × 43 columns

> **Important**
>
> Remember to comment & annotate your code

3.6. Follow Up questions

3.6.1. General Questions

How do you create code to scrape data from a website and compile it into a csv file?

Will we be using pandas a lot during the semester?

3.6.2. Clarifying

How do you auto finish your directories

How do you properly shut down Jupyter Notebook

Is pd some sort of variable we set or was it built in?

How should I be organized for this class? Keep it all in a single folder? Keep it on GitHub?

I'm still not sure how to keep everything together in a portfolio for the semester?

I am still wondering if I am using anaconda or just normal terminal

Can I push this code into my portfolio using the anaconda terminal

3.6.3. Grading Questions

How do we keep track of which achievements we've earned?

I don't really have many questions from today, but I was wondering if office hours were posted.

Will we always submit homework through the portfolio folder in github?

I'm just confused as how to view my feedback from the assignment

3.6.4. Questions we'll answer later this week

- does each column have a number assigned to it in data frames?
- Can other data types be imported into a notebook and edited the same way as .csv files?

3.7. Try it yourself

- How could you check if `pd` is built in or if we defined it?
- If we wanted to see more than 5 rows when printing the head of the dataset how would we do so?

4. Pandas DataFrames

Today, we're going to explore DataFrames in greater detail. We'll continue using that same coffee dataset.

```
coffee_data_url = 'https://raw.githubusercontent.com/jldbc/coffee-quality-database/master/data/robusta_data_cleaned.csv'
```

4.1. More about loading libraries

We can import pandas without the alias `pd` if we want, but then we have to use the full name everywhere

```
import pandas
```

```
pandas.read_csv(coffee_data_url)
```

	Unnamed: 0	Species	Owner	Country.of.Origin	Farm.Name	Lot.Number	
0	1	Robusta	ankole coffee producers coop	Uganda	kyangundu cooperative society	NaN	ar c produ
1	2	Robusta	nishant gurjer	India	sethuraman estate kaapi royale	25	sethura e
2	3	Robusta	andrew hetzel	India	sethuraman estate	NaN	
3	4	Robusta	ugacof	Uganda	ugacof project area	NaN	ug
4	5	Robusta	katuka development trust ltd	Uganda	katikamu capca farmers association	NaN	ka develop
5	6	Robusta	andrew hetzel	India	NaN	NaN	
6	7	Robusta	andrew hetzel	India	sethuraman estates	NaN	
7	8	Robusta	nishant gurjer	India	sethuraman estate kaapi royale	7	sethura e
8	9	Robusta	nishant gurjer	India	sethuraman estate	RKR	sethura e
9	10	Robusta	ugacof	Uganda	ishaka	NaN	nsu
10	11	Robusta	ugacof	Uganda	ugacof project area	NaN	ug
11	12	Robusta	nishant gurjer	India	sethuraman estate kaapi royale	RC AB	sethura e
12	13	Robusta	andrew hetzel	India	sethuraman estates	NaN	
13	14	Robusta	kasozi coffee farmers association	Uganda	kasozi coffee farmers	NaN	
14	15	Robusta	ankole coffee producers coop	Uganda	kyangundu coop society	NaN	ar c produ coop u
15	16	Robusta	andrew hetzel	India	sethuraman estate	NaN	
16	17	Robusta	andrew hetzel	India	sethuraman estates	NaN	sethura es
17	18	Robusta	kawacom uganda ltd	Uganda	bushenyi	NaN	kawa
18	19	Robusta	nitubaasa ltd	Uganda	kigezi coffee farmers association	NaN	nitub
19	20	Robusta	mannya coffee project	Uganda	mannya coffee project	NaN	ma c pr
20	21	Robusta	andrew hetzel	India	sethuraman estates	NaN	
21	22	Robusta	andrew hetzel	India	sethuraman estates	NaN	sethura es
22	23	Robusta	andrew hetzel	United States	sethuraman estates	NaN	sethura es

	Unnamed: 0	Species	Owner	Country.of.Origin	Farm.Name	Lot.Number	
23	24	Robusta	luis robles	Ecuador	robustasa	Lavado 1	our ow
24	25	Robusta	luis robles	Ecuador	robustasa	Lavado 3	labor
25	26	Robusta	james moore	United States	fazenda cazengo	NaN	caz
26	27	Robusta	cafe politico	India	NaN	NaN	
27	28	Robusta	cafe politico	Vietnam	NaN	NaN	

28 rows × 44 columns

We'll use `pd` because that's the more common convention and so that we can type fewer characters throughout our code

```
import pandas as pd
```

4.2. Examining DataFrames

```
df = pd.read_csv(coffee_data_url,index_col=0)
```

We can look at the first 5 rows with `head`

```
df.head()
```

	Species	Owner	Country.of.Origin	Farm.Name	Lot.Number	Mill	ICC
1	Robusta	ankole coffee producers coop	Uganda	kyangundu cooperative society	NaN	ankole coffee producers	
2	Robusta	nishant gurjer	India	sethuraman estate kaapi royale	25	sethuraman estate	14/114
3	Robusta	andrew hetzel	India	sethuraman estate	NaN	NaN	
4	Robusta	ugacof	Uganda	ugacof project area	NaN	ugacof	
5	Robusta	katuka development trust ltd	Uganda	katikamu capca farmers association	NaN	katuka development trust	

5 rows × 43 columns

Using help, we can see tha that head takes one parameter and has a default value of 5, which is why we got 5 rows, but we can get 2 instead

```
df.head(2)
```

	Species	Owner	Country.of.Origin	Farm.Name	Lot.Number	Mill	ICO.Nu
1	Robusta	ankole coffee producers coop	Uganda	kyangundu cooperative society	NaN	ankole coffee producers	
2	Robusta	nishant gurjer	India	sethuraman estate kaapi royale	25	sethuraman estate	14/1148/20

2 rows × 43 columns

We can look at the last rows with `tail`

```
df.tail(3)
```

	Species	Owner	Country.of.Origin	Farm.Name	Lot.Number	Mill	ICO.Number	
26	Robusta	james moore	United States	fazenda cazengo	NaN	cafe cazengo	NaN	c
27	Robusta	cafe politico	India	NaN	NaN	NaN	14-1118-2014-0087	
28	Robusta	cafe politico	Vietnam	NaN	NaN	NaN	NaN	

3 rows × 43 columns

I told you this was a DataFrame, but we can check with type.

```
type(df)
```

```
pandas.core.frame.DataFrame
```

We can also exmaine its parts. It consists of several; first the column headings

```
df.columns
```

```
Index(['Species', 'Owner', 'Country.of.Origin', 'Farm.Name', 'Lot.Number',
       'Mill', 'ICO.Number', 'Company', 'Altitude', 'Region', 'Producer',
       'Number.of.Bags', 'Bag.Weight', 'In.Country.Partner', 'Harvest.Year',
       'Grading.Date', 'Owner.1', 'Variety', 'Processing.Method',
       'Fragrance...Aroma', 'Flavor', 'Aftertaste', 'Salt...Acid',
       'Bitter...Sweet', 'Mouthfeel', 'Uniform.Cup', 'Clean.Cup', 'Balance',
       'Cupper.Points', 'Total.Cup.Points', 'Moisture', 'Category.One.Defects',
       'Quakers', 'Color', 'Category.Two.Defects', 'Expiration',
       'Certification.Body', 'Certification.Address', 'Certification.Contact',
       'unit_of_measurement', 'altitude_low_meters', 'altitude_high_meters',
       'altitude_mean_meters'],
      dtype='object')
```

These are a special type called Index

```
type(df.columns)
```

```
pandas.core.indexes.base.Index
```

It also has an index

```
df.index
```

```
Int64Index([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16, 17,
            18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28],
           dtype='int64')
```

and values

```
df.values
```

```
array([['Robusta', 'ankole coffee producers coop', 'Uganda', ..., 1488.0,
        1488.0, 1488.0],
       ['Robusta', 'nishant gurjer', 'India', ..., 3170.0, 3170.0,
        3170.0],
       ['Robusta', 'andrew hetzel', 'India', ..., 1000.0, 1000.0, 1000.0],
       ...,
       ['Robusta', 'james moore', 'United States', ..., 795.0, 795.0,
        795.0],
       ['Robusta', 'cafe politico', 'India', ..., nan, nan, nan],
       ['Robusta', 'cafe politico', 'Vietnam', ..., nan, nan, nan]],
      dtype=object)
```

it also knows its own shape

```
df.shape
```

```
(28, 43)
```

we can use builtin fucntions on our DataFrame too not just its own methods and attributes.

```
len(df)
```

```
28
```

Why does `len` turn green? it's a python reserve word

4.3. Building a Data Frame programmatically

One way to build a data frame is from a dictionary:

```
people = {'names':['Sarah','Connor','Kenza'],
         'username':['brownsarahm','sudoPsych','kbdlh']}
```

```
people
```

```
{'names': ['Sarah', 'Connor', 'Kenza'],
 'username': ['brownsarahm', 'sudoPsych', 'kbdlh']}
```

```
type(people)
```

```
dict
```

```
people_df = pd.DataFrame(people)
people_df
```

	names	username
0	Sarah	brownsarahm
1	Connor	sudoPsych
2	Kenza	kbdlh

```
type(people['names'])
```

```
list
```

```
type(people)
```

```
dict
```

```
type({4,5,5})
```

```
set
```

```
{4,5,5}
```

```
{4, 5}
```

```
people['names']
```

```
['Sarah', 'Connor', 'Kenza']
```

```
type(set(people['names']))
```

```
set
```

```
unique_people = set(people['names'])
type(unique_people)
```

```
set
```

```
df.columns
```

```
Index(['Species', 'Owner', 'Country.of.Origin', 'Farm.Name', 'Lot.Number',
       'Mill', 'ICO.Number', 'Company', 'Altitude', 'Region', 'Producer',
       'Number.of.Bags', 'Bag.Weight', 'In.Country.Partner', 'Harvest.Year',
       'Grading.Date', 'Owner.1', 'Variety', 'Processing.Method',
       'Fragrance...Aroma', 'Flavor', 'Aftertaste', 'Salt...Acid',
       'Bitter...Sweet', 'Mouthfeel', 'Uniform.Cup', 'Clean.Cup', 'Balance',
       'Cupper.Points', 'Total.Cup.Points', 'Moisture', 'Category.One.Defects',
       'Quakers', 'Color', 'Category.Two.Defects', 'Expiration',
       'Certification.Body', 'Certification.Address', 'Certification.Contact',
       'unit_of_measurement', 'altitude_low_meters', 'altitude_high_meters',
       'altitude_mean_meters'],
      dtype='object')
```

```
for col in df.columns:
    print(col.split('.'))
```

```
['Species']
['Owner']
['Country', 'of', 'Origin']
['Farm', 'Name']
['Lot', 'Number']
['Mill']
['ICO', 'Number']
['Company']
['Altitude']
['Region']
['Producer']
['Number', 'of', 'Bags']
['Bag', 'Weight']
['In', 'Country', 'Partner']
['Harvest', 'Year']
['Grading', 'Date']
['Owner', '1']
['Variety']
['Processing', 'Method']
['Fragrance', '', '', 'Aroma']
['Flavor']
['Aftertaste']
['Salt', '', '', 'Acid']
['Bitter', '', '', 'Sweet']
['Mouthfeel']
['Uniform', 'Cup']
['Clean', 'Cup']
['Balance']
['Cupper', 'Points']
['Total', 'Cup', 'Points']
['Moisture']
['Category', 'One', 'Defects']
['Quakers']
['Color']
['Category', 'Two', 'Defects']
['Expiration']
['Certification', 'Body']
['Certification', 'Address']
['Certification', 'Contact']
['unit_of_measurement']
['altitude_low_meters']
['altitude_high_meters']
['altitude_mean_meters']
```

```
for key,value in people.items():
    print(key,':',value)
```

```
names : ['Sarah', 'Connor', 'Kenza']
username : ['brownsarahm', 'sudoPsych', 'kbdlh']
```

```
df['Owner']
```

```
1          ankole coffee producers coop
2                        nishant gurjer
3                         andrew hetzel
4                                ugacof
5          katuka development trust ltd
6                         andrew hetzel
7                         andrew hetzel
8                        nishant gurjer
9                        nishant gurjer
10                               ugacof
11                               ugacof
12                       nishant gurjer
13                        andrew hetzel
14    kasozi coffee farmers association
15         ankole coffee producers coop
16                        andrew hetzel
17                        andrew hetzel
18                   kawacom uganda ltd
19                        nitubaasa ltd
20                mannya coffee project
21                        andrew hetzel
22                        andrew hetzel
23                        andrew hetzel
24                          luis robles
25                          luis robles
26                          james moore
27                        cafe politico
28                        cafe politico
Name: Owner, dtype: object
```

```
df.Owner
```

```
1          ankole coffee producers coop
2                        nishant gurjer
3                         andrew hetzel
4                                ugacof
5          katuka development trust ltd
6                         andrew hetzel
7                         andrew hetzel
8                        nishant gurjer
9                        nishant gurjer
10                               ugacof
11                               ugacof
12                       nishant gurjer
13                        andrew hetzel
14    kasozi coffee farmers association
15         ankole coffee producers coop
16                        andrew hetzel
17                        andrew hetzel
18                   kawacom uganda ltd
19                        nitubaasa ltd
20                mannya coffee project
21                        andrew hetzel
22                        andrew hetzel
23                        andrew hetzel
24                          luis robles
25                          luis robles
26                          james moore
27                        cafe politico
28                        cafe politico
Name: Owner, dtype: object
```

```
df
```

	Species	Owner	Country.of.Origin	Farm.Name	Lot.Number	Mill	IC
1	Robusta	ankole coffee producers coop	Uganda	kyangundu cooperative society	NaN	ankole coffee producers	
2	Robusta	nishant gurjer	India	sethuraman estate kaapi royale	25	sethuraman estate	14/114
3	Robusta	andrew hetzel	India	sethuraman estate	NaN	NaN	
4	Robusta	ugacof	Uganda	ugacof project area	NaN	ugacof	
5	Robusta	katuka development trust ltd	Uganda	katikamu capca farmers association	NaN	katuka development trust	
6	Robusta	andrew hetzel	India	NaN	NaN	(self)	
7	Robusta	andrew hetzel	India	sethuraman estates	NaN	NaN	
8	Robusta	nishant gurjer	India	sethuraman estate kaapi royale	7	sethuraman estate	14/114
9	Robusta	nishant gurjer	India	sethuraman estate	RKR	sethuraman estate	14/114
10	Robusta	ugacof	Uganda	ishaka	NaN	nsubuga umar	
11	Robusta	ugacof	Uganda	ugacof project area	NaN	ugacof	
12	Robusta	nishant gurjer	India	sethuraman estate kaapi royale	RC AB	sethuraman estate	14/114
13	Robusta	andrew hetzel	India	sethuraman estates	NaN	NaN	
14	Robusta	kasozi coffee farmers association	Uganda	kasozi coffee farmers	NaN	NaN	
15	Robusta	ankole coffee producers coop	Uganda	kyangundu coop society	NaN	ankole coffee producers coop union ltd	
16	Robusta	andrew hetzel	India	sethuraman estate	NaN	NaN	
17	Robusta	andrew hetzel	India	sethuraman estates	NaN	sethuraman estates	
18	Robusta	kawacom uganda ltd	Uganda	bushenyi	NaN	kawacom	
19	Robusta	nitubaasa ltd	Uganda	kigezi coffee farmers association	NaN	nitubaasa	
20	Robusta	mannya coffee project	Uganda	mannya coffee project	NaN	mannya coffee project	
21	Robusta	andrew hetzel	India	sethuraman estates	NaN	NaN	
22	Robusta	andrew hetzel	India	sethuraman estates	NaN	sethuraman estates	
23	Robusta	andrew hetzel	United States	sethuraman estates	NaN	sethuraman estates	

	Species	Owner	Country.of.Origin	Farm.Name	Lot.Number	Mill	IC
24	Robusta	luis robles	Ecuador	robustasa	Lavado 1	our own lab	
25	Robusta	luis robles	Ecuador	robustasa	Lavado 3	own laboratory	
26	Robusta	james moore	United States	fazenda cazengo	NaN	cafe cazengo	
27	Robusta	cafe politico	India	NaN	NaN	NaN	14-1
28	Robusta	cafe politico	Vietnam	NaN	NaN	NaN	

28 rows × 43 columns

Key points:

write three things to remember from today’s class

4.4. Questions After Classroom

many overlapping questions today

4.5. General

How to know which function to use in certain problems or situations

4.6. Clarifying

Is there a way to have a set show the duplicates that get discarded?

being able to access the code somewhere without asking to scroll would be nice

4.7. Course Admin

When will homeworks be posted/due typically?

4.8. Questions we’ll answer later

can you use cast a pandas dataframe into a set?

4.9. Try it yourself

- Create variables of three different types with facts about yourself. Use descriptive variable names relative to the contents, not their types.

- Create a list, again with a descriptive name, and print out the types

```
<class 'str'>
<class 'int'>
<class 'list'>
```

- Write a function, `type_extractor` that takes a list and a type and returns the item of that type from the list
- Test your function on all three items from your dictionary.
- Use one type of jupyter help on your function, what does it display? If it doesn't display anything modify your function so that help will work.
- Make yourself notes in the most memorable way for you about what a DataFrame is.

5. More Loading Data, Indexing, and Iterables

As always, we'll start with loading pandas.

```
import pandas as pd
```

5.1. Checking in on help hours

if you missed class, check over the office hours schedule and e-mail if you can or cannot attend at least one time

5.2. Portfolio Preparation and Maintainance

We'll spend a little time today getting your portfolio ready for the first check.

5.2.1. Access your portfolio

Go to your portflio

- from the course organization
- from the list of your recent repositories on the left hand side of the GitHub home page

optionally, open it locally as well (we're going to update content and)

5.2.2. Start your Know, Want to Know, Learned Table

In each portfolio submission introduction, you'll reflect on what you've learned. To get ready for that, we'll first make note of what you already know and what you want to know.

1. edit `submission_1_intro` in your portfolio locally or on GitHub:
2. In the KWL section in the first two bullets after each skill with what you know and want to know. You can edit these in more detail later.

This will render as a table in your built portflio, for reference on the syntax, refer to the Tables section of the jupyterbook Myst Markdown cheatsheet.

> ⚠ **Warning**
>
> If you work on this in the GitHub website, be sure to pull these chances locally before you start working offline next

5.2.3. Merge the setup work

Once you're done, Go to your pull request tab, and select the feedback Pull Reques. Commit any suggestions if you'd like and then merge the PR.

> ⚠ **Warning**
>
> only do this after grading

Note

To view the feedback, after merging the PR, remove `is:open` from the search bar on the PR page

5.3. Indexing

```
topics = ['what is data science', 'jupyter',
          'conditional','functions', 'lists',
          'dictionaries','pandas' ]
```

What will topics[-1] return?

```
topics[-1]
```

```
'pandas'
```

Using negative indices starts from the right. The last element is -1. The first is 0.

5.4. Reading DataFrames from Websites

We'll first read from the course website.

```
course_comms_url =
'https://rhodyprog4ds.github.io/BrownFall21/syllabus/communication.html'
```

So far, we've read data in from a .csv file with `pd.read_csv` and created a DataFrame with the constructor `pd.DataFrame` using a dictionary. Pandas provides many interfaces for reading in data. They're described on the Pandas IO page.

We can use the `read_html` method to read from this page. We know that it has multple tables on the page, so lets see what it t does:

Note

Using the documentation for a library (and the base language) is totally expected and normal part of programming. That's what you should use as your primary source for questions in this class. Other sources can become outdated pretty quickly as the language changes, but most of the libraries we'll use have processes in place to ensure that their own documentation gets updated at the same time the code does.

Warning

If you use other sources and get advised to solutions that are deprecated you may not earn achievements for that work.

```
pd.read_html(course_comms_url)
```

```
[              Day               Time             Location      Host
 0          Monday    1:00:00 PM-2:30     inperson roomtbd  Chamundi
 1       Wednesday         4:00:00 PM     inperson roomtbd  Chamundi
 2        Wedneday       2:00:00 PM-3     inperson roomtbd  Chamundi
 3       Wednesday    7:00:00 PM-8:30          gather.town     Sarah
 4          Friday  5:00:00 PM-6:30pm          gather.town  Chamundi
 5  By appointment                TBD  in person Tyler 134     Sarah,
                                           usage platform  \
 0                                      in class  prismia   
 1                                      any time  prismia   
 2                                      any time  prismia   
 3          private questions to your assignment   github   
 4    for general questions that can help others   github   
 5                            to share resources   github   
 6  matters that don't fit into another category   e-mail   

                        area                                               note  
 0                      chat  outside of class time this is not monitored cl...  
 1             message board                          for discussion with peers  
 2       download transcript  use after class to get preliminary notes eg if...  
 3  issue on assignment repo                              eg bugs in your code"  
 4   issue on course website  eg what the instructions of an assignment mean...  
 5   pull request on website       remember to request ram tokens if applicable  
 6    to brownsarahm@uri.edu  remember to include `[CSC310]` or `[DSP310]` (...  ,
                                           usage                    area  \
 0  matters that don't fit into another category  to brownsarahm@uri.edu   

                                                 note  
 0  remember to include `[CSC310]` or `[DSP310]` (...  ,
                                         usage                     area  \
 0        private questions to your assignment  issue on assignment repo   
 1  for general questions that can help others   issue on course website   
 2                          to share resources   pull request on website   

                                                 note  
 0                              eg bugs in your code"  
 1  eg what the instructions of an assignment mean...  
 2       remember to request ram tokens if applicable  ,
       usage                 area  \
 0  in class                 chat   
 1  any time        message board   
 2  any time  download transcript   

                                                 note  
 0  outside of class time this is not monitored cl...  
 1                          for discussion with peers  
 2  use after class to get preliminary notes eg if...  ]
```

It appears to have read all of them, lets check the type:

```
type(pd.read_html(course_comms_url))
```

```
list
```

Since we know it’s a list, we’ll save it to a variable that indicates that.

```
comms_list = pd.read_html(course_comms_url)
```

If we get just the first element,

```
type(comms_list[0])
```

```
pandas.core.frame.DataFrame
```

it’s a DataFrame and prints accordingly.

```
comms_list[0]
```

	Day	Time	Location	Host
0	Monday	1:00:00 PM-2:30	inperson roomtbd	Chamundi
1	Wednesday	4:00:00 PM	inperson roomtbd	Chamundi
2	Wedneday	2:00:00 PM-3	inperson roomtbd	Chamundi
3	Wednesday	7:00:00 PM-8:30	gather.town	Sarah
4	Friday	5:00:00 PM-6:30pm	gather.town	Chamundi
5	By appointment	TBD	in person Tyler 134	Sarah

Since it’s a list, we can use base python’s `len` function to check how many tables there are

```
len(comms_list)
```

```
5
```

We’ve seen the first table and know it’s the help hours, so we can save that to a separate variable and use it

```
help_df = comms_list[0]
```

We’ve inspected the dataframe some before, but we can also check the type of each column.

```
help_df.dtypes
```

```
Day         object
Time        object
Location    object
Host        object
dtype: object
```

Note

You can read more about the details of data types in Pandas

Question: Why does it have `dtype:object` after the type for each row?

First to understand this, let’s save the thing we’re curious to a variable so we can examine it multiple ways more easily.

```
help_df_types = help_df.dtypes
```

Next we’ll check the type of this object and its shape

```
type(help_df_types)
```

```
pandas.core.series.Series
```

a `Series` is like a dataframee, but just one row with headings, and then rotated.

```
help_df_types.shape
```

```
(4,)
```

This means that it’s length is 4 and it’s a 1 dimensional object; the column headers have converted to an index and are treated as metadata, but not a part of the actual data.

So, the line we’re interested in is not a part of the object, because it’s length 4 and the thing we’re curious about is the fifth line.

We’ll pick one variable from the DataFrame and check its type

```
type(help_df['Day'])
```

```
pandas.core.series.Series
```

This is also a Series, so lets check its output

```
help_df['Day']
```

```
0           Monday
1        Wednesday
2         Wedneday
3        Wednesday
4           Friday
5    By appointment
Name: Day, dtype: object
```

THe last line of this one is information about the Series, its name, and its dtype.

Let’s make another series, and see how it prints

```
pd.Series([5,4,5])
```

```
0    5
1    4
2    5
dtype: int64
```

The last line is the dtype of the Series; so in our original object, that last line is because the list of dtypes is the of type object.

```
help_df_types
```

```
Day         object
Time        object
Location    object
Host        object
dtype: object
```

5.5. How do we know what to check?

we examined the DataFrame so far by (me) knowing what to look for.

In python objects you can progrmamatically find what to look for with the `__dict__` attribute or we can rely on the [online documentation](#) or use it via help.

In ipython (what we use in jupyter, by default) we can use the `?` for help

```
pd.DataFrame?
```

```
help(pd.DataFrame)
```

Everything is Data

writing good documentation lets people who use your code get help for free Not only do help tools use the docs, but that website is generated programmatically using a tool called Sphinx from the documentation inside the code. You can access the docstring of a python using the `.__doc__` attribute

[ipython help](#) read about how it works, if it doesn’t work for you to try to figure out why

```
Help on class DataFrame in module pandas.core.frame:

class DataFrame(pandas.core.generic.NDFrame, pandas.core.arraylike.OpsMixin)
 |  DataFrame(data=None, index: 'Axes | None' = None, columns: 'Axes | None' = None,
dtype: 'Dtype | None' = None, copy: 'bool | None' = None)
 |
 |  Two-dimensional, size-mutable, potentially heterogeneous tabular data.
 |
 |  Data structure also contains labeled axes (rows and columns).
 |  Arithmetic operations align on both row and column labels. Can be
 |  thought of as a dict-like container for Series objects. The primary
 |  pandas data structure.
 |
 |  Parameters
 |  ----------
 |  data : ndarray (structured or homogeneous), Iterable, dict, or DataFrame
 |      Dict can contain Series, arrays, constants, dataclass or list-like objects.
If
 |      data is a dict, column order follows insertion-order.
 |
 |      .. versionchanged:: 0.25.0
 |         If data is a list of dicts, column order follows insertion-order.
 |
 |  index : Index or array-like
 |      Index to use for resulting frame. Will default to RangeIndex if
 |      no indexing information part of input data and no index provided.
 |  columns : Index or array-like
 |      Column labels to use for resulting frame when data does not have them,
 |      defaulting to RangeIndex(0, 1, 2, ..., n). If data contains column labels,
 |      will perform column selection instead.
 |  dtype : dtype, default None
 |      Data type to force. Only a single dtype is allowed. If None, infer.
 |  copy : bool or None, default None
 |      Copy data from inputs.
 |      For dict data, the default of None behaves like ``copy=True``.  For DataFrame
 |      or 2d ndarray input, the default of None behaves like ``copy=False``.
 |
 |      .. versionchanged:: 1.3.0
 |
 |  See Also
 |  --------
 |  DataFrame.from_records : Constructor from tuples, also record arrays.
 |  DataFrame.from_dict : From dicts of Series, arrays, or dicts.
 |  read_csv : Read a comma-separated values (csv) file into DataFrame.
 |  read_table : Read general delimited file into DataFrame.
 |  read_clipboard : Read text from clipboard into DataFrame.
 |
 |  Examples
 |  --------
 |  Constructing DataFrame from a dictionary.
 |
 |  >>> d = {'col1': [1, 2], 'col2': [3, 4]}
 |  >>> df = pd.DataFrame(data=d)
 |  >>> df
 |     col1  col2
 |  0     1     3
 |  1     2     4
 |
 |  Notice that the inferred dtype is int64.
 |
 |  >>> df.dtypes
 |  col1    int64
 |  col2    int64
 |  dtype: object
 |
 |  To enforce a single dtype:
 |
 |  >>> df = pd.DataFrame(data=d, dtype=np.int8)
 |  >>> df.dtypes
 |  col1    int8
 |  col2    int8
 |  dtype: object
 |
 |  Constructing DataFrame from numpy ndarray:
 |
 |  >>> df2 = pd.DataFrame(np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]]),
 |  ...                    columns=['a', 'b', 'c'])
 |  >>> df2
 |     a  b  c
 |  0  1  2  3
 |  1  4  5  6
 |  2  7  8  9
 |
 |  Constructing DataFrame from a numpy ndarray that has labeled columns:
 |
 |  >>> data = np.array([(1, 2, 3), (4, 5, 6), (7, 8, 9)],
 |  ...                 dtype=[("a", "i4"), ("b", "i4"), ("c", "i4")])
 |  >>> df3 = pd.DataFrame(data, columns=['c', 'a'])
 |  ...
 |  >>> df3
 |     c  a
 |  0  3  1
 |  1  6  4
 |  2  9  7
 |
 |  Constructing DataFrame from dataclass:
 |
 |  >>> from dataclasses import make_dataclass
 |  >>> Point = make_dataclass("Point", [("x", int), ("y", int)])
 |  >>> pd.DataFrame([Point(0, 0), Point(0, 3), Point(2, 3)])
 |     x  y
 |  0  0  0
 |  1  0  3
 |  2  2  3
 |
 |  Method resolution order:
 |      DataFrame
 |      pandas.core.generic.NDFrame
 |      pandas.core.base.PandasObject
 |      pandas.core.accessor.DirNamesMixin
 |      pandas.core.indexing.IndexingMixin
 |      pandas.core.arraylike.OpsMixin
 |      builtins.object
 |
 |  Methods defined here:
 |
 |  __divmod__(self, other) -> 'tuple[DataFrame, DataFrame]'
 |  
 |  __getitem__(self, key)
 |  
 |  __init__(self, data=None, index: 'Axes | None' = None, columns: 'Axes | None' = 
None, dtype: 'Dtype | None' = None, copy: 'bool | None' = None)
 |      Initialize self.  See help(type(self)) for accurate signature.
 |  
 |  __len__(self) -> 'int'
 |      Returns length of info axis, but here we use the index.
 |  
 |  __matmul__(self, other: 'AnyArrayLike | FrameOrSeriesUnion') -> 
'FrameOrSeriesUnion'
 |      Matrix multiplication using binary `@` operator in Python>=3.5.
 |  
 |  __rdivmod__(self, other) -> 'tuple[DataFrame, DataFrame]'
 |  
 |  __repr__(self) -> 'str'
 |      Return a string representation for a particular DataFrame.
 |  
 |  __rmatmul__(self, other)
 |      Matrix multiplication using binary `@` operator in Python>=3.5.
 |  
 |  __setitem__(self, key, value)
 |  
 |  add(self, other, axis='columns', level=None, fill_value=None)
 |      Get Addition of dataframe and other, element-wise (binary operator `add`).
 |      
 |      Equivalent to ``dataframe + other``, but with support to substitute a 
fill_value
 |      for missing data in one of the inputs. With reverse version, `radd`.
 |      
 |      Among flexible wrappers (`add`, `sub`, `mul`, `div`, `mod`, `pow`) to
 |      arithmetic operators: `+`, `-`, `*`, `/`, `//`, `%`, `**`.
 |      
 |      Parameters
 |      ----------
 |      other : scalar, sequence, Series, or DataFrame
 |          Any single or multiple element data structure, or list-like object.
 |      axis : {0 or 'index', 1 or 'columns'}
 |          Whether to compare by the index (0 or 'index') or columns
 |          (1 or 'columns'). For Series input, axis to match Series index on.
 |      level : int or label
 |          Broadcast across a level, matching Index values on the
 |          passed MultiIndex level.
 |      fill_value : float or None, default None
 |          Fill existing missing (NaN) values, and any new element needed for
 |          successful DataFrame alignment, with this value before computation.
 |          If data in both corresponding DataFrame locations is missing
 |          the result will be missing.
 |      
 |      Returns
 |      -------
 |      DataFrame
 |          Result of the arithmetic operation.
 |      
 |      See Also
 |      --------
 |      DataFrame.add : Add DataFrames.
 |      DataFrame.sub : Subtract DataFrames.
 |      DataFrame.mul : Multiply DataFrames.
 |      DataFrame.div : Divide DataFrames (float division).
 |      DataFrame.truediv : Divide DataFrames (float division).
 |      DataFrame.floordiv : Divide DataFrames (integer division).
 |      DataFrame.mod : Calculate modulo (remainder after division).
 |      DataFrame.pow : Calculate exponential power.
 |      
 |      Notes
 |      -----
 |      Mismatched indices will be unioned together.
 |      
 |      Examples
 |      --------
 |      >>> df = pd.DataFrame({'angles': [0, 3, 4],
 |      ...                    'degrees': [360, 180, 360]},
 |      ...                   index=['circle', 'triangle', 'rectangle'])
 |      >>> df
 |                 angles  degrees
 |      circle          0      360
 |      triangle        3      180
 |      rectangle       4      360
 |      
 |      Add a scalar with operator version which return the same
 |      results.
 |      
 |      >>> df + 1
 |                 angles  degrees
 |      circle          1      361
 |      triangle        4      181
 |      rectangle       5      361
 |
 |      >>> df.add(1)
 |                 angles  degrees
 |      circle          1      361
 |      triangle        4      181
 |      rectangle       5      361
 |
 |      Divide by constant with reverse version.
 |
 |      >>> df.div(10)
 |                 angles  degrees
 |      circle        0.0     36.0
 |      triangle      0.3     18.0
 |      rectangle     0.4     36.0
 |
 |      >>> df.rdiv(10)
 |                   angles   degrees
 |      circle          inf  0.027778
 |      triangle   3.333333  0.055556
 |      rectangle  2.500000  0.027778
 |
 |      Subtract a list and Series by axis with operator version.
 |
 |      >>> df - [1, 2]
 |                 angles  degrees
 |      circle         -1      358
 |      triangle        2      178
 |      rectangle       3      358
 |
 |      >>> df.sub([1, 2], axis='columns')
 |                 angles  degrees
 |      circle         -1      358
 |      triangle        2      178
 |      rectangle       3      358
 |
 |      >>> df.sub(pd.Series([1, 1, 1], index=['circle', 'triangle', 'rectangle']),
 |      ...        axis='index')
 |                 angles  degrees
 |      circle         -1      359
 |      triangle        2      179
 |      rectangle       3      359
 |
 |      Multiply a DataFrame of different shape with operator version.
 |
 |      >>> other = pd.DataFrame({'angles': [0, 3, 4]},
 |      ...                      index=['circle', 'triangle', 'rectangle'])
 |      >>> other
 |                 angles
 |      circle          0
 |      triangle        3
 |      rectangle       4
 |
 |      >>> df * other
 |                 angles  degrees
 |      circle          0      NaN
 |      triangle        9      NaN
 |      rectangle      16      NaN
 |
 |      >>> df.mul(other, fill_value=0)
 |                 angles  degrees
 |      circle          0      0.0
 |      triangle        9      0.0
 |      rectangle      16      0.0
 |
 |      Divide by a MultiIndex by level.
 |
 |      >>> df_multindex = pd.DataFrame({'angles': [0, 3, 4, 4, 5, 6],
 |      ...                              'degrees': [360, 180, 360, 360, 540, 720]},
 |      ...                             index=[['A', 'A', 'A', 'B', 'B', 'B'],
 |      ...                                    ['circle', 'triangle', 'rectangle',
 |      ...                                     'square', 'pentagon', 'hexagon']])
 |      >>> df_multindex
 |                   angles  degrees
 |      A circle          0      360
 |        triangle        3      180
 |        rectangle       4      360
 |      B square          4      360
 |        pentagon        5      540
 |        hexagon         6      720
 |
 |      >>> df.div(df_multindex, level=1, fill_value=0)
 |                   angles  degrees
 |      A circle        NaN      1.0
 |        triangle      1.0      1.0
 |        rectangle     1.0      1.0
```

```
 |      B square        0.0      0.0
 |        pentagon      0.0      0.0
 |        hexagon       0.0      0.0
 |
 |  agg = aggregate(self, func=None, axis: 'Axis' = 0, *args, **kwargs)
 |
 |  aggregate(self, func=None, axis: 'Axis' = 0, *args, **kwargs)
 |      Aggregate using one or more operations over the specified axis.
 |
 |      Parameters
 |      ----------
 |      func : function, str, list or dict
 |          Function to use for aggregating the data. If a function, must either
 |          work when passed a DataFrame or when passed to DataFrame.apply.
 |
 |          Accepted combinations are:
 |
 |          - function
 |          - string function name
 |          - list of functions and/or function names, e.g. ``[np.sum, 'mean']``
 |          - dict of axis labels -> functions, function names or list of such.
 |      axis : {0 or 'index', 1 or 'columns'}, default 0
 |              If 0 or 'index': apply function to each column.
 |              If 1 or 'columns': apply function to each row.
 |      *args
 |          Positional arguments to pass to `func`.
 |      **kwargs
 |          Keyword arguments to pass to `func`.
 |
 |      Returns
 |      -------
 |      scalar, Series or DataFrame
 |
 |          The return can be:
 |
 |          * scalar : when Series.agg is called with single function
 |          * Series : when DataFrame.agg is called with a single function
 |          * DataFrame : when DataFrame.agg is called with several functions
 |
 |          Return scalar, Series or DataFrame.
 |
 |      The aggregation operations are always performed over an axis, either the
 |      index (default) or the column axis. This behavior is different from
 |      `numpy` aggregation functions (`mean`, `median`, `prod`, `sum`, `std`,
 |      `var`), where the default is to compute the aggregation of the flattened
 |      array, e.g., ``numpy.mean(arr_2d)`` as opposed to
 |      ``numpy.mean(arr_2d, axis=0)``.
 |
 |      `agg` is an alias for `aggregate`. Use the alias.
 |
 |      See Also
 |      --------
 |      DataFrame.apply : Perform any type of operations.
 |      DataFrame.transform : Perform transformation type operations.
 |      core.groupby.GroupBy : Perform operations over groups.
 |      core.resample.Resampler : Perform operations over resampled bins.
 |      core.window.Rolling : Perform operations over rolling window.
 |      core.window.Expanding : Perform operations over expanding window.
 |      core.window.ExponentialMovingWindow : Perform operation over exponential
weighted
 |          window.
 |
 |      Notes
 |      -----
 |      `agg` is an alias for `aggregate`. Use the alias.
 |
 |      Functions that mutate the passed object can produce unexpected
 |      behavior or errors and are not supported. See :ref:`gotchas.udf-mutation`
 |      for more details.
 |
 |      A passed user-defined-function will be passed a Series for evaluation.
 |
 |      Examples
 |      --------
 |      >>> df = pd.DataFrame([[1, 2, 3],
 |      ...                    [4, 5, 6],
 |      ...                    [7, 8, 9],
 |      ...                    [np.nan, np.nan, np.nan]],
 |      ...                   columns=['A', 'B', 'C'])
 |
 |      Aggregate these functions over the rows.
 |
 |      >>> df.agg(['sum', 'min'])
 |              A     B     C
 |      sum  12.0  15.0  18.0
 |      min   1.0   2.0   3.0
 |
 |      Different aggregations per column.
 |
 |      >>> df.agg({'A' : ['sum', 'min'], 'B' : ['min', 'max']})
 |              A    B
 |      sum  12.0  NaN
 |      min   1.0  2.0
 |      max   NaN  8.0
 |
 |      Aggregate different functions over the columns and rename the index of the
resulting
 |      DataFrame.
 |
 |      >>> df.agg(x=('A', max), y=('B', 'min'), z=('C', np.mean))
 |           A    B    C
 |      x  7.0  NaN  NaN
 |      y  NaN  2.0  NaN
 |      z  NaN  NaN  6.0
 |
 |      Aggregate over the columns.
 |
 |      >>> df.agg("mean", axis="columns")
 |      0    2.0
 |      1    5.0
 |      2    8.0
 |      3    NaN
 |      dtype: float64
 |
 |  align(self, other, join: 'str' = 'outer', axis: 'Axis | None' = None, level:
'Level | None' = None, copy: 'bool' = True, fill_value=None, method: 'str | None' =
None, limit=None, fill_axis: 'Axis' = 0, broadcast_axis: 'Axis | None' = None) ->
'DataFrame'
 |      Align two objects on their axes with the specified join method.
 |
 |      Join method is specified for each axis Index.
 |
 |      Parameters
 |      ----------
 |      other : DataFrame or Series
 |      join : {'outer', 'inner', 'left', 'right'}, default 'outer'
 |      axis : allowed axis of the other object, default None
 |          Align on index (0), columns (1), or both (None).
 |      level : int or level name, default None
 |          Broadcast across a level, matching Index values on the
 |          passed MultiIndex level.
 |      copy : bool, default True
 |          Always returns new objects. If copy=False and no reindexing is
 |          required then original objects are returned.
 |      fill_value : scalar, default np.NaN
 |          Value to use for missing values. Defaults to NaN, but can be any
 |          "compatible" value.
 |      method : {'backfill', 'bfill', 'pad', 'ffill', None}, default None
 |          Method to use for filling holes in reindexed Series:
 |
 |          - pad / ffill: propagate last valid observation forward to next valid.
 |          - backfill / bfill: use NEXT valid observation to fill gap.
 |
 |      limit : int, default None
 |          If method is specified, this is the maximum number of consecutive
 |          NaN values to forward/backward fill. In other words, if there is
 |          a gap with more than this number of consecutive NaNs, it will only
 |          be partially filled. If method is not specified, this is the
 |          maximum number of entries along the entire axis where NaNs will be
 |          filled. Must be greater than 0 if not None.
 |      fill_axis : {0 or 'index', 1 or 'columns'}, default 0
 |          Filling axis, method and limit.
 |      broadcast_axis : {0 or 'index', 1 or 'columns'}, default None
 |          Broadcast values along this axis, if aligning two objects of
 |          different dimensions.
 |
 |      Returns
 |      -------
 |      (left, right) : (DataFrame, type of other)
 |          Aligned objects.
 |
 |  all(self, axis=0, bool_only=None, skipna=True, level=None, **kwargs)
 |      Return whether all elements are True, potentially over an axis.
 |
 |      Returns True unless there at least one element within a series or
 |      along a Dataframe axis that is False or equivalent (e.g. zero or
 |      empty).
 |
 |      Parameters
 |      ----------
 |      axis : {0 or 'index', 1 or 'columns', None}, default 0
 |          Indicate which axis or axes should be reduced.
 |
 |          * 0 / 'index' : reduce the index, return a Series whose index is the
 |            original column labels.
 |          * 1 / 'columns' : reduce the columns, return a Series whose index is the
 |            original index.
 |          * None : reduce all axes, return a scalar.
 |
 |      bool_only : bool, default None
 |          Include only boolean columns. If None, will attempt to use everything,
 |          then use only boolean data. Not implemented for Series.
 |      skipna : bool, default True
 |          Exclude NA/null values. If the entire row/column is NA and skipna is
 |          True, then the result will be True, as for an empty row/column.
 |          If skipna is False, then NA are treated as True, because these are not
 |          equal to zero.
 |      level : int or level name, default None
 |          If the axis is a MultiIndex (hierarchical), count along a
 |          particular level, collapsing into a Series.
 |      **kwargs : any, default None
 |          Additional keywords have no effect but might be accepted for
 |          compatibility with NumPy.
 |
 |      Returns
 |      -------
 |      Series or DataFrame
 |          If level is specified, then, DataFrame is returned; otherwise, Series
 |          is returned.
 |
 |      See Also
 |      --------
 |      Series.all : Return True if all elements are True.
 |      DataFrame.any : Return True if one (or more) elements are True.
 |
 |      Examples
 |      --------
 |      **Series**
 |
 |      >>> pd.Series([True, True]).all()
 |      True
 |      >>> pd.Series([True, False]).all()
 |      False
 |      >>> pd.Series([], dtype="float64").all()
 |      True
 |      >>> pd.Series([np.nan]).all()
 |      True
 |      >>> pd.Series([np.nan]).all(skipna=False)
 |      True
 |
 |      **DataFrames**
 |
 |      Create a dataframe from a dictionary.
 |
 |      >>> df = pd.DataFrame({'col1': [True, True], 'col2': [True, False]})
 |      >>> df
 |         col1   col2
 |      0  True   True
 |      1  True  False
 |
 |      Default behaviour checks if column-wise values all return True.
 |
 |      >>> df.all()
 |      col1     True
 |      col2    False
 |      dtype: bool
 |
 |      Specify ``axis='columns'`` to check if row-wise values all return True.
 |
 |      >>> df.all(axis='columns')
 |      0     True
 |      1    False
 |      dtype: bool
 |
 |      Or ``axis=None`` for whether every value is True.
 |
 |      >>> df.all(axis=None)
 |      False
 |
 |  any(self, axis=0, bool_only=None, skipna=True, level=None, **kwargs)
 |      Return whether any element is True, potentially over an axis.
 |
 |      Returns False unless there is at least one element within a series or
 |      along a Dataframe axis that is True or equivalent (e.g. non-zero or
 |      non-empty).
 |
 |      Parameters
 |      ----------
 |      axis : {0 or 'index', 1 or 'columns', None}, default 0
 |          Indicate which axis or axes should be reduced.
 |
 |          * 0 / 'index' : reduce the index, return a Series whose index is the
 |            original column labels.
 |          * 1 / 'columns' : reduce the columns, return a Series whose index is the
 |            original index.
 |          * None : reduce all axes, return a scalar.
 |
 |      bool_only : bool, default None
 |          Include only boolean columns. If None, will attempt to use everything,
 |          then use only boolean data. Not implemented for Series.
 |      skipna : bool, default True
 |          Exclude NA/null values. If the entire row/column is NA and skipna is
 |          True, then the result will be False, as for an empty row/column.
 |          If skipna is False, then NA are treated as True, because these are not
 |          equal to zero.
 |      level : int or level name, default None
 |          If the axis is a MultiIndex (hierarchical), count along a
 |          particular level, collapsing into a Series.
 |      **kwargs : any, default None
 |          Additional keywords have no effect but might be accepted for
 |          compatibility with NumPy.
 |
 |      Returns
 |      -------
 |      Series or DataFrame
 |          If level is specified, then, DataFrame is returned; otherwise, Series
 |          is returned.
 |
 |      See Also
 |      --------
 |      numpy.any : Numpy version of this method.
 |      Series.any : Return whether any element is True.
 |      Series.all : Return whether all elements are True.
 |      DataFrame.any : Return whether any element is True over requested axis.
 |      DataFrame.all : Return whether all elements are True over requested axis.
 |
 |      Examples
 |      --------
 |      **Series**
 |
 |      For Series input, the output is a scalar indicating whether any element
 |      is True.
 |
 |      >>> pd.Series([False, False]).any()
 |      False
 |      >>> pd.Series([True, False]).any()
 |      True
 |      >>> pd.Series([], dtype="float64").any()
 |      False
 |      >>> pd.Series([np.nan]).any()
 |      False
 |      >>> pd.Series([np.nan]).any(skipna=False)
 |      True
 |
 |      **DataFrame**
 |
 |      Whether each column contains at least one True element (the default).
 |
 |      >>> df = pd.DataFrame({"A": [1, 2], "B": [0, 2], "C": [0, 0]})
 |      >>> df
 |         A  B  C
 |      0  1  0  0
 |      1  2  2  0
 |
 |      >>> df.any()
 |      A     True
 |      B     True
 |      C    False
 |      dtype: bool
 |
 |      Aggregating over the columns.
 |
 |      >>> df = pd.DataFrame({"A": [True, False], "B": [1, 2]})
 |      >>> df
 |             A  B
 |      0   True  1
 |      1  False  2
 |
 |      >>> df.any(axis='columns')
 |      0    True
 |      1    True
 |      dtype: bool
 |
 |      >>> df = pd.DataFrame({"A": [True, False], "B": [1, 0]})
 |      >>> df
 |             A  B
 |      0   True  1
 |      1  False  0
```

```
 |
 |      >>> df.any(axis='columns')
 |      0    True
 |      1    False
 |      dtype: bool
 |
 |      Aggregating over the entire DataFrame with ``axis=None``.
 |
 |      >>> df.any(axis=None)
 |      True
 |
 |      `any` for an empty DataFrame is an empty Series.
 |
 |      >>> pd.DataFrame([]).any()
 |      Series([], dtype: bool)
 |
 |  append(self, other, ignore_index: 'bool' = False, verify_integrity: 'bool' =
False, sort: 'bool' = False) -> 'DataFrame'
 |      Append rows of `other` to the end of caller, returning a new object.
 |
 |      Columns in `other` that are not in the caller are added as new columns.
 |
 |      Parameters
 |      ----------
 |      other : DataFrame or Series/dict-like object, or list of these
 |          The data to append.
 |      ignore_index : bool, default False
 |          If True, the resulting axis will be labeled 0, 1, …, n - 1.
 |      verify_integrity : bool, default False
 |          If True, raise ValueError on creating index with duplicates.
 |      sort : bool, default False
 |          Sort columns if the columns of `self` and `other` are not aligned.
 |
 |          .. versionchanged:: 1.0.0
 |
 |              Changed to not sort by default.
 |
 |      Returns
 |      -------
 |      DataFrame
 |          A new DataFrame consisting of the rows of caller and the rows of `other`.
 |
 |      See Also
 |      --------
 |      concat : General function to concatenate DataFrame or Series objects.
 |
 |      Notes
 |      -----
 |      If a list of dict/series is passed and the keys are all contained in
 |      the DataFrame's index, the order of the columns in the resulting
 |      DataFrame will be unchanged.
 |
 |      Iteratively appending rows to a DataFrame can be more computationally
 |      intensive than a single concatenate. A better solution is to append
 |      those rows to a list and then concatenate the list with the original
 |      DataFrame all at once.
 |
 |      Examples
 |      --------
 |      >>> df = pd.DataFrame([[1, 2], [3, 4]], columns=list('AB'), index=['x', 'y'])
 |      >>> df
 |         A  B
 |      x  1  2
 |      y  3  4
 |      >>> df2 = pd.DataFrame([[5, 6], [7, 8]], columns=list('AB'), index=['x',
'y'])
 |      >>> df.append(df2)
 |         A  B
 |      x  1  2
 |      y  3  4
 |      x  5  6
 |      y  7  8
 |
 |      With `ignore_index` set to True:
 |
 |      >>> df.append(df2, ignore_index=True)
 |         A  B
 |      0  1  2
 |      1  3  4
 |      2  5  6
 |      3  7  8
 |
 |      The following, while not recommended methods for generating DataFrames,
 |      show two ways to generate a DataFrame from multiple data sources.
 |
 |      Less efficient:
 |
 |      >>> df = pd.DataFrame(columns=['A'])
 |      >>> for i in range(5):
 |      ...     df = df.append({'A': i}, ignore_index=True)
 |      >>> df
 |         A
 |      0  0
 |      1  1
 |      2  2
 |      3  3
 |      4  4
 |      
 |      More efficient:
 |      
 |      >>> pd.concat([pd.DataFrame([i], columns=['A']) for i in range(5)],
 |      ...           ignore_index=True)
 |         A
 |      0  0
 |      1  1
 |      2  2
 |      3  3
 |      4  4
 |  
 |  apply(self, func: 'AggFuncType', axis: 'Axis' = 0, raw: 'bool' = False,
result_type=None, args=(), **kwargs)
 |      Apply a function along an axis of the DataFrame.
 |      
 |      Objects passed to the function are Series objects whose index is
 |      either the DataFrame's index (``axis=0``) or the DataFrame's columns
 |      (``axis=1``). By default (``result_type=None``), the final return type
 |      is inferred from the return type of the applied function. Otherwise,
 |      it depends on the `result_type` argument.
 |      
 |      Parameters
 |      ----------
 |      func : function
 |          Function to apply to each column or row.
 |      axis : {0 or 'index', 1 or 'columns'}, default 0
 |          Axis along which the function is applied:
 |      
 |          * 0 or 'index': apply function to each column.
 |          * 1 or 'columns': apply function to each row.
 |      
 |      raw : bool, default False
 |          Determines if row or column is passed as a Series or ndarray object:
 |      
 |          * ``False`` : passes each row or column as a Series to the
 |            function.
 |          * ``True`` : the passed function will receive ndarray objects
 |            instead.
 |            If you are just applying a NumPy reduction function this will
 |            achieve much better performance.
 |      
 |      result_type : {'expand', 'reduce', 'broadcast', None}, default None
 |          These only act when ``axis=1`` (columns):
 |      
 |          * 'expand' : list-like results will be turned into columns.
 |          * 'reduce' : returns a Series if possible rather than expanding
 |            list-like results. This is the opposite of 'expand'.
 |          * 'broadcast' : results will be broadcast to the original shape
 |            of the DataFrame, the original index and columns will be
 |            retained.
 |      
 |          The default behaviour (None) depends on the return value of the
 |          applied function: list-like results will be returned as a Series
 |          of those. However if the apply function returns a Series these
 |          are expanded to columns.
 |      args : tuple
 |          Positional arguments to pass to `func` in addition to the
 |          array/series.
 |      **kwargs
 |          Additional keyword arguments to pass as keywords arguments to
 |          `func`.
 |      
 |      Returns
 |      -------
 |      Series or DataFrame
 |          Result of applying ``func`` along the given axis of the
 |          DataFrame.
 |      
 |      See Also
 |      --------
 |      DataFrame.applymap: For elementwise operations.
 |      DataFrame.aggregate: Only perform aggregating type operations.
 |      DataFrame.transform: Only perform transforming type operations.
 |      
 |      Notes
 |      -----
 |      Functions that mutate the passed object can produce unexpected
 |      behavior or errors and are not supported. See :ref:`gotchas.udf-mutation`
 |      for more details.
 |      
 |      Examples
 |      --------
 |      >>> df = pd.DataFrame([[4, 9]] * 3, columns=['A', 'B'])
 |      >>> df
 |         A  B
 |      0  4  9
 |      1  4  9
 |      2  4  9
 |      
 |      Using a numpy universal function (in this case the same as
 |      ``np.sqrt(df)``):
 |      
 |      >>> df.apply(np.sqrt)
 |           A    B
 |      0  2.0  3.0
 |      1  2.0  3.0
 |      2  2.0  3.0
 |      
 |      Using a reducing function on either axis
 |      
 |      >>> df.apply(np.sum, axis=0)
 |      A    12
 |      B    27
 |      dtype: int64
 |      
 |      >>> df.apply(np.sum, axis=1)
 |      0    13
 |      1    13
 |      2    13
 |      dtype: int64
 |      
 |      Returning a list-like will result in a Series
 |      
 |      >>> df.apply(lambda x: [1, 2], axis=1)
 |      0    [1, 2]
 |      1    [1, 2]
 |      2    [1, 2]
 |      dtype: object
 |      
 |      Passing ``result_type='expand'`` will expand list-like results
 |      to columns of a Dataframe
 |      
 |      >>> df.apply(lambda x: [1, 2], axis=1, result_type='expand')
 |         0  1
 |      0  1  2
 |      1  1  2
 |      2  1  2
 |      
 |      Returning a Series inside the function is similar to passing
 |      ``result_type='expand'``. The resulting column names
 |      will be the Series index.
 |      
 |      >>> df.apply(lambda x: pd.Series([1, 2], index=['foo', 'bar']), axis=1)
 |         foo  bar
 |      0    1    2
 |      1    1    2
 |      2    1    2
 |      
 |      Passing ``result_type='broadcast'`` will ensure the same shape
 |      result, whether list-like or scalar is returned by the function,
 |      and broadcast it along the axis. The resulting column names will
 |      be the originals.
 |      
 |      >>> df.apply(lambda x: [1, 2], axis=1, result_type='broadcast')
 |         A  B
 |      0  1  2
 |      1  1  2
 |      2  1  2
 |  
 |  applymap(self, func: 'PythonFuncType', na_action: 'str | None' = None, **kwargs)
-> 'DataFrame'
 |      Apply a function to a Dataframe elementwise.
 |      
 |      This method applies a function that accepts and returns a scalar
 |      to every element of a DataFrame.
 |      
 |      Parameters
 |      ----------
 |      func : callable
 |          Python function, returns a single value from a single value.
 |      na_action : {None, 'ignore'}, default None
 |          If ‘ignore’, propagate NaN values, without passing them to func.
 |      
 |          .. versionadded:: 1.2
 |      
 |      **kwargs
 |          Additional keyword arguments to pass as keywords arguments to
 |          `func`.
 |      
 |          .. versionadded:: 1.3.0
 |      
 |      Returns
 |      -------
 |      DataFrame
 |          Transformed DataFrame.
 |      
 |      See Also
 |      --------
 |      DataFrame.apply : Apply a function along input axis of DataFrame.
 |      
 |      Examples
 |      --------
 |      >>> df = pd.DataFrame([[1, 2.12], [3.356, 4.567]])
 |      >>> df
 |             0      1
 |      0  1.000  2.120
 |      1  3.356  4.567
 |      
 |      >>> df.applymap(lambda x: len(str(x)))
 |         0  1
 |      0  3  4
 |      1  5  5
 |      
 |      Like Series.map, NA values can be ignored:
 |      
 |      >>> df_copy = df.copy()
 |      >>> df_copy.iloc[0, 0] = pd.NA
 |      >>> df_copy.applymap(lambda x: len(str(x)), na_action='ignore')
 |            0  1
 |      0  <NA>  4
 |      1     5  5
 |      
 |      Note that a vectorized version of `func` often exists, which will
 |      be much faster. You could square each number elementwise.
 |      
 |      >>> df.applymap(lambda x: x**2)
 |                 0          1
 |      0   1.000000   4.494400
 |      1  11.262736  20.857489
 |      
 |      But it's better to avoid applymap in that case.
 |      
 |      >>> df ** 2
 |                 0          1
 |      0   1.000000   4.494400
 |      1  11.262736  20.857489
 |  
 |  asfreq(self, freq: 'Frequency', method=None, how: 'str | None' = None, normalize: 
'bool' = False, fill_value=None) -> 'DataFrame'
 |      Convert time series to specified frequency.
 |      
 |      Returns the original data conformed to a new index with the specified
 |      frequency.
 |      
 |      If the index of this DataFrame is a :class:`~pandas.PeriodIndex`, the new 
index
 |      is the result of transforming the original index with
 |      :meth:`PeriodIndex.asfreq <pandas.PeriodIndex.asfreq>` (so the original index
 |      will map one-to-one to the new index).
 |      
 |      Otherwise, the new index will be equivalent to ``pd.date_range(start, end,
 |      freq=freq)`` where ``start`` and ``end`` are, respectively, the first and
 |      last entries in the original index (see :func:`pandas.date_range`). The
 |      values corresponding to any timesteps in the new index which were not present
 |      in the original index will be null (``NaN``), unless a method for filling
 |      such unknowns is provided (see the ``method`` parameter below).
 |      
 |      The :meth:`resample` method is more appropriate if an operation on each group 
of
 |      timesteps (such as an aggregate) is necessary to represent the data at the 
new
 |      frequency.
 |      
 |      Parameters
 |      ----------
 |      freq : DateOffset or str
 |          Frequency DateOffset or string.
 |      method : {'backfill'/'bfill', 'pad'/'ffill'}, default None
 |          Method to use for filling holes in reindexed Series (note this
 |          does not fill NaNs that already were present):
```

```
 |
 |          * 'pad' / 'ffill': propagate last valid observation forward to next
 |            valid
 |          * 'backfill' / 'bfill': use NEXT valid observation to fill.
 |      how : {'start', 'end'}, default end
 |          For PeriodIndex only (see PeriodIndex.asfreq).
 |      normalize : bool, default False
 |          Whether to reset output index to midnight.
 |      fill_value : scalar, optional
 |          Value to use for missing values, applied during upsampling (note
 |          this does not fill NaNs that already were present).
 |
 |      Returns
 |      -------
 |      DataFrame
 |          DataFrame object reindexed to the specified frequency.
 |
 |      See Also
 |      --------
 |      reindex : Conform DataFrame to new index with optional filling logic.
 |
 |      Notes
 |      -----
 |      To learn more about the frequency strings, please see `this link
 |      <https://pandas.pydata.org/pandas-
docs/stable/user_guide/timeseries.html#offset-aliases>`__.
 |
 |      Examples
 |      --------
 |      Start by creating a series with 4 one minute timestamps.
 |
 |      >>> index = pd.date_range('1/1/2000', periods=4, freq='T')
 |      >>> series = pd.Series([0.0, None, 2.0, 3.0], index=index)
 |      >>> df = pd.DataFrame({'s': series})
 |      >>> df
 |                             s
 |      2000-01-01 00:00:00    0.0
 |      2000-01-01 00:01:00    NaN
 |      2000-01-01 00:02:00    2.0
 |      2000-01-01 00:03:00    3.0
 |
 |      Upsample the series into 30 second bins.
 |
 |      >>> df.asfreq(freq='30S')
 |                             s
 |      2000-01-01 00:00:00    0.0
 |      2000-01-01 00:00:30    NaN
 |      2000-01-01 00:01:00    NaN
 |      2000-01-01 00:01:30    NaN
 |      2000-01-01 00:02:00    2.0
 |      2000-01-01 00:02:30    NaN
 |      2000-01-01 00:03:00    3.0
 |
 |      Upsample again, providing a ``fill value``.
 |
 |      >>> df.asfreq(freq='30S', fill_value=9.0)
 |                             s
 |      2000-01-01 00:00:00    0.0
 |      2000-01-01 00:00:30    9.0
 |      2000-01-01 00:01:00    NaN
 |      2000-01-01 00:01:30    9.0
 |      2000-01-01 00:02:00    2.0
 |      2000-01-01 00:02:30    9.0
 |      2000-01-01 00:03:00    3.0
 |
 |      Upsample again, providing a ``method``.
 |
 |      >>> df.asfreq(freq='30S', method='bfill')
 |                             s
 |      2000-01-01 00:00:00    0.0
 |      2000-01-01 00:00:30    NaN
 |      2000-01-01 00:01:00    NaN
 |      2000-01-01 00:01:30    2.0
 |      2000-01-01 00:02:00    2.0
 |      2000-01-01 00:02:30    3.0
 |      2000-01-01 00:03:00    3.0
 |
 |  assign(self, **kwargs) -> 'DataFrame'
 |      Assign new columns to a DataFrame.
 |
 |      Returns a new object with all original columns in addition to new ones.
 |      Existing columns that are re-assigned will be overwritten.
 |
 |      Parameters
 |      ----------
 |      **kwargs : dict of {str: callable or Series}
 |          The column names are keywords. If the values are
 |          callable, they are computed on the DataFrame and
 |          assigned to the new columns. The callable must not
 |          change input DataFrame (though pandas doesn't check it).
 |          If the values are not callable, (e.g. a Series, scalar, or array),
 |          they are simply assigned.
 |
 |      Returns
 |      -------
 |      DataFrame
 |          A new DataFrame with the new columns in addition to
 |          all the existing columns.
 |
 |      Notes
 |      -----
 |      Assigning multiple columns within the same ``assign`` is possible.
 |      Later items in '\*\*kwargs' may refer to newly created or modified
 |      columns in 'df'; items are computed and assigned into 'df' in order.
 |
 |      Examples
 |      --------
 |      >>> df = pd.DataFrame({'temp_c': [17.0, 25.0]},
 |      ...                   index=['Portland', 'Berkeley'])
 |      >>> df
 |                temp_c
 |      Portland    17.0
 |      Berkeley    25.0
 |
 |      Where the value is a callable, evaluated on `df`:
 |
 |      >>> df.assign(temp_f=lambda x: x.temp_c * 9 / 5 + 32)
 |                temp_c  temp_f
 |      Portland    17.0    62.6
 |      Berkeley    25.0    77.0
 |
 |      Alternatively, the same behavior can be achieved by directly
 |      referencing an existing Series or sequence:
 |
 |      >>> df.assign(temp_f=df['temp_c'] * 9 / 5 + 32)
 |                temp_c  temp_f
 |      Portland    17.0    62.6
 |      Berkeley    25.0    77.0
 |
 |      You can create multiple columns within the same assign where one
 |      of the columns depends on another one defined within the same assign:
 |
 |      >>> df.assign(temp_f=lambda x: x['temp_c'] * 9 / 5 + 32,
 |      ...           temp_k=lambda x: (x['temp_f'] +  459.67) * 5 / 9)
 |                temp_c  temp_f  temp_k
 |      Portland    17.0    62.6  290.15
 |      Berkeley    25.0    77.0  298.15
 |
 |  bfill(self: 'DataFrame', axis: 'None | Axis' = None, inplace: 'bool' = False,
limit: 'None | int' = None, downcast=None) -> 'DataFrame | None'
 |      Synonym for :meth:`DataFrame.fillna` with ``method='bfill'``.
 |
 |      Returns
 |      -------
 |      Series/DataFrame or None
 |          Object with missing values filled or None if ``inplace=True``.
 |
 |  boxplot = boxplot_frame(self, column=None, by=None, ax=None, fontsize=None,
rot=0, grid=True, figsize=None, layout=None, return_type=None, backend=None,
**kwargs)
 |      Make a box plot from DataFrame columns.
 |
 |      Make a box-and-whisker plot from DataFrame columns, optionally grouped
 |      by some other columns. A box plot is a method for graphically depicting
 |      groups of numerical data through their quartiles.
 |      The box extends from the Q1 to Q3 quartile values of the data,
 |      with a line at the median (Q2). The whiskers extend from the edges
 |      of box to show the range of the data. By default, they extend no more than
 |      `1.5 * IQR (IQR = Q3 - Q1)` from the edges of the box, ending at the farthest
 |      data point within that interval. Outliers are plotted as separate dots.
 |
 |      For further details see
 |      Wikipedia's entry for `boxplot <https://en.wikipedia.org/wiki/Box_plot>`_.
 |
 |      Parameters
 |      ----------
 |      column : str or list of str, optional
 |          Column name or list of names, or vector.
 |          Can be any valid input to :meth:`pandas.DataFrame.groupby`.
 |      by : str or array-like, optional
 |          Column in the DataFrame to :meth:`pandas.DataFrame.groupby`.
 |          One box-plot will be done per value of columns in `by`.
 |      ax : object of class matplotlib.axes.Axes, optional
 |          The matplotlib axes to be used by boxplot.
 |      fontsize : float or str
 |          Tick label font size in points or as a string (e.g., `large`).
 |      rot : int or float, default 0
 |          The rotation angle of labels (in degrees)
 |          with respect to the screen coordinate system.
 |      grid : bool, default True
 |          Setting this to True will show the grid.
 |      figsize : A tuple (width, height) in inches
 |          The size of the figure to create in matplotlib.
 |      layout : tuple (rows, columns), optional
 |          For example, (3, 5) will display the subplots
 |          using 3 columns and 5 rows, starting from the top-left.
 |      return_type : {'axes', 'dict', 'both'} or None, default 'axes'
 |          The kind of object to return. The default is ``axes``.
 |
 |          * 'axes' returns the matplotlib axes the boxplot is drawn on.
 |          * 'dict' returns a dictionary whose values are the matplotlib
 |            Lines of the boxplot.
 |          * 'both' returns a namedtuple with the axes and dict.
 |          * when grouping with ``by``, a Series mapping columns to
 |            ``return_type`` is returned.
 |
 |            If ``return_type`` is `None`, a NumPy array
 |            of axes with the same shape as ``layout`` is returned.
 |      backend : str, default None
 |          Backend to use instead of the backend specified in the option
 |          ``plotting.backend``. For instance, 'matplotlib'. Alternatively, to
 |          specify the ``plotting.backend`` for the whole session, set
 |          ``pd.options.plotting.backend``.
 |
 |          .. versionadded:: 1.0.0
 |
 |      **kwargs
 |          All other plotting keyword arguments to be passed to
 |          :func:`matplotlib.pyplot.boxplot`.
 |
 |      Returns
 |      -------
 |      result
 |          See Notes.
 |
 |      See Also
 |      --------
 |      Series.plot.hist: Make a histogram.
 |      matplotlib.pyplot.boxplot : Matplotlib equivalent plot.
 |
 |      Notes
 |      -----
 |      The return type depends on the `return_type` parameter:
 |
 |      * 'axes' : object of class matplotlib.axes.Axes
 |      * 'dict' : dict of matplotlib.lines.Line2D objects
 |      * 'both' : a namedtuple with structure (ax, lines)
 |
 |      For data grouped with ``by``, return a Series of the above or a numpy
 |      array:
 |
 |      * :class:`~pandas.Series`
 |      * :class:`~numpy.array` (for ``return_type = None``)
 |
 |      Use ``return_type='dict'`` when you want to tweak the appearance
 |      of the lines after plotting. In this case a dict containing the Lines
 |      making up the boxes, caps, fliers, medians, and whiskers is returned.
 |
 |      Examples
 |      --------
 |
 |      Boxplots can be created for every column in the dataframe
 |      by ``df.boxplot()`` or indicating the columns to be used:
 |
 |      .. plot::
 |          :context: close-figs
 |
 |          >>> np.random.seed(1234)
 |          >>> df = pd.DataFrame(np.random.randn(10, 4),
 |          ...                   columns=['Col1', 'Col2', 'Col3', 'Col4'])
 |          >>> boxplot = df.boxplot(column=['Col1', 'Col2', 'Col3'])
 |
 |      Boxplots of variables distributions grouped by the values of a third
 |      variable can be created using the option ``by``. For instance:
 |
 |      .. plot::
 |          :context: close-figs
 |
 |          >>> df = pd.DataFrame(np.random.randn(10, 2),
 |          ...                   columns=['Col1', 'Col2'])
 |          >>> df['X'] = pd.Series(['A', 'A', 'A', 'A', 'A',
 |          ...                      'B', 'B', 'B', 'B', 'B'])
 |          >>> boxplot = df.boxplot(by='X')
 |      
 |      A list of strings (i.e. ``['X', 'Y']``) can be passed to boxplot
 |      in order to group the data by combination of the variables in the x-axis:
 |      
 |      .. plot::
 |          :context: close-figs
 |      
 |          >>> df = pd.DataFrame(np.random.randn(10, 3),
 |          ...                   columns=['Col1', 'Col2', 'Col3'])
 |          >>> df['X'] = pd.Series(['A', 'A', 'A', 'A', 'A',
 |          ...                      'B', 'B', 'B', 'B', 'B'])
 |          >>> df['Y'] = pd.Series(['A', 'B', 'A', 'B', 'A',
 |          ...                      'B', 'A', 'B', 'A', 'B'])
 |          >>> boxplot = df.boxplot(column=['Col1', 'Col2'], by=['X', 'Y'])
 |      
 |      The layout of boxplot can be adjusted giving a tuple to ``layout``:
 |      
 |      .. plot::
 |          :context: close-figs
 |      
 |          >>> boxplot = df.boxplot(column=['Col1', 'Col2'], by='X',
 |          ...                      layout=(2, 1))
 |      
 |      Additional formatting can be done to the boxplot, like suppressing the grid
 |      (``grid=False``), rotating the labels in the x-axis (i.e. ``rot=45``)
 |      or changing the fontsize (i.e. ``fontsize=15``):
 |      
 |      .. plot::
 |          :context: close-figs
 |      
 |          >>> boxplot = df.boxplot(grid=False, rot=45, fontsize=15)
 |      
 |      The parameter ``return_type`` can be used to select the type of element
 |      returned by `boxplot`.  When ``return_type='axes'`` is selected,
 |      the matplotlib axes on which the boxplot is drawn are returned:
 |      
 |          >>> boxplot = df.boxplot(column=['Col1', 'Col2'], return_type='axes')
 |          >>> type(boxplot)
 |          <class 'matplotlib.axes._subplots.AxesSubplot'>
 |      
 |      When grouping with ``by``, a Series mapping columns to ``return_type``
 |      is returned:
 |      
 |          >>> boxplot = df.boxplot(column=['Col1', 'Col2'], by='X',
 |          ...                      return_type='axes')
 |          >>> type(boxplot)
 |          <class 'pandas.core.series.Series'>
 |      
 |      If ``return_type`` is `None`, a NumPy array of axes with the same shape
 |      as ``layout`` is returned:
 |      
 |          >>> boxplot = df.boxplot(column=['Col1', 'Col2'], by='X',
 |          ...                      return_type=None)
 |          >>> type(boxplot)
 |          <class 'numpy.ndarray'>
 |  
 |  clip(self: 'DataFrame', lower=None, upper=None, axis: 'Axis | None' = None, 
inplace: 'bool' = False, *args, **kwargs) -> 'DataFrame | None'
 |      Trim values at input threshold(s).
 |      
 |      Assigns values outside boundary to boundary values. Thresholds
 |      can be singular values or array like, and in the latter case
 |      the clipping is performed element-wise in the specified axis.
 |      
 |      Parameters
 |      ----------
 |      lower : float or array-like, default None
 |          Minimum threshold value. All values below this
 |          threshold will be set to it. A missing
 |          threshold (e.g `NA`) will not clip the value.
 |      upper : float or array-like, default None
 |          Maximum threshold value. All values above this
 |          threshold will be set to it. A missing
 |          threshold (e.g `NA`) will not clip the value.
 |      axis : int or str axis name, optional
 |          Align object with lower and upper along the given axis.
 |      inplace : bool, default False
 |          Whether to perform the operation in place on the data.
 |      *args, **kwargs
 |          Additional keywords have no effect but might be accepted
 |          for compatibility with numpy.
 |      
 |      Returns
 |      -------
 |      Series or DataFrame or None
 |          Same type as calling object with the values outside the
 |          clip boundaries replaced or None if ``inplace=True``.
 |      
 |      See Also
 |      --------
 |      Series.clip : Trim values at input threshold in series.
 |      DataFrame.clip : Trim values at input threshold in dataframe.
 |      numpy.clip : Clip (limit) the values in an array.
 |      
 |      Examples
 |      --------
 |      >>> data = {'col_0': [9, -3, 0, -1, 5], 'col_1': [-2, -7, 6, 8, -5]}
 |      >>> df = pd.DataFrame(data)
 |      >>> df
 |         col_0  col_1
 |      0      9     -2
 |      1     -3     -7
 |      2      0      6
 |      3     -1      8
 |      4      5     -5
 |      
 |      Clips per column using lower and upper thresholds:
 |      
 |      >>> df.clip(-4, 6)
 |         col_0  col_1
 |      0      6     -2
 |      1     -3     -4
 |      2      0      6
 |      3     -1      6
 |      4      5     -4
 |      
 |      Clips using specific lower and upper thresholds per column element:
 |      
 |      >>> t = pd.Series([2, -4, -1, 6, 3])
 |      >>> t
 |      0    2
 |      1   -4
 |      2   -1
 |      3    6
 |      4    3
 |      dtype: int64
 |      
 |      >>> df.clip(t, t + 4, axis=0)
 |         col_0  col_1
 |      0      6      2
 |      1     -3     -4
 |      2      0      3
 |      3      6      8
 |      4      5      3
 |      
 |      Clips using specific lower threshold per column element, with missing values:
 |      
 |      >>> t = pd.Series([2, -4, np.NaN, 6, 3])
 |      >>> t
 |      0    2.0
 |      1   -4.0
 |      2    NaN
 |      3    6.0
 |      4    3.0
 |      dtype: float64
 |      
 |      >>> df.clip(t, axis=0)
 |      col_0  col_1
 |      0      9      2
 |      1     -3     -4
 |      2      0      6
 |      3      6      8
 |      4      5      3
 |  
 |  combine(self, other: 'DataFrame', func, fill_value=None, overwrite: 'bool' =
True) -> 'DataFrame'
 |      Perform column-wise combine with another DataFrame.
 |      
 |      Combines a DataFrame with `other` DataFrame using `func`
 |      to element-wise combine columns. The row and column indexes of the
 |      resulting DataFrame will be the union of the two.
 |      
 |      Parameters
 |      ----------
 |      other : DataFrame
 |          The DataFrame to merge column-wise.
 |      func : function
 |          Function that takes two series as inputs and return a Series or a
 |          scalar. Used to merge the two dataframes column by columns.
 |      fill_value : scalar value, default None
 |          The value to fill NaNs with prior to passing any column to the
 |          merge func.
 |      overwrite : bool, default True
 |          If True, columns in `self` that do not exist in `other` will be
 |          overwritten with NaNs.
 |
 |      Returns
 |      -------
 |      DataFrame
 |          Combination of the provided DataFrames.
 |
 |      See Also
 |      --------
 |      DataFrame.combine_first : Combine two DataFrame objects and default to
 |          non-null values in frame calling the method.
 |
 |      Examples
 |      --------
 |      Combine using a simple function that chooses the smaller column.
 |
 |      >>> df1 = pd.DataFrame({'A': [0, 0], 'B': [4, 4]})
 |      >>> df2 = pd.DataFrame({'A': [1, 1], 'B': [3, 3]})
 |      >>> take_smaller = lambda s1, s2: s1 if s1.sum() < s2.sum() else s2
 |      >>> df1.combine(df2, take_smaller)
 |         A  B
 |      0  0  3
 |      1  0  3
 |
 |      Example using a true element-wise combine function.
 |
 |      >>> df1 = pd.DataFrame({'A': [5, 0], 'B': [2, 4]})
 |      >>> df2 = pd.DataFrame({'A': [1, 1], 'B': [3, 3]})
 |      >>> df1.combine(df2, np.minimum)
 |         A  B
 |      0  1  2
 |      1  0  3
 |
 |      Using `fill_value` fills Nones prior to passing the column to the
 |      merge function.
 |
 |      >>> df1 = pd.DataFrame({'A': [0, 0], 'B': [None, 4]})
 |      >>> df2 = pd.DataFrame({'A': [1, 1], 'B': [3, 3]})
 |      >>> df1.combine(df2, take_smaller, fill_value=-5)
 |         A    B
 |      0  0 -5.0
 |      1  0  4.0
 |
 |      However, if the same element in both dataframes is None, that None
 |      is preserved
 |
 |      >>> df1 = pd.DataFrame({'A': [0, 0], 'B': [None, 4]})
 |      >>> df2 = pd.DataFrame({'A': [1, 1], 'B': [None, 3]})
 |      >>> df1.combine(df2, take_smaller, fill_value=-5)
 |          A    B
 |      0  0 -5.0
 |      1  0  3.0
 |
 |      Example that demonstrates the use of `overwrite` and behavior when
 |      the axis differ between the dataframes.
 |
 |      >>> df1 = pd.DataFrame({'A': [0, 0], 'B': [4, 4]})
 |      >>> df2 = pd.DataFrame({'B': [3, 3], 'C': [-10, 1], }, index=[1, 2])
 |      >>> df1.combine(df2, take_smaller)
 |           A    B     C
 |      0  NaN  NaN   NaN
 |      1  NaN  3.0 -10.0
 |      2  NaN  3.0   1.0
 |
 |      >>> df1.combine(df2, take_smaller, overwrite=False)
 |           A    B     C
 |      0  0.0  NaN   NaN
 |      1  0.0  3.0 -10.0
 |      2  NaN  3.0   1.0
 |
 |      Demonstrating the preference of the passed in dataframe.
 |
 |      >>> df2 = pd.DataFrame({'B': [3, 3], 'C': [1, 1], }, index=[1, 2])
 |      >>> df2.combine(df1, take_smaller)
 |         A    B   C
 |      0  0.0  NaN NaN
 |      1  0.0  3.0 NaN
 |      2  NaN  3.0 NaN
 |
 |      >>> df2.combine(df1, take_smaller, overwrite=False)
 |           A    B   C
 |      0  0.0  NaN NaN
 |      1  0.0  3.0 1.0
 |      2  NaN  3.0 1.0
 |
 |  combine_first(self, other: 'DataFrame') -> 'DataFrame'
 |      Update null elements with value in the same location in `other`.
 |
 |      Combine two DataFrame objects by filling null values in one DataFrame
 |      with non-null values from other DataFrame. The row and column indexes
 |      of the resulting DataFrame will be the union of the two.
 |
 |      Parameters
 |      ----------
 |      other : DataFrame
 |          Provided DataFrame to use to fill null values.
 |
 |      Returns
 |      -------
 |      DataFrame
 |          The result of combining the provided DataFrame with the other object.
 |
 |      See Also
 |      --------
 |      DataFrame.combine : Perform series-wise operation on two DataFrames
 |          using a given function.
 |
 |      Examples
 |      --------
 |      >>> df1 = pd.DataFrame({'A': [None, 0], 'B': [None, 4]})
 |      >>> df2 = pd.DataFrame({'A': [1, 1], 'B': [3, 3]})
 |      >>> df1.combine_first(df2)
 |           A    B
 |      0  1.0  3.0
 |      1  0.0  4.0
 |
 |      Null values still persist if the location of that null value
 |      does not exist in `other`
 |
 |      >>> df1 = pd.DataFrame({'A': [None, 0], 'B': [4, None]})
 |      >>> df2 = pd.DataFrame({'B': [3, 3], 'C': [1, 1]}, index=[1, 2])
 |      >>> df1.combine_first(df2)
 |           A    B    C
 |      0  NaN  4.0  NaN
 |      1  0.0  3.0  1.0
 |      2  NaN  3.0  1.0
 |
 |  compare(self, other: 'DataFrame', align_axis: 'Axis' = 1, keep_shape: 'bool' =
False, keep_equal: 'bool' = False) -> 'DataFrame'
 |      Compare to another DataFrame and show the differences.
 |
 |      .. versionadded:: 1.1.0
 |
 |      Parameters
 |      ----------
 |      other : DataFrame
 |          Object to compare with.
 |
 |      align_axis : {0 or 'index', 1 or 'columns'}, default 1
 |          Determine which axis to align the comparison on.
 |
 |          * 0, or 'index' : Resulting differences are stacked vertically
 |              with rows drawn alternately from self and other.
 |          * 1, or 'columns' : Resulting differences are aligned horizontally
 |              with columns drawn alternately from self and other.
 |
 |      keep_shape : bool, default False
 |          If true, all rows and columns are kept.
 |          Otherwise, only the ones with different values are kept.
 |
 |      keep_equal : bool, default False
 |          If true, the result keeps values that are equal.
 |          Otherwise, equal values are shown as NaNs.
 |
 |      Returns
 |      -------
 |      DataFrame
 |          DataFrame that shows the differences stacked side by side.
 |
 |          The resulting index will be a MultiIndex with 'self' and 'other'
 |          stacked alternately at the inner level.
 |
 |      Raises
 |      ------
 |      ValueError
 |          When the two DataFrames don't have identical labels or shape.
 |
 |      See Also
 |      --------
 |      Series.compare : Compare with another Series and show differences.
 |      DataFrame.equals : Test whether two objects contain the same elements.
 |
 |      Notes
 |      -----
 |      Matching NaNs will not appear as a difference.
 |
 |      Can only compare identically-labeled
 |      (i.e. same shape, identical row and column labels) DataFrames
 |
 |      Examples
 |      --------
 |      >>> df = pd.DataFrame(
 |      ...     {
 |      ...         "col1": ["a", "a", "b", "b", "a"],
 |      ...         "col2": [1.0, 2.0, 3.0, np.nan, 5.0],
 |      ...         "col3": [1.0, 2.0, 3.0, 4.0, 5.0]
 |      ...     },
 |      ...     columns=["col1", "col2", "col3"],
 |      ... )
 |      >>> df
 |        col1  col2  col3
 |      0    a   1.0   1.0
 |      1    a   2.0   2.0
 |      2    b   3.0   3.0
 |      3    b   NaN   4.0
 |      4    a   5.0   5.0
 |
 |      >>> df2 = df.copy()
 |      >>> df2.loc[0, 'col1'] = 'c'
 |      >>> df2.loc[2, 'col3'] = 4.0
 |      >>> df2
 |        col1  col2  col3
 |      0    c   1.0   1.0
 |      1    a   2.0   2.0
 |      2    b   3.0   4.0
 |      3    b   NaN   4.0
 |      4    a   5.0   5.0
 |
 |      Align the differences on columns
 |
 |      >>> df.compare(df2)
 |        col1       col3
 |        self other self other
 |      0    a     c  NaN   NaN
 |      2  NaN   NaN  3.0   4.0
 |
 |      Stack the differences on rows
 |
 |      >>> df.compare(df2, align_axis=0)
 |              col1  col3
 |      0 self     a   NaN
 |        other    c   NaN
 |      2 self   NaN   3.0
 |        other  NaN   4.0
 |
 |      Keep the equal values
 |
 |      >>> df.compare(df2, keep_equal=True)
 |        col1       col3
 |        self other self other
 |      0    a     c  1.0   1.0
 |      2    b     b  3.0   4.0
 |
 |      Keep all original rows and columns
 |
 |      >>> df.compare(df2, keep_shape=True)
 |        col1       col2       col3
 |        self other self other self other
 |      0    a     c  NaN   NaN  NaN   NaN
 |      1  NaN   NaN  NaN   NaN  NaN   NaN
 |      2  NaN   NaN  NaN   NaN  3.0   4.0
 |      3  NaN   NaN  NaN   NaN  NaN   NaN
 |      4  NaN   NaN  NaN   NaN  NaN   NaN
 |
 |      Keep all original rows and columns and also all original values
 |
 |      >>> df.compare(df2, keep_shape=True, keep_equal=True)
 |        col1       col2       col3
 |        self other self other self other
 |      0    a     c  1.0   1.0  1.0   1.0
 |      1    a     a  2.0   2.0  2.0   2.0
 |      2    b     b  3.0   3.0  3.0   4.0
 |      3    b     b  NaN   NaN  4.0   4.0
 |      4    a     a  5.0   5.0  5.0   5.0
 |
 |  corr(self, method: 'str | Callable[[np.ndarray, np.ndarray], float]' = 'pearson',
min_periods: 'int' = 1) -> 'DataFrame'
 |      Compute pairwise correlation of columns, excluding NA/null values.
 |
 |      Parameters
 |      ----------
 |      method : {'pearson', 'kendall', 'spearman'} or callable
 |          Method of correlation:
 |      
 |          * pearson : standard correlation coefficient
 |          * kendall : Kendall Tau correlation coefficient
 |          * spearman : Spearman rank correlation
 |          * callable: callable with input two 1d ndarrays
 |              and returning a float. Note that the returned matrix from corr
 |              will have 1 along the diagonals and will be symmetric
 |              regardless of the callable's behavior.
 |      min_periods : int, optional
 |          Minimum number of observations required per pair of columns
 |          to have a valid result. Currently only available for Pearson
 |          and Spearman correlation.
 |      
 |      Returns
 |      -------
 |      DataFrame
 |          Correlation matrix.
 |      
 |      See Also
 |      --------
 |      DataFrame.corrwith : Compute pairwise correlation with another
 |          DataFrame or Series.
 |      Series.corr : Compute the correlation between two Series.
 |      
 |      Examples
 |      --------
 |      >>> def histogram_intersection(a, b):
 |      ...     v = np.minimum(a, b).sum().round(decimals=1)
 |      ...     return v
 |      >>> df = pd.DataFrame([(.2, .3), (.0, .6), (.6, .0), (.2, .1)],
 |      ...                   columns=['dogs', 'cats'])
 |      >>> df.corr(method=histogram_intersection)
 |            dogs  cats
 |      dogs   1.0   0.3
 |      cats   0.3   1.0
 |  
 |  corrwith(self, other, axis: 'Axis' = 0, drop=False, method='pearson') -> 'Series'
 |      Compute pairwise correlation.
 |      
 |      Pairwise correlation is computed between rows or columns of
 |      DataFrame with rows or columns of Series or DataFrame. DataFrames
 |      are first aligned along both axes before computing the
 |      correlations.
 |      
 |      Parameters
 |      ----------
 |      other : DataFrame, Series
 |          Object with which to compute correlations.
 |      axis : {0 or 'index', 1 or 'columns'}, default 0
 |          The axis to use. 0 or 'index' to compute column-wise, 1 or 'columns' for
 |          row-wise.
 |      drop : bool, default False
 |          Drop missing indices from result.
 |      method : {'pearson', 'kendall', 'spearman'} or callable
 |          Method of correlation:
 |      
 |          * pearson : standard correlation coefficient
 |          * kendall : Kendall Tau correlation coefficient
 |          * spearman : Spearman rank correlation
 |          * callable: callable with input two 1d ndarrays
 |              and returning a float.
 |      
 |      Returns
 |      -------
 |      Series
 |          Pairwise correlations.
 |      
 |      See Also
 |      --------
 |      DataFrame.corr : Compute pairwise correlation of columns.
 |  
 |  count(self, axis: 'Axis' = 0, level: 'Level | None' = None, numeric_only: 'bool'
= False)
 |      Count non-NA cells for each column or row.
 |      
 |      The values `None`, `NaN`, `NaT`, and optionally `numpy.inf` (depending
 |      on `pandas.options.mode.use_inf_as_na`) are considered NA.
 |      
 |      Parameters
 |      ----------
 |      axis : {0 or 'index', 1 or 'columns'}, default 0
 |          If 0 or 'index' counts are generated for each column.
 |          If 1 or 'columns' counts are generated for each row.
 |      level : int or str, optional
 |          If the axis is a `MultiIndex` (hierarchical), count along a
 |          particular `level`, collapsing into a `DataFrame`.
 |          A `str` specifies the level name.
 |      numeric_only : bool, default False
 |          Include only `float`, `int` or `boolean` data.
 |
 |      Returns
 |      -------
 |      Series or DataFrame
 |          For each column/row the number of non-NA/null entries.
 |          If `level` is specified returns a `DataFrame`.
 |
 |      See Also
 |      --------
 |      Series.count: Number of non-NA elements in a Series.
 |      DataFrame.value_counts: Count unique combinations of columns.
 |      DataFrame.shape: Number of DataFrame rows and columns (including NA
 |          elements).
 |      DataFrame.isna: Boolean same-sized DataFrame showing places of NA
 |          elements.
 |
 |      Examples
 |      --------
 |      Constructing DataFrame from a dictionary:
 |
 |      >>> df = pd.DataFrame({"Person":
 |      ...                    ["John", "Myla", "Lewis", "John", "Myla"],
 |      ...                    "Age": [24., np.nan, 21., 33, 26],
 |      ...                    "Single": [False, True, True, True, False]})
 |      >>> df
 |         Person   Age  Single
 |      0    John  24.0   False
 |      1    Myla   NaN    True
 |      2   Lewis  21.0    True
 |      3    John  33.0    True
 |      4    Myla  26.0   False
 |
 |      Notice the uncounted NA values:
 |
 |      >>> df.count()
 |      Person    5
 |      Age       4
 |      Single    5
 |      dtype: int64
 |
 |      Counts for each **row**:
 |
 |      >>> df.count(axis='columns')
 |      0    3
 |      1    2
 |      2    3
 |      3    3
 |      4    3
 |      dtype: int64
 |
 |  cov(self, min_periods: 'int | None' = None, ddof: 'int | None' = 1) ->
'DataFrame'
 |      Compute pairwise covariance of columns, excluding NA/null values.
 |
 |      Compute the pairwise covariance among the series of a DataFrame.
 |      The returned data frame is the `covariance matrix
 |      <https://en.wikipedia.org/wiki/Covariance_matrix>`__ of the columns
 |      of the DataFrame.
 |
 |      Both NA and null values are automatically excluded from the
 |      calculation. (See the note below about bias from missing values.)
 |      A threshold can be set for the minimum number of
 |      observations for each value created. Comparisons with observations
 |      below this threshold will be returned as ``NaN``.
 |
 |      This method is generally used for the analysis of time series data to
 |      understand the relationship between different measures
 |      across time.
 |
 |      Parameters
 |      ----------
 |      min_periods : int, optional
 |          Minimum number of observations required per pair of columns
 |          to have a valid result.
 |
 |      ddof : int, default 1
 |          Delta degrees of freedom.  The divisor used in calculations
 |          is ``N - ddof``, where ``N`` represents the number of elements.
 |
 |          .. versionadded:: 1.1.0
 |
 |      Returns
 |      -------
 |      DataFrame
 |          The covariance matrix of the series of the DataFrame.
 |      
 |      See Also
 |      --------
 |      Series.cov : Compute covariance with another Series.
 |      core.window.ExponentialMovingWindow.cov: Exponential weighted sample 
covariance.
 |      core.window.Expanding.cov : Expanding sample covariance.
 |      core.window.Rolling.cov : Rolling sample covariance.
 |      
 |      Notes
 |      -----
 |      Returns the covariance matrix of the DataFrame's time series.
 |      The covariance is normalized by N-ddof.
 |      
 |      For DataFrames that have Series that are missing data (assuming that
 |      data is `missing at random
 |      <https://en.wikipedia.org/wiki/Missing_data#Missing_at_random>`__)
 |      the returned covariance matrix will be an unbiased estimate
 |      of the variance and covariance between the member Series.
 |      
 |      However, for many applications this estimate may not be acceptable
 |      because the estimate covariance matrix is not guaranteed to be positive
 |      semi-definite. This could lead to estimate correlations having
 |      absolute values which are greater than one, and/or a non-invertible
 |      covariance matrix. See `Estimation of covariance matrices
 |      <https://en.wikipedia.org/w/index.php?title=Estimation_of_covariance_
 |      matrices>`__ for more details.
 |      
 |      Examples
 |      --------
 |      >>> df = pd.DataFrame([(1, 2), (0, 3), (2, 0), (1, 1)],
 |      ...                   columns=['dogs', 'cats'])
 |      >>> df.cov()
 |                dogs      cats
 |      dogs  0.666667 -1.000000
 |      cats -1.000000  1.666667
 |      
 |      >>> np.random.seed(42)
 |      >>> df = pd.DataFrame(np.random.randn(1000, 5),
 |      ...                   columns=['a', 'b', 'c', 'd', 'e'])
 |      >>> df.cov()
 |                a         b         c         d         e
 |      a  0.998438 -0.020161  0.059277 -0.008943  0.014144
 |      b -0.020161  1.059352 -0.008543 -0.024738  0.009826
 |      c  0.059277 -0.008543  1.010670 -0.001486 -0.000271
 |      d -0.008943 -0.024738 -0.001486  0.921297 -0.013692
 |      e  0.014144  0.009826 -0.000271 -0.013692  0.977795
 |      
 |      **Minimum number of periods**
 |      
 |      This method also supports an optional ``min_periods`` keyword
 |      that specifies the required minimum number of non-NA observations for
 |      each column pair in order to have a valid result:
 |      
 |      >>> np.random.seed(42)
 |      >>> df = pd.DataFrame(np.random.randn(20, 3),
 |      ...                   columns=['a', 'b', 'c'])
 |      >>> df.loc[df.index[:5], 'a'] = np.nan
 |      >>> df.loc[df.index[5:10], 'b'] = np.nan
 |      >>> df.cov(min_periods=12)
 |                a         b         c
 |      a  0.316741       NaN -0.150812
 |      b       NaN  1.248003  0.191417
 |      c -0.150812  0.191417  0.895202
 |  
 |  cummax(self, axis=None, skipna=True, *args, **kwargs)
 |      Return cumulative maximum over a DataFrame or Series axis.
 |      
 |      Returns a DataFrame or Series of the same size containing the cumulative
 |      maximum.
 |      
 |      Parameters
 |      ----------
 |      axis : {0 or 'index', 1 or 'columns'}, default 0
 |          The index or the name of the axis. 0 is equivalent to None or 'index'.
 |      skipna : bool, default True
 |          Exclude NA/null values. If an entire row/column is NA, the result
 |          will be NA.
 |      *args, **kwargs
 |          Additional keywords have no effect but might be accepted for
 |          compatibility with NumPy.
 |      
 |      Returns
 |      -------
 |      Series or DataFrame
 |          Return cumulative maximum of Series or DataFrame.
 |      
 |      See Also
 |      --------
 |      core.window.Expanding.max : Similar functionality
 |          but ignores ``NaN`` values.
 |      DataFrame.max : Return the maximum over
 |          DataFrame axis.
 |      DataFrame.cummax : Return cumulative maximum over DataFrame axis.
 |      DataFrame.cummin : Return cumulative minimum over DataFrame axis.
 |      DataFrame.cumsum : Return cumulative sum over DataFrame axis.
 |      DataFrame.cumprod : Return cumulative product over DataFrame axis.
 |      
 |      Examples
 |      --------
 |      **Series**
 |      
 |      >>> s = pd.Series([2, np.nan, 5, -1, 0])
 |      >>> s
 |      0    2.0
 |      1    NaN
 |      2    5.0
 |      3   -1.0
 |      4    0.0
 |      dtype: float64
 |      
 |      By default, NA values are ignored.
 |      
 |      >>> s.cummax()
 |      0    2.0
 |      1    NaN
 |      2    5.0
 |      3    5.0
 |      4    5.0
 |      dtype: float64
 |      
 |      To include NA values in the operation, use ``skipna=False``
 |      
 |      >>> s.cummax(skipna=False)
 |      0    2.0
 |      1    NaN
 |      2    NaN
 |      3    NaN
 |      4    NaN
 |      dtype: float64
 |      
 |      **DataFrame**
 |      
 |      >>> df = pd.DataFrame([[2.0, 1.0],
 |      ...                    [3.0, np.nan],
 |      ...                    [1.0, 0.0]],
 |      ...                    columns=list('AB'))
 |      >>> df
 |           A    B
 |      0  2.0  1.0
 |      1  3.0  NaN
 |      2  1.0  0.0
 |      
 |      By default, iterates over rows and finds the maximum
 |      in each column. This is equivalent to ``axis=None`` or ``axis='index'``.
 |      
 |      >>> df.cummax()
 |           A    B
 |      0  2.0  1.0
 |      1  3.0  NaN
 |      2  3.0  1.0
 |      
 |      To iterate over columns and find the maximum in each row,
 |      use ``axis=1``
 |      
 |      >>> df.cummax(axis=1)
 |           A    B
 |      0  2.0  2.0
 |      1  3.0  NaN
 |      2  1.0  1.0
 |  
 |  cummin(self, axis=None, skipna=True, *args, **kwargs)
 |      Return cumulative minimum over a DataFrame or Series axis.
 |      
 |      Returns a DataFrame or Series of the same size containing the cumulative
 |      minimum.
 |      
 |      Parameters
 |      ----------
 |      axis : {0 or 'index', 1 or 'columns'}, default 0
 |          The index or the name of the axis. 0 is equivalent to None or 'index'.
 |      skipna : bool, default True
 |          Exclude NA/null values. If an entire row/column is NA, the result
 |          will be NA.
 |      *args, **kwargs
 |          Additional keywords have no effect but might be accepted for
 |          compatibility with NumPy.
 |
 |      Returns
 |      -------
 |      Series or DataFrame
 |          Return cumulative minimum of Series or DataFrame.
 |
 |      See Also
 |      --------
 |      core.window.Expanding.min : Similar functionality
 |          but ignores ``NaN`` values.
 |      DataFrame.min : Return the minimum over
 |          DataFrame axis.
 |      DataFrame.cummax : Return cumulative maximum over DataFrame axis.
 |      DataFrame.cummin : Return cumulative minimum over DataFrame axis.
 |      DataFrame.cumsum : Return cumulative sum over DataFrame axis.
 |      DataFrame.cumprod : Return cumulative product over DataFrame axis.
 |
 |      Examples
 |      --------
 |      **Series**
 |
 |      >>> s = pd.Series([2, np.nan, 5, -1, 0])
 |      >>> s
 |      0    2.0
 |      1    NaN
 |      2    5.0
 |      3   -1.0
 |      4    0.0
 |      dtype: float64
 |
 |      By default, NA values are ignored.
 |
 |      >>> s.cummin()
 |      0    2.0
 |      1    NaN
 |      2    2.0
 |      3   -1.0
 |      4   -1.0
 |      dtype: float64
 |
 |      To include NA values in the operation, use ``skipna=False``
 |
 |      >>> s.cummin(skipna=False)
 |      0    2.0
 |      1    NaN
 |      2    NaN
 |      3    NaN
 |      4    NaN
 |      dtype: float64
 |
 |      **DataFrame**
 |
 |      >>> df = pd.DataFrame([[2.0, 1.0],
 |      ...                    [3.0, np.nan],
 |      ...                    [1.0, 0.0]],
 |      ...                    columns=list('AB'))
 |      >>> df
 |           A    B
 |      0  2.0  1.0
 |      1  3.0  NaN
 |      2  1.0  0.0
 |
 |      By default, iterates over rows and finds the minimum
 |      in each column. This is equivalent to ``axis=None`` or ``axis='index'``.
 |
 |      >>> df.cummin()
 |           A    B
 |      0  2.0  1.0
 |      1  2.0  NaN
 |      2  1.0  0.0
 |
 |      To iterate over columns and find the minimum in each row,
 |      use ``axis=1``
 |
 |      >>> df.cummin(axis=1)
 |           A    B
 |      0  2.0  1.0
 |      1  3.0  NaN
 |      2  1.0  0.0
 |
 |  cumprod(self, axis=None, skipna=True, *args, **kwargs)
 |      Return cumulative product over a DataFrame or Series axis.
 |
 |      Returns a DataFrame or Series of the same size containing the cumulative
 |      product.
 |
 |      Parameters
 |      ----------
 |      axis : {0 or 'index', 1 or 'columns'}, default 0
 |          The index or the name of the axis. 0 is equivalent to None or 'index'.
 |      skipna : bool, default True
 |          Exclude NA/null values. If an entire row/column is NA, the result
 |          will be NA.
 |      *args, **kwargs
 |          Additional keywords have no effect but might be accepted for
 |          compatibility with NumPy.
 |
 |      Returns
 |      -------
 |      Series or DataFrame
 |          Return cumulative product of Series or DataFrame.
 |
 |      See Also
 |      --------
 |      core.window.Expanding.prod : Similar functionality
 |          but ignores ``NaN`` values.
 |      DataFrame.prod : Return the product over
 |          DataFrame axis.
 |      DataFrame.cummax : Return cumulative maximum over DataFrame axis.
 |      DataFrame.cummin : Return cumulative minimum over DataFrame axis.
 |      DataFrame.cumsum : Return cumulative sum over DataFrame axis.
 |      DataFrame.cumprod : Return cumulative product over DataFrame axis.
 |
 |      Examples
 |      --------
 |      **Series**
 |
 |      >>> s = pd.Series([2, np.nan, 5, -1, 0])
 |      >>> s
 |      0    2.0
 |      1    NaN
 |      2    5.0
 |      3   -1.0
 |      4    0.0
 |      dtype: float64
 |
 |      By default, NA values are ignored.
 |
 |      >>> s.cumprod()
 |      0     2.0
 |      1     NaN
 |      2    10.0
 |      3   -10.0
 |      4    -0.0
 |      dtype: float64
 |
 |      To include NA values in the operation, use ``skipna=False``
 |
 |      >>> s.cumprod(skipna=False)
 |      0    2.0
 |      1    NaN
 |      2    NaN
 |      3    NaN
 |      4    NaN
 |      dtype: float64
 |
 |      **DataFrame**
 |
 |      >>> df = pd.DataFrame([[2.0, 1.0],
 |      ...                    [3.0, np.nan],
 |      ...                    [1.0, 0.0]],
 |      ...                    columns=list('AB'))
 |      >>> df
 |           A    B
 |      0  2.0  1.0
 |      1  3.0  NaN
 |      2  1.0  0.0
 |
 |      By default, iterates over rows and finds the product
 |      in each column. This is equivalent to ``axis=None`` or ``axis='index'``.
 |
 |      >>> df.cumprod()
 |           A    B
 |      0  2.0  1.0
 |      1  6.0  NaN
 |      2  6.0  0.0
 |
 |      To iterate over columns and find the product in each row,
 |      use ``axis=1``
 |      
 |      >>> df.cumprod(axis=1)
 |           A    B
 |      0  2.0  2.0
 |      1  3.0  NaN
 |      2  1.0  0.0
 |  
 |  cumsum(self, axis=None, skipna=True, *args, **kwargs)
 |      Return cumulative sum over a DataFrame or Series axis.
 |      
 |      Returns a DataFrame or Series of the same size containing the cumulative
 |      sum.
 |      
 |      Parameters
 |      ----------
 |      axis : {0 or 'index', 1 or 'columns'}, default 0
 |          The index or the name of the axis. 0 is equivalent to None or 'index'.
 |      skipna : bool, default True
 |          Exclude NA/null values. If an entire row/column is NA, the result
 |          will be NA.
 |      *args, **kwargs
 |          Additional keywords have no effect but might be accepted for
 |          compatibility with NumPy.
 |      
 |      Returns
 |      -------
 |      Series or DataFrame
 |          Return cumulative sum of Series or DataFrame.
 |      
 |      See Also
 |      --------
 |      core.window.Expanding.sum : Similar functionality
 |          but ignores ``NaN`` values.
 |      DataFrame.sum : Return the sum over
 |          DataFrame axis.
 |      DataFrame.cummax : Return cumulative maximum over DataFrame axis.
 |      DataFrame.cummin : Return cumulative minimum over DataFrame axis.
 |      DataFrame.cumsum : Return cumulative sum over DataFrame axis.
 |      DataFrame.cumprod : Return cumulative product over DataFrame axis.
 |      
 |      Examples
 |      --------
 |      **Series**
 |      
 |      >>> s = pd.Series([2, np.nan, 5, -1, 0])
 |      >>> s
 |      0    2.0
 |      1    NaN
 |      2    5.0
 |      3   -1.0
 |      4    0.0
 |      dtype: float64
 |      
 |      By default, NA values are ignored.
 |      
 |      >>> s.cumsum()
 |      0    2.0
 |      1    NaN
 |      2    7.0
 |      3    6.0
 |      4    6.0
 |      dtype: float64
 |      
 |      To include NA values in the operation, use ``skipna=False``
 |      
 |      >>> s.cumsum(skipna=False)
 |      0    2.0
 |      1    NaN
 |      2    NaN
 |      3    NaN
 |      4    NaN
 |      dtype: float64
 |      
 |      **DataFrame**
 |      
 |      >>> df = pd.DataFrame([[2.0, 1.0],
 |      ...                    [3.0, np.nan],
 |      ...                    [1.0, 0.0]],
 |      ...                    columns=list('AB'))
 |      >>> df
 |           A    B
 |      0  2.0  1.0
 |      1  3.0  NaN
 |      2  1.0  0.0
 |      
 |      By default, iterates over rows and finds the sum
 |      in each column. This is equivalent to ``axis=None`` or ``axis='index'``.
 |
 |      >>> df.cumsum()
 |           A    B
 |      0  2.0  1.0
 |      1  5.0  NaN
 |      2  6.0  1.0
 |
 |      To iterate over columns and find the sum in each row,
 |      use ``axis=1``
 |
 |      >>> df.cumsum(axis=1)
 |           A    B
 |      0  2.0  3.0
 |      1  3.0  NaN
 |      2  1.0  1.0
 |
 |  diff(self, periods: 'int' = 1, axis: 'Axis' = 0) -> 'DataFrame'
 |      First discrete difference of element.
 |
 |      Calculates the difference of a Dataframe element compared with another
 |      element in the Dataframe (default is element in previous row).
 |
 |      Parameters
 |      ----------
 |      periods : int, default 1
 |          Periods to shift for calculating difference, accepts negative
 |          values.
 |      axis : {0 or 'index', 1 or 'columns'}, default 0
 |          Take difference over rows (0) or columns (1).
 |
 |      Returns
 |      -------
 |      Dataframe
 |          First differences of the Series.
 |
 |      See Also
 |      --------
 |      Dataframe.pct_change: Percent change over given number of periods.
 |      Dataframe.shift: Shift index by desired number of periods with an
 |          optional time freq.
 |      Series.diff: First discrete difference of object.
 |
 |      Notes
 |      -----
 |      For boolean dtypes, this uses :meth:`operator.xor` rather than
 |      :meth:`operator.sub`.
 |      The result is calculated according to current dtype in Dataframe,
 |      however dtype of the result is always float64.
 |
 |      Examples
 |      --------
 |
 |      Difference with previous row
 |
 |      >>> df = pd.DataFrame({'a': [1, 2, 3, 4, 5, 6],
 |      ...                    'b': [1, 1, 2, 3, 5, 8],
 |      ...                    'c': [1, 4, 9, 16, 25, 36]})
 |      >>> df
 |         a  b   c
 |      0  1  1   1
 |      1  2  1   4
 |      2  3  2   9
 |      3  4  3  16
 |      4  5  5  25
 |      5  6  8  36
 |
 |      >>> df.diff()
 |           a    b     c
 |      0  NaN  NaN   NaN
 |      1  1.0  0.0   3.0
 |      2  1.0  1.0   5.0
 |      3  1.0  1.0   7.0
 |      4  1.0  2.0   9.0
 |      5  1.0  3.0  11.0
 |
 |      Difference with previous column
 |
 |      >>> df.diff(axis=1)
 |          a  b   c
 |      0 NaN  0   0
 |      1 NaN -1   3
 |      2 NaN -1   7
 |      3 NaN -1  13
 |      4 NaN  0  20
 |      5 NaN  2  28
```

```
|
|      Difference with 3rd previous row
|
|      >>> df.diff(periods=3)
|           a    b     c
|      0  NaN  NaN   NaN
|      1  NaN  NaN   NaN
|      2  NaN  NaN   NaN
|      3  3.0  2.0  15.0
|      4  3.0  4.0  21.0
|      5  3.0  6.0  27.0
|
|      Difference with following row
|
|      >>> df.diff(periods=-1)
|           a    b     c
|      0 -1.0  0.0  -3.0
|      1 -1.0 -1.0  -5.0
|      2 -1.0 -1.0  -7.0
|      3 -1.0 -2.0  -9.0
|      4 -1.0 -3.0 -11.0
|      5  NaN  NaN   NaN
|
|      Overflow in input dtype
|
|      >>> df = pd.DataFrame({'a': [1, 0]}, dtype=np.uint8)
|      >>> df.diff()
|             a
|      0    NaN
|      1  255.0
|
|  div = truediv(self, other, axis='columns', level=None, fill_value=None)
|
|  divide = truediv(self, other, axis='columns', level=None, fill_value=None)
|
|  dot(self, other: 'AnyArrayLike | FrameOrSeriesUnion') -> 'FrameOrSeriesUnion'
|      Compute the matrix multiplication between the DataFrame and other.
|
|      This method computes the matrix product between the DataFrame and the
|      values of an other Series, DataFrame or a numpy array.
|
|      It can also be called using ``self @ other`` in Python >= 3.5.
|
|      Parameters
|      ----------
|      other : Series, DataFrame or array-like
|          The other object to compute the matrix product with.
|
|      Returns
|      -------
|      Series or DataFrame
|          If other is a Series, return the matrix product between self and
|          other as a Series. If other is a DataFrame or a numpy.array, return
|          the matrix product of self and other in a DataFrame of a np.array.
|
|      See Also
|      --------
|      Series.dot: Similar method for Series.
|
|      Notes
|      -----
|      The dimensions of DataFrame and other must be compatible in order to
|      compute the matrix multiplication. In addition, the column names of
|      DataFrame and the index of other must contain the same values, as they
|      will be aligned prior to the multiplication.
|
|      The dot method for Series computes the inner product, instead of the
|      matrix product here.
|
|      Examples
|      --------
|      Here we multiply a DataFrame with a Series.
|
|      >>> df = pd.DataFrame([[0, 1, -2, -1], [1, 1, 1, 1]])
|      >>> s = pd.Series([1, 1, 2, 1])
|      >>> df.dot(s)
|      0    -4
|      1     5
|      dtype: int64
|
|      Here we multiply a DataFrame with another DataFrame.
|
|      >>> other = pd.DataFrame([[0, 1], [1, 2], [-1, -1], [2, 0]])
|      >>> df.dot(other)
|          0   1
|      0   1   4
|      1   2   2
```

```
 |
 |      Note that the dot method give the same result as @
 |
 |      >>> df @ other
 |          0   1
 |      0   1   4
 |      1   2   2
 |
 |      The dot method works also if other is an np.array.
 |
 |      >>> arr = np.array([[0, 1], [1, 2], [-1, -1], [2, 0]])
 |      >>> df.dot(arr)
 |          0   1
 |      0   1   4
 |      1   2   2
 |
 |      Note how shuffling of the objects does not change the result.
 |
 |      >>> s2 = s.reindex([1, 0, 2, 3])
 |      >>> df.dot(s2)
 |      0    -4
 |      1     5
 |      dtype: int64
 |
 |  drop(self, labels=None, axis: 'Axis' = 0, index=None, columns=None, level: 'Level
| None' = None, inplace: 'bool' = False, errors: 'str' = 'raise')
 |      Drop specified labels from rows or columns.
 |
 |      Remove rows or columns by specifying label names and corresponding
 |      axis, or by specifying directly index or column names. When using a
 |      multi-index, labels on different levels can be removed by specifying
 |      the level. See the `user guide <advanced.shown_levels>`
 |      for more information about the now unused levels.
 |
 |      Parameters
 |      ----------
 |      labels : single label or list-like
 |          Index or column labels to drop.
 |      axis : {0 or 'index', 1 or 'columns'}, default 0
 |          Whether to drop labels from the index (0 or 'index') or
 |          columns (1 or 'columns').
 |      index : single label or list-like
 |          Alternative to specifying axis (``labels, axis=0``
 |          is equivalent to ``index=labels``).
 |      columns : single label or list-like
 |          Alternative to specifying axis (``labels, axis=1``
 |          is equivalent to ``columns=labels``).
 |      level : int or level name, optional
 |          For MultiIndex, level from which the labels will be removed.
 |      inplace : bool, default False
 |          If False, return a copy. Otherwise, do operation
 |          inplace and return None.
 |      errors : {'ignore', 'raise'}, default 'raise'
 |          If 'ignore', suppress error and only existing labels are
 |          dropped.
 |
 |      Returns
 |      -------
 |      DataFrame or None
 |          DataFrame without the removed index or column labels or
 |          None if ``inplace=True``.
 |
 |      Raises
 |      ------
 |      KeyError
 |          If any of the labels is not found in the selected axis.
 |
 |      See Also
 |      --------
 |      DataFrame.loc : Label-location based indexer for selection by label.
 |      DataFrame.dropna : Return DataFrame with labels on given axis omitted
 |          where (all or any) data are missing.
 |      DataFrame.drop_duplicates : Return DataFrame with duplicate rows
 |          removed, optionally only considering certain columns.
 |      Series.drop : Return Series with specified index labels removed.
 |
 |      Examples
 |      --------
 |      >>> df = pd.DataFrame(np.arange(12).reshape(3, 4),
 |      ...                   columns=['A', 'B', 'C', 'D'])
 |      >>> df
 |         A  B   C   D
 |      0  0  1   2   3
 |      1  4  5   6   7
 |      2  8  9  10  11
 |
 |      Drop columns
 |
 |      >>> df.drop(['B', 'C'], axis=1)
 |         A   D
 |      0  0   3
 |      1  4   7
 |      2  8  11
 |
 |      >>> df.drop(columns=['B', 'C'])
 |         A   D
 |      0  0   3
 |      1  4   7
 |      2  8  11
 |
 |      Drop a row by index
 |
 |      >>> df.drop([0, 1])
 |         A  B   C   D
 |      2  8  9  10  11
 |
 |      Drop columns and/or rows of MultiIndex DataFrame
 |
 |      >>> midx = pd.MultiIndex(levels=[['lama', 'cow', 'falcon'],
 |      ...                              ['speed', 'weight', 'length']],
 |      ...                      codes=[[0, 0, 0, 1, 1, 1, 2, 2, 2],
 |      ...                             [0, 1, 2, 0, 1, 2, 0, 1, 2]])
 |      >>> df = pd.DataFrame(index=midx, columns=['big', 'small'],
 |      ...                   data=[[45, 30], [200, 100], [1.5, 1], [30, 20],
 |      ...                         [250, 150], [1.5, 0.8], [320, 250],
 |      ...                         [1, 0.8], [0.3, 0.2]])
 |      >>> df
 |                      big     small
 |      lama    speed   45.0    30.0
 |              weight  200.0   100.0
 |              length  1.5     1.0
 |      cow     speed   30.0    20.0
 |              weight  250.0   150.0
 |              length  1.5     0.8
 |      falcon  speed   320.0   250.0
 |              weight  1.0     0.8
 |              length  0.3     0.2
 |
 |      >>> df.drop(index='cow', columns='small')
 |                      big
 |      lama    speed   45.0
 |              weight  200.0
 |              length  1.5
 |      falcon  speed   320.0
 |              weight  1.0
 |              length  0.3
 |
 |      >>> df.drop(index='length', level=1)
 |                      big     small
 |      lama    speed   45.0    30.0
 |              weight  200.0   100.0
 |      cow     speed   30.0    20.0
 |              weight  250.0   150.0
 |      falcon  speed   320.0   250.0
 |              weight  1.0     0.8
 |
 |  drop_duplicates(self, subset: 'Hashable | Sequence[Hashable] | None' = None,
keep: "Literal['first'] | Literal['last'] | Literal[False]" = 'first', inplace:
'bool' = False, ignore_index: 'bool' = False) -> 'DataFrame | None'
 |      Return DataFrame with duplicate rows removed.
 |
 |      Considering certain columns is optional. Indexes, including time indexes
 |      are ignored.
 |
 |      Parameters
 |      ----------
 |      subset : column label or sequence of labels, optional
 |          Only consider certain columns for identifying duplicates, by
 |          default use all of the columns.
 |      keep : {'first', 'last', False}, default 'first'
 |          Determines which duplicates (if any) to keep.
 |          - ``first`` : Drop duplicates except for the first occurrence.
 |          - ``last`` : Drop duplicates except for the last occurrence.
 |          - False : Drop all duplicates.
 |      inplace : bool, default False
 |          Whether to drop duplicates in place or to return a copy.
 |      ignore_index : bool, default False
 |          If True, the resulting axis will be labeled 0, 1, …, n - 1.
 |
 |          .. versionadded:: 1.0.0
 |
 |      Returns
 |      -------
 |      DataFrame or None
 |          DataFrame with duplicates removed or None if ``inplace=True``.
 |      
 |      See Also
 |      --------
 |      DataFrame.value_counts: Count unique combinations of columns.
 |      
 |      Examples
 |      --------
 |      Consider dataset containing ramen rating.
 |      
 |      >>> df = pd.DataFrame({
 |      ...     'brand': ['Yum Yum', 'Yum Yum', 'Indomie', 'Indomie', 'Indomie'],
 |      ...     'style': ['cup', 'cup', 'cup', 'pack', 'pack'],
 |      ...     'rating': [4, 4, 3.5, 15, 5]
 |      ... })
 |      >>> df
 |          brand style  rating
 |      0  Yum Yum   cup     4.0
 |      1  Yum Yum   cup     4.0
 |      2  Indomie   cup     3.5
 |      3  Indomie  pack    15.0
 |      4  Indomie  pack     5.0
 |      
 |      By default, it removes duplicate rows based on all columns.
 |      
 |      >>> df.drop_duplicates()
 |          brand style  rating
 |      0  Yum Yum   cup     4.0
 |      2  Indomie   cup     3.5
 |      3  Indomie  pack    15.0
 |      4  Indomie  pack     5.0
 |      
 |      To remove duplicates on specific column(s), use ``subset``.
 |      
 |      >>> df.drop_duplicates(subset=['brand'])
 |          brand style  rating
 |      0  Yum Yum   cup     4.0
 |      2  Indomie   cup     3.5
 |      
 |      To remove duplicates and keep last occurrences, use ``keep``.
 |      
 |      >>> df.drop_duplicates(subset=['brand', 'style'], keep='last')
 |          brand style  rating
 |      1  Yum Yum   cup     4.0
 |      2  Indomie   cup     3.5
 |      4  Indomie  pack     5.0
 |  
 |  dropna(self, axis: 'Axis' = 0, how: 'str' = 'any', thresh=None, subset=None,
inplace: 'bool' = False)
 |      Remove missing values.
 |      
 |      See the :ref:`User Guide <missing_data>` for more on which values are
 |      considered missing, and how to work with missing data.
 |      
 |      Parameters
 |      ----------
 |      axis : {0 or 'index', 1 or 'columns'}, default 0
 |          Determine if rows or columns which contain missing values are
 |          removed.
 |      
 |          * 0, or 'index' : Drop rows which contain missing values.
 |          * 1, or 'columns' : Drop columns which contain missing value.
 |      
 |          .. versionchanged:: 1.0.0
 |      
 |             Pass tuple or list to drop on multiple axes.
 |             Only a single axis is allowed.
 |      
 |      how : {'any', 'all'}, default 'any'
 |          Determine if row or column is removed from DataFrame, when we have
 |          at least one NA or all NA.
 |      
 |          * 'any' : If any NA values are present, drop that row or column.
 |          * 'all' : If all values are NA, drop that row or column.
 |      
 |      thresh : int, optional
 |          Require that many non-NA values.
 |      subset : array-like, optional
 |          Labels along other axis to consider, e.g. if you are dropping rows
 |          these would be a list of columns to include.
 |      inplace : bool, default False
 |          If True, do operation inplace and return None.
 |      
 |      Returns
 |      -------
 |      DataFrame or None
 |          DataFrame with NA entries dropped from it or None if ``inplace=True``.
 |      
 |      See Also
 |      --------
 |      DataFrame.isna: Indicate missing values.
 |      DataFrame.notna : Indicate existing (non-missing) values.
 |      DataFrame.fillna : Replace missing values.
 |      Series.dropna : Drop missing values.
 |      Index.dropna : Drop missing indices.
 |      
 |      Examples
 |      --------
 |      >>> df = pd.DataFrame({"name": ['Alfred', 'Batman', 'Catwoman'],
 |      ...                    "toy": [np.nan, 'Batmobile', 'Bullwhip'],
 |      ...                    "born": [pd.NaT, pd.Timestamp("1940-04-25"),
 |      ...                             pd.NaT]})
 |      >>> df
 |             name        toy       born
 |      0    Alfred        NaN        NaT
 |      1    Batman  Batmobile 1940-04-25
 |      2  Catwoman   Bullwhip        NaT
 |      
 |      Drop the rows where at least one element is missing.
 |      
 |      >>> df.dropna()
 |           name        toy       born
 |      1  Batman  Batmobile 1940-04-25
 |      
 |      Drop the columns where at least one element is missing.
 |      
 |      >>> df.dropna(axis='columns')
 |             name
 |      0    Alfred
 |      1    Batman
 |      2  Catwoman
 |      
 |      Drop the rows where all elements are missing.
 |      
 |      >>> df.dropna(how='all')
 |             name        toy       born
 |      0    Alfred        NaN        NaT
 |      1    Batman  Batmobile 1940-04-25
 |      2  Catwoman   Bullwhip        NaT
 |      
 |      Keep only the rows with at least 2 non-NA values.
 |      
 |      >>> df.dropna(thresh=2)
 |             name        toy       born
 |      1    Batman  Batmobile 1940-04-25
 |      2  Catwoman   Bullwhip        NaT
 |      
 |      Define in which columns to look for missing values.
 |      
 |      >>> df.dropna(subset=['name', 'toy'])
 |             name        toy       born
 |      1    Batman  Batmobile 1940-04-25
 |      2  Catwoman   Bullwhip        NaT
 |      
 |      Keep the DataFrame with valid entries in the same variable.
 |      
 |      >>> df.dropna(inplace=True)
 |      >>> df
 |           name        toy       born
 |      1  Batman  Batmobile 1940-04-25
 |  
 |  duplicated(self, subset: 'Hashable | Sequence[Hashable] | None' = None, keep:
"Literal['first'] | Literal['last'] | Literal[False]" = 'first') -> 'Series'
 |      Return boolean Series denoting duplicate rows.
 |      
 |      Considering certain columns is optional.
 |      
 |      Parameters
 |      ----------
 |      subset : column label or sequence of labels, optional
 |          Only consider certain columns for identifying duplicates, by
 |          default use all of the columns.
 |      keep : {'first', 'last', False}, default 'first'
 |          Determines which duplicates (if any) to mark.
 |      
 |          - ``first`` : Mark duplicates as ``True`` except for the first
occurrence.
 |          - ``last`` : Mark duplicates as ``True`` except for the last occurrence.
 |          - False : Mark all duplicates as ``True``.
 |      
 |      Returns
 |      -------
 |      Series
 |          Boolean series for each duplicated rows.
 |  
 |      See Also
 |      --------
 |      Index.duplicated : Equivalent method on index.
 |      Series.duplicated : Equivalent method on Series.
 |      Series.drop_duplicates : Remove duplicate values from Series.
 |      DataFrame.drop_duplicates : Remove duplicate values from DataFrame.
 |  
 |      Examples
 |      --------
 |      Consider dataset containing ramen rating.
 |  
 |      >>> df = pd.DataFrame({
 |      ...     'brand': ['Yum Yum', 'Yum Yum', 'Indomie', 'Indomie', 'Indomie'],
 |      ...     'style': ['cup', 'cup', 'cup', 'pack', 'pack'],
 |      ...     'rating': [4, 4, 3.5, 15, 5]
 |      ... })
 |      >>> df
 |          brand style  rating
 |      0  Yum Yum   cup     4.0
 |      1  Yum Yum   cup     4.0
 |      2  Indomie   cup     3.5
 |      3  Indomie  pack    15.0
 |      4  Indomie  pack     5.0
 |  
 |      By default, for each set of duplicated values, the first occurrence
 |      is set on False and all others on True.
 |  
 |      >>> df.duplicated()
 |      0    False
 |      1     True
 |      2    False
 |      3    False
 |      4    False
 |      dtype: bool
 |  
 |      By using 'last', the last occurrence of each set of duplicated values
 |      is set on False and all others on True.
 |  
 |      >>> df.duplicated(keep='last')
 |      0     True
 |      1    False
 |      2    False
 |      3    False
 |      4    False
 |      dtype: bool
 |  
 |      By setting ``keep`` on False, all duplicates are True.
 |  
 |      >>> df.duplicated(keep=False)
 |      0     True
 |      1     True
 |      2    False
 |      3    False
 |      4    False
 |      dtype: bool
 |  
 |      To find duplicates on specific column(s), use ``subset``.
 |  
 |      >>> df.duplicated(subset=['brand'])
 |      0    False
 |      1     True
 |      2    False
 |      3     True
 |      4     True
 |      dtype: bool
 |  
 |  eq(self, other, axis='columns', level=None)
 |      Get Equal to of dataframe and other, element-wise (binary operator `eq`).
 |  
 |      Among flexible wrappers (`eq`, `ne`, `le`, `lt`, `ge`, `gt`) to comparison
 |      operators.
 |  
 |      Equivalent to `==`, `!=`, `<=`, `<`, `>=`, `>` with support to choose axis
 |      (rows or columns) and level for comparison.
 |  
 |      Parameters
 |      ----------
 |      other : scalar, sequence, Series, or DataFrame
 |          Any single or multiple element data structure, or list-like object.
 |      axis : {0 or 'index', 1 or 'columns'}, default 'columns'
 |          Whether to compare by the index (0 or 'index') or columns
 |          (1 or 'columns').
 |      level : int or label
 |          Broadcast across a level, matching Index values on the passed
 |          MultiIndex level.
 |  
 |      Returns
 |      -------
 |      DataFrame of bool
 |          Result of the comparison.
 |
 |      See Also
 |      --------
 |      DataFrame.eq : Compare DataFrames for equality elementwise.
 |      DataFrame.ne : Compare DataFrames for inequality elementwise.
 |      DataFrame.le : Compare DataFrames for less than inequality
 |          or equality elementwise.
 |      DataFrame.lt : Compare DataFrames for strictly less than
 |          inequality elementwise.
 |      DataFrame.ge : Compare DataFrames for greater than inequality
 |          or equality elementwise.
 |      DataFrame.gt : Compare DataFrames for strictly greater than
 |          inequality elementwise.
 |
 |      Notes
 |      -----
 |      Mismatched indices will be unioned together.
 |      `NaN` values are considered different (i.e. `NaN` != `NaN`).
 |
 |      Examples
 |      --------
 |      >>> df = pd.DataFrame({'cost': [250, 150, 100],
 |      ...                    'revenue': [100, 250, 300]},
 |      ...                   index=['A', 'B', 'C'])
 |      >>> df
 |         cost  revenue
 |      A   250      100
 |      B   150      250
 |      C   100      300
 |
 |      Comparison with a scalar, using either the operator or method:
 |
 |      >>> df == 100
 |          cost  revenue
 |      A  False     True
 |      B  False    False
 |      C   True    False
 |
 |      >>> df.eq(100)
 |          cost  revenue
 |      A  False     True
 |      B  False    False
 |      C   True    False
 |
 |      When `other` is a :class:`Series`, the columns of a DataFrame are aligned
 |      with the index of `other` and broadcast:
 |
 |      >>> df != pd.Series([100, 250], index=["cost", "revenue"])
 |          cost  revenue
 |      A   True     True
 |      B   True    False
 |      C  False     True
 |
 |      Use the method to control the broadcast axis:
 |
 |      >>> df.ne(pd.Series([100, 300], index=["A", "D"]), axis='index')
 |         cost  revenue
 |      A  True    False
 |      B  True     True
 |      C  True     True
 |      D  True     True
 |
 |      When comparing to an arbitrary sequence, the number of columns must
 |      match the number elements in `other`:
 |
 |      >>> df == [250, 100]
 |          cost  revenue
 |      A   True     True
 |      B  False    False
 |      C  False    False
 |
 |      Use the method to control the axis:
 |
 |      >>> df.eq([250, 250, 100], axis='index')
 |          cost  revenue
 |      A   True    False
 |      B  False     True
 |      C   True    False
 |
 |      Compare to a DataFrame of different shape.
 |
 |      >>> other = pd.DataFrame({'revenue': [300, 250, 100, 150]},
 |      ...                      index=['A', 'B', 'C', 'D'])
 |      >>> other
 |         revenue
 |      A      300
 |      B      250
 |      C      100
 |      D      150
 |
 |      >>> df.gt(other)
 |          cost  revenue
 |      A  False    False
 |      B  False    False
 |      C  False     True
 |      D  False    False
 |
 |      Compare to a MultiIndex by level.
 |
 |      >>> df_multiindex = pd.DataFrame({'cost': [250, 150, 100, 150, 300, 220],
 |      ...                              'revenue': [100, 250, 300, 200, 175, 225]},
 |      ...                             index=[['Q1', 'Q1', 'Q1', 'Q2', 'Q2', 'Q2'],
 |      ...                                    ['A', 'B', 'C', 'A', 'B', 'C']])
 |      >>> df_multiindex
 |            cost  revenue
 |      Q1 A   250      100
 |         B   150      250
 |         C   100      300
 |      Q2 A   150      200
 |         B   300      175
 |         C   220      225
 |
 |      >>> df.le(df_multiindex, level=1)
 |             cost  revenue
 |      Q1 A   True     True
 |         B   True     True
 |         C   True     True
 |      Q2 A  False     True
 |         B   True    False
 |         C   True    False
 |
 |  eval(self, expr: 'str', inplace: 'bool' = False, **kwargs)
 |      Evaluate a string describing operations on DataFrame columns.
 |
 |      Operates on columns only, not specific rows or elements.  This allows
 |      `eval` to run arbitrary code, which can make you vulnerable to code
 |      injection if you pass user input to this function.
 |
 |      Parameters
 |      ----------
 |      expr : str
 |          The expression string to evaluate.
 |      inplace : bool, default False
 |          If the expression contains an assignment, whether to perform the
 |          operation inplace and mutate the existing DataFrame. Otherwise,
 |          a new DataFrame is returned.
 |      **kwargs
 |          See the documentation for :func:`eval` for complete details
 |          on the keyword arguments accepted by
 |          :meth:`~pandas.DataFrame.query`.
 |
 |      Returns
 |      -------
 |      ndarray, scalar, pandas object, or None
 |          The result of the evaluation or None if ``inplace=True``.
 |
 |      See Also
 |      --------
 |      DataFrame.query : Evaluates a boolean expression to query the columns
 |          of a frame.
 |      DataFrame.assign : Can evaluate an expression or function to create new
 |          values for a column.
 |      eval : Evaluate a Python expression as a string using various
 |          backends.
 |
 |      Notes
 |      -----
 |      For more details see the API documentation for :func:`~eval`.
 |      For detailed examples see :ref:`enhancing performance with eval
 |      <enhancingperf.eval>`.
 |
 |      Examples
 |      --------
 |      >>> df = pd.DataFrame({'A': range(1, 6), 'B': range(10, 0, -2)})
 |      >>> df
 |         A   B
 |      0  1  10
 |      1  2   8
 |      2  3   6
 |      3  4   4
 |      4  5   2
 |      >>> df.eval('A + B')
 |      0    11
 |      1    10
 |      2     9
 |      3     8
 |      4     7
 |      dtype: int64
 |
 |      Assignment is allowed though by default the original DataFrame is not
 |      modified.
 |
 |      >>> df.eval('C = A + B')
 |         A   B   C
 |      0  1  10  11
 |      1  2   8  10
 |      2  3   6   9
 |      3  4   4   8
 |      4  5   2   7
 |      >>> df
 |         A   B
 |      0  1  10
 |      1  2   8
 |      2  3   6
 |      3  4   4
 |      4  5   2
 |
 |      Use ``inplace=True`` to modify the original DataFrame.
 |
 |      >>> df.eval('C = A + B', inplace=True)
 |      >>> df
 |         A   B   C
 |      0  1  10  11
 |      1  2   8  10
 |      2  3   6   9
 |      3  4   4   8
 |      4  5   2   7
 |
 |      Multiple columns can be assigned to using multi-line expressions:
 |
 |      >>> df.eval(
 |      ...     '''
 |      ... C = A + B
 |      ... D = A - B
 |      ... '''
 |      ... )
 |         A   B   C  D
 |      0  1  10  11 -9
 |      1  2   8  10 -6
 |      2  3   6   9 -3
 |      3  4   4   8  0
 |      4  5   2   7  3
 |
 |  explode(self, column: 'str | tuple | list[str | tuple]', ignore_index: 'bool' =
False) -> 'DataFrame'
 |      Transform each element of a list-like to a row, replicating index values.
 |
 |      .. versionadded:: 0.25.0
 |
 |      Parameters
 |      ----------
 |      column : str or tuple or list thereof
 |          Column(s) to explode.
 |          For multiple columns, specify a non-empty list with each element
 |          be str or tuple, and all specified columns their list-like data
 |          on same row of the frame must have matching length.
 |
 |          .. versionadded:: 1.3.0
 |              Multi-column explode
 |
 |      ignore_index : bool, default False
 |          If True, the resulting index will be labeled 0, 1, …, n - 1.
 |
 |          .. versionadded:: 1.1.0
 |
 |      Returns
 |      -------
 |      DataFrame
 |          Exploded lists to rows of the subset columns;
 |          index will be duplicated for these rows.
 |
 |      Raises
 |      ------
 |      ValueError :
 |          * If columns of the frame are not unique.
 |          * If specified columns to explode is empty list.
 |          * If specified columns to explode have not matching count of
 |             elements rowwise in the frame.
 |
 |      See Also
 |      --------
 |      DataFrame.unstack : Pivot a level of the (necessarily hierarchical)
 |          index labels.
 |      DataFrame.melt : Unpivot a DataFrame from wide format to long format.
 |      Series.explode : Explode a DataFrame from list-like columns to long format.
 |
 |      Notes
 |      -----
 |      This routine will explode list-likes including lists, tuples, sets,
 |      Series, and np.ndarray. The result dtype of the subset rows will
 |      be object. Scalars will be returned unchanged, and empty list-likes will
 |      result in a np.nan for that row. In addition, the ordering of rows in the
 |      output will be non-deterministic when exploding sets.
 |
 |      Examples
 |      --------
 |      >>> df = pd.DataFrame({'A': [[0, 1, 2], 'foo', [], [3, 4]],
 |      ...                    'B': 1,
 |      ...                    'C': [['a', 'b', 'c'], np.nan, [], ['d', 'e']]})
 |      >>> df
 |                 A  B          C
 |      0  [0, 1, 2]  1  [a, b, c]
 |      1        foo  1        NaN
 |      2         []  1         []
 |      3     [3, 4]  1     [d, e]
 |
 |      Single-column explode.
 |
 |      >>> df.explode('A')
 |           A  B          C
 |      0    0  1  [a, b, c]
 |      0    1  1  [a, b, c]
 |      0    2  1  [a, b, c]
 |      1  foo  1        NaN
 |      2  NaN  1         []
 |      3    3  1     [d, e]
 |      3    4  1     [d, e]
 |
 |      Multi-column explode.
 |
 |      >>> df.explode(list('AC'))
 |           A  B    C
 |      0    0  1    a
 |      0    1  1    b
 |      0    2  1    c
 |      1  foo  1  NaN
 |      2  NaN  1  NaN
 |      3    3  1    d
 |      3    4  1    e
 |
 |  ffill(self: 'DataFrame', axis: 'None | Axis' = None, inplace: 'bool' = False,
limit: 'None | int' = None, downcast=None) -> 'DataFrame | None'
 |      Synonym for :meth:`DataFrame.fillna` with ``method='ffill'``.
 |
 |      Returns
 |      -------
 |      Series/DataFrame or None
 |          Object with missing values filled or None if ``inplace=True``.
 |
 |  fillna(self, value: 'object | ArrayLike | None' = None, method: 'FillnaOptions |
None' = None, axis: 'Axis | None' = None, inplace: 'bool' = False, limit=None,
downcast=None) -> 'DataFrame | None'
 |      Fill NA/NaN values using the specified method.
 |
 |      Parameters
 |      ----------
 |      value : scalar, dict, Series, or DataFrame
 |          Value to use to fill holes (e.g. 0), alternately a
 |          dict/Series/DataFrame of values specifying which value to use for
 |          each index (for a Series) or column (for a DataFrame).  Values not
 |          in the dict/Series/DataFrame will not be filled. This value cannot
 |          be a list.
 |      method : {'backfill', 'bfill', 'pad', 'ffill', None}, default None
 |          Method to use for filling holes in reindexed Series
 |          pad / ffill: propagate last valid observation forward to next valid
 |          backfill / bfill: use next valid observation to fill gap.
 |      axis : {0 or 'index', 1 or 'columns'}
 |          Axis along which to fill missing values.
 |      inplace : bool, default False
 |          If True, fill in-place. Note: this will modify any
 |          other views on this object (e.g., a no-copy slice for a column in a
 |          DataFrame).
 |      limit : int, default None
 |          If method is specified, this is the maximum number of consecutive
 |          NaN values to forward/backward fill. In other words, if there is
 |          a gap with more than this number of consecutive NaNs, it will only
 |          be partially filled. If method is not specified, this is the
 |          maximum number of entries along the entire axis where NaNs will be
 |          filled. Must be greater than 0 if not None.
 |      downcast : dict, default is None
 |          A dict of item->dtype of what to downcast if possible,
 |          or the string 'infer' which will try to downcast to an appropriate
 |          equal type (e.g. float64 to int64 if possible).
 |
 |      Returns
 |      -------
 |      DataFrame or None
 |          Object with missing values filled or None if ``inplace=True``.
 |
 |      See Also
 |      --------
 |      interpolate : Fill NaN values using interpolation.
 |      reindex : Conform object to new index.
 |      asfreq : Convert TimeSeries to specified frequency.
 |
 |      Examples
 |      --------
 |      >>> df = pd.DataFrame([[np.nan, 2, np.nan, 0],
 |      ...                    [3, 4, np.nan, 1],
 |      ...                    [np.nan, np.nan, np.nan, 5],
 |      ...                    [np.nan, 3, np.nan, 4]],
 |      ...                   columns=list("ABCD"))
 |      >>> df
 |           A    B   C  D
 |      0  NaN  2.0 NaN  0
 |      1  3.0  4.0 NaN  1
 |      2  NaN  NaN NaN  5
 |      3  NaN  3.0 NaN  4
 |
 |      Replace all NaN elements with 0s.
 |
 |      >>> df.fillna(0)
 |          A   B   C   D
 |      0   0.0 2.0 0.0 0
 |      1   3.0 4.0 0.0 1
 |      2   0.0 0.0 0.0 5
 |      3   0.0 3.0 0.0 4
 |
 |      We can also propagate non-null values forward or backward.
 |
 |      >>> df.fillna(method="ffill")
 |          A   B   C   D
 |      0   NaN 2.0 NaN 0
 |      1   3.0 4.0 NaN 1
 |      2   3.0 4.0 NaN 5
 |      3   3.0 3.0 NaN 4
 |
 |      Replace all NaN elements in column 'A', 'B', 'C', and 'D', with 0, 1,
 |      2, and 3 respectively.
 |
 |      >>> values = {"A": 0, "B": 1, "C": 2, "D": 3}
 |      >>> df.fillna(value=values)
 |          A   B   C   D
 |      0   0.0 2.0 2.0 0
 |      1   3.0 4.0 2.0 1
 |      2   0.0 1.0 2.0 5
 |      3   0.0 3.0 2.0 4
 |
 |      Only replace the first NaN element.
 |
 |      >>> df.fillna(value=values, limit=1)
 |          A   B   C   D
 |      0   0.0 2.0 2.0 0
 |      1   3.0 4.0 NaN 1
 |      2   NaN 1.0 NaN 5
 |      3   NaN 3.0 NaN 4
 |
 |      When filling using a DataFrame, replacement happens along
 |      the same column names and same indices
 |
 |      >>> df2 = pd.DataFrame(np.zeros((4, 4)), columns=list("ABCE"))
 |      >>> df.fillna(df2)
 |          A   B   C   D
 |      0   0.0 2.0 0.0 0
 |      1   3.0 4.0 0.0 1
 |      2   0.0 0.0 0.0 5
 |      3   0.0 3.0 0.0 4
 |
 |  floordiv(self, other, axis='columns', level=None, fill_value=None)
 |      Get Integer division of dataframe and other, element-wise (binary operator
`floordiv`).
```

```
 |
 |      Equivalent to ``dataframe // other``, but with support to substitute a
fill_value
 |      for missing data in one of the inputs. With reverse version, `rfloordiv`.
 |
 |      Among flexible wrappers (`add`, `sub`, `mul`, `div`, `mod`, `pow`) to
 |      arithmetic operators: `+`, `-`, `*`, `/`, `//`, `%`, `**`.
 |
 |      Parameters
 |      ----------
 |      other : scalar, sequence, Series, or DataFrame
 |          Any single or multiple element data structure, or list-like object.
 |      axis : {0 or 'index', 1 or 'columns'}
 |          Whether to compare by the index (0 or 'index') or columns
 |          (1 or 'columns'). For Series input, axis to match Series index on.
 |      level : int or label
 |          Broadcast across a level, matching Index values on the
 |          passed MultiIndex level.
 |      fill_value : float or None, default None
 |          Fill existing missing (NaN) values, and any new element needed for
 |          successful DataFrame alignment, with this value before computation.
 |          If data in both corresponding DataFrame locations is missing
 |          the result will be missing.
 |
 |      Returns
 |      -------
 |      DataFrame
 |          Result of the arithmetic operation.
 |
 |      See Also
 |      --------
 |      DataFrame.add : Add DataFrames.
 |      DataFrame.sub : Subtract DataFrames.
 |      DataFrame.mul : Multiply DataFrames.
 |      DataFrame.div : Divide DataFrames (float division).
 |      DataFrame.truediv : Divide DataFrames (float division).
 |      DataFrame.floordiv : Divide DataFrames (integer division).
 |      DataFrame.mod : Calculate modulo (remainder after division).
 |      DataFrame.pow : Calculate exponential power.
 |
 |      Notes
 |      -----
 |      Mismatched indices will be unioned together.
 |
 |      Examples
 |      --------
 |      >>> df = pd.DataFrame({'angles': [0, 3, 4],
 |      ...                    'degrees': [360, 180, 360]},
 |      ...                   index=['circle', 'triangle', 'rectangle'])
 |      >>> df
 |                 angles  degrees
 |      circle          0      360
 |      triangle        3      180
 |      rectangle       4      360
 |
 |      Add a scalar with operator version which return the same
 |      results.
 |
 |      >>> df + 1
 |                 angles  degrees
 |      circle          1      361
 |      triangle        4      181
 |      rectangle       5      361
 |
 |      >>> df.add(1)
 |                 angles  degrees
 |      circle          1      361
 |      triangle        4      181
 |      rectangle       5      361
 |
 |      Divide by constant with reverse version.
 |
 |      >>> df.div(10)
 |                 angles  degrees
 |      circle        0.0     36.0
 |      triangle      0.3     18.0
 |      rectangle     0.4     36.0
 |
 |      >>> df.rdiv(10)
 |                   angles   degrees
 |      circle          inf  0.027778
 |      triangle   3.333333  0.055556
 |      rectangle  2.500000  0.027778
 |
 |      Subtract a list and Series by axis with operator version.
 |
 |      >>> df - [1, 2]
 |                 angles  degrees
 |      circle         -1      358
 |      triangle        2      178
 |      rectangle       3      358
 |      
 |      >>> df.sub([1, 2], axis='columns')
 |                 angles  degrees
 |      circle         -1      358
 |      triangle        2      178
 |      rectangle       3      358
 |      
 |      >>> df.sub(pd.Series([1, 1, 1], index=['circle', 'triangle', 'rectangle']),
 |      ...        axis='index')
 |                 angles  degrees
 |      circle         -1      359
 |      triangle        2      179
 |      rectangle       3      359
 |      
 |      Multiply a DataFrame of different shape with operator version.
 |      
 |      >>> other = pd.DataFrame({'angles': [0, 3, 4]},
 |      ...                      index=['circle', 'triangle', 'rectangle'])
 |      >>> other
 |                 angles
 |      circle          0
 |      triangle        3
 |      rectangle       4
 |      
 |      >>> df * other
 |                 angles  degrees
 |      circle          0      NaN
 |      triangle        9      NaN
 |      rectangle      16      NaN
 |      
 |      >>> df.mul(other, fill_value=0)
 |                 angles  degrees
 |      circle          0      0.0
 |      triangle        9      0.0
 |      rectangle      16      0.0
 |      
 |      Divide by a MultiIndex by level.
 |      
 |      >>> df_multindex = pd.DataFrame({'angles': [0, 3, 4, 4, 5, 6],
 |      ...                              'degrees': [360, 180, 360, 360, 540, 720]},
 |      ...                             index=[['A', 'A', 'A', 'B', 'B', 'B'],
 |      ...                                    ['circle', 'triangle', 'rectangle',
 |      ...                                     'square', 'pentagon', 'hexagon']])
 |      >>> df_multindex
 |                   angles  degrees
 |      A circle          0      360
 |        triangle        3      180
 |        rectangle       4      360
 |      B square          4      360
 |        pentagon        5      540
 |        hexagon         6      720
 |      
 |      >>> df.div(df_multindex, level=1, fill_value=0)
 |                   angles  degrees
 |      A circle        NaN      1.0
 |        triangle      1.0      1.0
 |        rectangle     1.0      1.0
 |      B square        0.0      0.0
 |        pentagon      0.0      0.0
 |        hexagon       0.0      0.0
 |  
 |  ge(self, other, axis='columns', level=None)
 |      Get Greater than or equal to of dataframe and other, element-wise (binary
operator `ge`).
 |      
 |      Among flexible wrappers (`eq`, `ne`, `le`, `lt`, `ge`, `gt`) to comparison
 |      operators.
 |      
 |      Equivalent to `==`, `!=`, `<=`, `<`, `>=`, `>` with support to choose axis
 |      (rows or columns) and level for comparison.
 |      
 |      Parameters
 |      ----------
 |      other : scalar, sequence, Series, or DataFrame
 |          Any single or multiple element data structure, or list-like object.
 |      axis : {0 or 'index', 1 or 'columns'}, default 'columns'
 |          Whether to compare by the index (0 or 'index') or columns
 |          (1 or 'columns').
 |      level : int or label
 |          Broadcast across a level, matching Index values on the passed
 |          MultiIndex level.
 |      
 |      Returns
 |      -------
 |      DataFrame of bool
 |          Result of the comparison.
 |
 |      See Also
 |      --------
 |      DataFrame.eq : Compare DataFrames for equality elementwise.
 |      DataFrame.ne : Compare DataFrames for inequality elementwise.
 |      DataFrame.le : Compare DataFrames for less than inequality
 |          or equality elementwise.
 |      DataFrame.lt : Compare DataFrames for strictly less than
 |          inequality elementwise.
 |      DataFrame.ge : Compare DataFrames for greater than inequality
 |          or equality elementwise.
 |      DataFrame.gt : Compare DataFrames for strictly greater than
 |          inequality elementwise.
 |
 |      Notes
 |      -----
 |      Mismatched indices will be unioned together.
 |      `NaN` values are considered different (i.e. `NaN` != `NaN`).
 |
 |      Examples
 |      --------
 |      >>> df = pd.DataFrame({'cost': [250, 150, 100],
 |      ...                    'revenue': [100, 250, 300]},
 |      ...                   index=['A', 'B', 'C'])
 |      >>> df
 |         cost  revenue
 |      A   250      100
 |      B   150      250
 |      C   100      300
 |
 |      Comparison with a scalar, using either the operator or method:
 |
 |      >>> df == 100
 |          cost  revenue
 |      A  False     True
 |      B  False    False
 |      C   True    False
 |
 |      >>> df.eq(100)
 |          cost  revenue
 |      A  False     True
 |      B  False    False
 |      C   True    False
 |
 |      When `other` is a :class:`Series`, the columns of a DataFrame are aligned
 |      with the index of `other` and broadcast:
 |
 |      >>> df != pd.Series([100, 250], index=["cost", "revenue"])
 |          cost  revenue
 |      A   True     True
 |      B   True    False
 |      C  False     True
 |
 |      Use the method to control the broadcast axis:
 |
 |      >>> df.ne(pd.Series([100, 300], index=["A", "D"]), axis='index')
 |         cost  revenue
 |      A  True    False
 |      B  True     True
 |      C  True     True
 |      D  True     True
 |
 |      When comparing to an arbitrary sequence, the number of columns must
 |      match the number elements in `other`:
 |
 |      >>> df == [250, 100]
 |          cost  revenue
 |      A   True     True
 |      B  False    False
 |      C  False    False
 |
 |      Use the method to control the axis:
 |
 |      >>> df.eq([250, 250, 100], axis='index')
 |          cost  revenue
 |      A   True    False
 |      B  False     True
 |      C   True    False
 |
 |      Compare to a DataFrame of different shape.
 |
 |      >>> other = pd.DataFrame({'revenue': [300, 250, 100, 150]},
 |      ...                      index=['A', 'B', 'C', 'D'])
 |      >>> other
 |         revenue
 |      A      300
 |      B      250
 |      C      100
 |      D      150
 |      
 |      >>> df.gt(other)
 |          cost  revenue
 |      A  False    False
 |      B  False    False
 |      C  False     True
 |      D  False    False
 |      
 |      Compare to a MultiIndex by level.
 |      
 |      >>> df_multiindex = pd.DataFrame({'cost': [250, 150, 100, 150, 300, 220],
 |      ...                              'revenue': [100, 250, 300, 200, 175, 225]},
 |      ...                             index=[['Q1', 'Q1', 'Q1', 'Q2', 'Q2', 'Q2'],
 |      ...                                    ['A', 'B', 'C', 'A', 'B', 'C']])
 |      >>> df_multiindex
 |            cost  revenue
 |      Q1 A   250      100
 |         B   150      250
 |         C   100      300
 |      Q2 A   150      200
 |         B   300      175
 |         C   220      225
 |      
 |      >>> df.le(df_multiindex, level=1)
 |             cost  revenue
 |      Q1 A   True     True
 |         B   True     True
 |         C   True     True
 |      Q2 A  False     True
 |         B   True    False
 |         C   True    False
 |  
 |  groupby(self, by=None, axis: 'Axis' = 0, level: 'Level | None' = None, as_index:
'bool' = True, sort: 'bool' = True, group_keys: 'bool' = True, squeeze: 'bool |
lib.NoDefault' = <no_default>, observed: 'bool' = False, dropna: 'bool' = True) ->
'DataFrameGroupBy'
 |      Group DataFrame using a mapper or by a Series of columns.
 |      
 |      A groupby operation involves some combination of splitting the
 |      object, applying a function, and combining the results. This can be
 |      used to group large amounts of data and compute operations on these
 |      groups.
 |      
 |      Parameters
 |      ----------
 |      by : mapping, function, label, or list of labels
 |          Used to determine the groups for the groupby.
 |          If ``by`` is a function, it's called on each value of the object's
 |          index. If a dict or Series is passed, the Series or dict VALUES
 |          will be used to determine the groups (the Series' values are first
 |          aligned; see ``.align()`` method). If an ndarray is passed, the
 |          values are used as-is to determine the groups. A label or list of
 |          labels may be passed to group by the columns in ``self``. Notice
 |          that a tuple is interpreted as a (single) key.
 |      axis : {0 or 'index', 1 or 'columns'}, default 0
 |          Split along rows (0) or columns (1).
 |      level : int, level name, or sequence of such, default None
 |          If the axis is a MultiIndex (hierarchical), group by a particular
 |          level or levels.
 |      as_index : bool, default True
 |          For aggregated output, return object with group labels as the
 |          index. Only relevant for DataFrame input. as_index=False is
 |          effectively "SQL-style" grouped output.
 |      sort : bool, default True
 |          Sort group keys. Get better performance by turning this off.
 |          Note this does not influence the order of observations within each
 |          group. Groupby preserves the order of rows within each group.
 |      group_keys : bool, default True
 |          When calling apply, add group keys to index to identify pieces.
 |      squeeze : bool, default False
 |          Reduce the dimensionality of the return type if possible,
 |          otherwise return a consistent type.
 |      
 |          .. deprecated:: 1.1.0
 |      
 |      observed : bool, default False
 |          This only applies if any of the groupers are Categoricals.
 |          If True: only show observed values for categorical groupers.
 |          If False: show all values for categorical groupers.
 |      dropna : bool, default True
 |          If True, and if group keys contain NA values, NA values together
 |          with row/column will be dropped.
 |          If False, NA values will also be treated as the key in groups
 |
 |          .. versionadded:: 1.1.0
 |
 |      Returns
 |      -------
 |      DataFrameGroupBy
 |          Returns a groupby object that contains information about the groups.
 |
 |      See Also
 |      --------
 |      resample : Convenience method for frequency conversion and resampling
 |          of time series.
 |
 |      Notes
 |      -----
 |      See the `user guide
 |      <https://pandas.pydata.org/pandas-docs/stable/groupby.html>`__ for more.
 |
 |      Examples
 |      --------
 |      >>> df = pd.DataFrame({'Animal': ['Falcon', 'Falcon',
 |      ...                               'Parrot', 'Parrot'],
 |      ...                    'Max Speed': [380., 370., 24., 26.]})
 |      >>> df
 |         Animal  Max Speed
 |      0  Falcon      380.0
 |      1  Falcon      370.0
 |      2  Parrot       24.0
 |      3  Parrot       26.0
 |      >>> df.groupby(['Animal']).mean()
 |              Max Speed
 |      Animal
 |      Falcon      375.0
 |      Parrot       25.0
 |
 |      **Hierarchical Indexes**
 |
 |      We can groupby different levels of a hierarchical index
 |      using the `level` parameter:
 |
 |      >>> arrays = [['Falcon', 'Falcon', 'Parrot', 'Parrot'],
 |      ...           ['Captive', 'Wild', 'Captive', 'Wild']]
 |      >>> index = pd.MultiIndex.from_arrays(arrays, names=('Animal', 'Type'))
 |      >>> df = pd.DataFrame({'Max Speed': [390., 350., 30., 20.]},
 |      ...                   index=index)
 |      >>> df
 |                      Max Speed
 |      Animal Type
 |      Falcon Captive      390.0
 |             Wild         350.0
 |      Parrot Captive       30.0
 |             Wild          20.0
 |      >>> df.groupby(level=0).mean()
 |              Max Speed
 |      Animal
 |      Falcon      370.0
 |      Parrot       25.0
 |      >>> df.groupby(level="Type").mean()
 |               Max Speed
 |      Type
 |      Captive      210.0
 |      Wild         185.0
 |
 |      We can also choose to include NA in group keys or not by setting
 |      `dropna` parameter, the default setting is `True`:
 |
 |      >>> l = [[1, 2, 3], [1, None, 4], [2, 1, 3], [1, 2, 2]]
 |      >>> df = pd.DataFrame(l, columns=["a", "b", "c"])
 |
 |      >>> df.groupby(by=["b"]).sum()
 |          a   c
 |      b
 |      1.0 2   3
 |      2.0 2   5
 |
 |      >>> df.groupby(by=["b"], dropna=False).sum()
 |          a   c
 |      b
 |      1.0 2   3
 |      2.0 2   5
 |      NaN 1   4
 |
 |      >>> l = [["a", 12, 12], [None, 12.3, 33.], ["b", 12.3, 123], ["a", 1, 1]]
 |      >>> df = pd.DataFrame(l, columns=["a", "b", "c"])
 |
 |      >>> df.groupby(by="a").sum()
 |          b     c
 |      a
 |      a   13.0   13.0
 |      b   12.3  123.0
 |
 |      >>> df.groupby(by="a", dropna=False).sum()
 |          b     c
 |      a
 |      a   13.0   13.0
 |      b   12.3  123.0
 |      NaN 12.3   33.0
 |
 |  gt(self, other, axis='columns', level=None)
 |      Get Greater than of dataframe and other, element-wise (binary operator `gt`).
 |
 |      Among flexible wrappers (`eq`, `ne`, `le`, `lt`, `ge`, `gt`) to comparison
 |      operators.
 |
 |      Equivalent to `==`, `!=`, `<=`, `<`, `>=`, `>` with support to choose axis
 |      (rows or columns) and level for comparison.
 |
 |      Parameters
 |      ----------
 |      other : scalar, sequence, Series, or DataFrame
 |          Any single or multiple element data structure, or list-like object.
 |      axis : {0 or 'index', 1 or 'columns'}, default 'columns'
 |          Whether to compare by the index (0 or 'index') or columns
 |          (1 or 'columns').
 |      level : int or label
 |          Broadcast across a level, matching Index values on the passed
 |          MultiIndex level.
 |
 |      Returns
 |      -------
 |      DataFrame of bool
 |          Result of the comparison.
 |
 |      See Also
 |      --------
 |      DataFrame.eq : Compare DataFrames for equality elementwise.
 |      DataFrame.ne : Compare DataFrames for inequality elementwise.
 |      DataFrame.le : Compare DataFrames for less than inequality
 |          or equality elementwise.
 |      DataFrame.lt : Compare DataFrames for strictly less than
 |          inequality elementwise.
 |      DataFrame.ge : Compare DataFrames for greater than inequality
 |          or equality elementwise.
 |      DataFrame.gt : Compare DataFrames for strictly greater than
 |          inequality elementwise.
 |
 |      Notes
 |      -----
 |      Mismatched indices will be unioned together.
 |      `NaN` values are considered different (i.e. `NaN` != `NaN`).
 |
 |      Examples
 |      --------
 |      >>> df = pd.DataFrame({'cost': [250, 150, 100],
 |      ...                    'revenue': [100, 250, 300]},
 |      ...                   index=['A', 'B', 'C'])
 |      >>> df
 |         cost  revenue
 |      A   250      100
 |      B   150      250
 |      C   100      300
 |
 |      Comparison with a scalar, using either the operator or method:
 |
 |      >>> df == 100
 |          cost  revenue
 |      A  False     True
 |      B  False    False
 |      C   True    False
 |
 |      >>> df.eq(100)
 |          cost  revenue
 |      A  False     True
 |      B  False    False
 |      C   True    False
 |
 |      When `other` is a :class:`Series`, the columns of a DataFrame are aligned
 |      with the index of `other` and broadcast:
 |
 |      >>> df != pd.Series([100, 250], index=["cost", "revenue"])
 |          cost  revenue
 |      A   True     True
 |      B   True    False
```

```
 |      C  False     True
 |      
 |      Use the method to control the broadcast axis:
 |      
 |      >>> df.ne(pd.Series([100, 300], index=["A", "D"]), axis='index')
 |         cost  revenue
 |      A  True    False
 |      B  True     True
 |      C  True     True
 |      D  True     True
 |      
 |      When comparing to an arbitrary sequence, the number of columns must
 |      match the number elements in `other`:
 |      
 |      >>> df == [250, 100]
 |          cost  revenue
 |      A   True     True
 |      B  False    False
 |      C  False    False
 |      
 |      Use the method to control the axis:
 |      
 |      >>> df.eq([250, 250, 100], axis='index')
 |          cost  revenue
 |      A   True    False
 |      B  False     True
 |      C   True    False
 |      
 |      Compare to a DataFrame of different shape.
 |      
 |      >>> other = pd.DataFrame({'revenue': [300, 250, 100, 150]},
 |      ...                      index=['A', 'B', 'C', 'D'])
 |      >>> other
 |         revenue
 |      A      300
 |      B      250
 |      C      100
 |      D      150
 |      
 |      >>> df.gt(other)
 |          cost  revenue
 |      A  False    False
 |      B  False    False
 |      C  False     True
 |      D  False    False
 |      
 |      Compare to a MultiIndex by level.
 |      
 |      >>> df_multiindex = pd.DataFrame({'cost': [250, 150, 100, 150, 300, 220],
 |      ...                               'revenue': [100, 250, 300, 200, 175, 225]},
 |      ...                              index=[['Q1', 'Q1', 'Q1', 'Q2', 'Q2', 'Q2'],
 |      ...                                     ['A', 'B', 'C', 'A', 'B', 'C']])
 |      >>> df_multiindex
 |            cost  revenue
 |      Q1 A   250      100
 |         B   150      250
 |         C   100      300
 |      Q2 A   150      200
 |         B   300      175
 |         C   220      225
 |      
 |      >>> df.le(df_multiindex, level=1)
 |             cost  revenue
 |      Q1 A   True     True
 |         B   True     True
 |         C   True     True
 |      Q2 A  False     True
 |         B   True    False
 |         C   True    False
 |  
 |  hist = hist_frame(data: 'DataFrame', column: 'IndexLabel' = None, by=None, grid:
'bool' = True, xlabelsize: 'int | None' = None, xrot: 'float | None' = None,
ylabelsize: 'int | None' = None, yrot: 'float | None' = None, ax=None, sharex: 'bool'
= False, sharey: 'bool' = False, figsize: 'tuple[int, int] | None' = None, layout:
'tuple[int, int] | None' = None, bins: 'int | Sequence[int]' = 10, backend: 'str |
None' = None, legend: 'bool' = False, **kwargs)
 |      Make a histogram of the DataFrame's columns.
 |      
 |      A `histogram`_ is a representation of the distribution of data.
 |      This function calls :meth:`matplotlib.pyplot.hist`, on each series in
 |      the DataFrame, resulting in one histogram per column.
 |      
 |      .. _histogram: https://en.wikipedia.org/wiki/Histogram
 |      
 |      Parameters
 |      ----------
 |      data : DataFrame
 |          The pandas object holding the data.
 |      column : str or sequence, optional
 |          If passed, will be used to limit data to a subset of columns.
 |      by : object, optional
 |          If passed, then used to form histograms for separate groups.
 |      grid : bool, default True
 |          Whether to show axis grid lines.
 |      xlabelsize : int, default None
 |          If specified changes the x-axis label size.
 |      xrot : float, default None
 |          Rotation of x axis labels. For example, a value of 90 displays the
 |          x labels rotated 90 degrees clockwise.
 |      ylabelsize : int, default None
 |          If specified changes the y-axis label size.
 |      yrot : float, default None
 |          Rotation of y axis labels. For example, a value of 90 displays the
 |          y labels rotated 90 degrees clockwise.
 |      ax : Matplotlib axes object, default None
 |          The axes to plot the histogram on.
 |      sharex : bool, default True if ax is None else False
 |          In case subplots=True, share x axis and set some x axis labels to
 |          invisible; defaults to True if ax is None otherwise False if an ax
 |          is passed in.
 |          Note that passing in both an ax and sharex=True will alter all x axis
 |          labels for all subplots in a figure.
 |      sharey : bool, default False
 |          In case subplots=True, share y axis and set some y axis labels to
 |          invisible.
 |      figsize : tuple, optional
 |          The size in inches of the figure to create. Uses the value in
 |          `matplotlib.rcParams` by default.
 |      layout : tuple, optional
 |          Tuple of (rows, columns) for the layout of the histograms.
 |      bins : int or sequence, default 10
 |          Number of histogram bins to be used. If an integer is given, bins + 1
 |          bin edges are calculated and returned. If bins is a sequence, gives
 |          bin edges, including left edge of first bin and right edge of last
 |          bin. In this case, bins is returned unmodified.
 |
 |      backend : str, default None
 |          Backend to use instead of the backend specified in the option
 |          ``plotting.backend``. For instance, 'matplotlib'. Alternatively, to
 |          specify the ``plotting.backend`` for the whole session, set
 |          ``pd.options.plotting.backend``.
 |
 |          .. versionadded:: 1.0.0
 |
 |      legend : bool, default False
 |          Whether to show the legend.
 |
 |          .. versionadded:: 1.1.0
 |
 |      **kwargs
 |          All other plotting keyword arguments to be passed to
 |          :meth:`matplotlib.pyplot.hist`.
 |
 |      Returns
 |      -------
 |      matplotlib.AxesSubplot or numpy.ndarray of them
 |
 |      See Also
 |      --------
 |      matplotlib.pyplot.hist : Plot a histogram using matplotlib.
 |
 |      Examples
 |      --------
 |      This example draws a histogram based on the length and width of
 |      some animals, displayed in three bins
 |
 |      .. plot::
 |          :context: close-figs
 |
 |          >>> df = pd.DataFrame({
 |          ...     'length': [1.5, 0.5, 1.2, 0.9, 3],
 |          ...     'width': [0.7, 0.2, 0.15, 0.2, 1.1]
 |          ...     }, index=['pig', 'rabbit', 'duck', 'chicken', 'horse'])
 |          >>> hist = df.hist(bins=3)
 |
 |  idxmax(self, axis: 'Axis' = 0, skipna: 'bool' = True) -> 'Series'
 |      Return index of first occurrence of maximum over requested axis.
 |
 |      NA/null values are excluded.
 |
 |      Parameters
 |      ----------
 |      axis : {0 or 'index', 1 or 'columns'}, default 0
 |          The axis to use. 0 or 'index' for row-wise, 1 or 'columns' for column-
```

```
wise.
 |      skipna : bool, default True
 |          Exclude NA/null values. If an entire row/column is NA, the result
 |          will be NA.
 |
 |      Returns
 |      -------
 |      Series
 |          Indexes of maxima along the specified axis.
 |
 |      Raises
 |      ------
 |      ValueError
 |          * If the row/column is empty
 |
 |      See Also
 |      --------
 |      Series.idxmax : Return index of the maximum element.
 |
 |      Notes
 |      -----
 |      This method is the DataFrame version of ``ndarray.argmax``.
 |
 |      Examples
 |      --------
 |      Consider a dataset containing food consumption in Argentina.
 |
 |      >>> df = pd.DataFrame({'consumption': [10.51, 103.11, 55.48],
 |      ...                    'co2_emissions': [37.2, 19.66, 1712]},
 |      ...                    index=['Pork', 'Wheat Products', 'Beef'])
 |
 |      >>> df
 |                      consumption  co2_emissions
 |      Pork                  10.51         37.20
 |      Wheat Products       103.11         19.66
 |      Beef                  55.48       1712.00
 |
 |      By default, it returns the index for the maximum value in each column.
 |
 |      >>> df.idxmax()
 |      consumption     Wheat Products
 |      co2_emissions             Beef
 |      dtype: object
 |
 |      To return the index for the maximum value in each row, use
``axis="columns"``.
 |
 |      >>> df.idxmax(axis="columns")
 |      Pork              co2_emissions
 |      Wheat Products     consumption
 |      Beef              co2_emissions
 |      dtype: object
 |
 |  idxmin(self, axis: 'Axis' = 0, skipna: 'bool' = True) -> 'Series'
 |      Return index of first occurrence of minimum over requested axis.
 |
 |      NA/null values are excluded.
 |
 |      Parameters
 |      ----------
 |      axis : {0 or 'index', 1 or 'columns'}, default 0
 |          The axis to use. 0 or 'index' for row-wise, 1 or 'columns' for column-
wise.
 |      skipna : bool, default True
 |          Exclude NA/null values. If an entire row/column is NA, the result
 |          will be NA.
 |
 |      Returns
 |      -------
 |      Series
 |          Indexes of minima along the specified axis.
 |
 |      Raises
 |      ------
 |      ValueError
 |          * If the row/column is empty
 |
 |      See Also
 |      --------
 |      Series.idxmin : Return index of the minimum element.
 |
 |      Notes
 |      -----
 |      This method is the DataFrame version of ``ndarray.argmin``.
 |
 |      Examples
 |      --------
 |
 |      Consider a dataset containing food consumption in Argentina.
 |
 |      >>> df = pd.DataFrame({'consumption': [10.51, 103.11, 55.48],
 |      ...                    'co2_emissions': [37.2, 19.66, 1712]},
 |      ...                    index=['Pork', 'Wheat Products', 'Beef'])
 |
 |      >>> df
 |                      consumption  co2_emissions
 |      Pork                  10.51          37.20
 |      Wheat Products       103.11          19.66
 |      Beef                  55.48        1712.00
 |
 |      By default, it returns the index for the minimum value in each column.
 |
 |      >>> df.idxmin()
 |      consumption                Pork
 |      co2_emissions    Wheat Products
 |      dtype: object
 |
 |      To return the index for the minimum value in each row, use
``axis="columns"``.
 |
 |      >>> df.idxmin(axis="columns")
 |      Pork                consumption
 |      Wheat Products    co2_emissions
 |      Beef                consumption
 |      dtype: object
 |
 |  info(self, verbose: 'bool | None' = None, buf: 'IO[str] | None' = None, max_cols:
'int | None' = None, memory_usage: 'bool | str | None' = None, show_counts: 'bool |
None' = None, null_counts: 'bool | None' = None) -> 'None'
 |      Print a concise summary of a DataFrame.
 |
 |      This method prints information about a DataFrame including
 |      the index dtype and columns, non-null values and memory usage.
 |
 |      Parameters
 |      ----------
 |      data : DataFrame
 |          DataFrame to print information about.
 |      verbose : bool, optional
 |          Whether to print the full summary. By default, the setting in
 |          ``pandas.options.display.max_info_columns`` is followed.
 |      buf : writable buffer, defaults to sys.stdout
 |          Where to send the output. By default, the output is printed to
 |          sys.stdout. Pass a writable buffer if you need to further process
 |          the output.
 |      max_cols : int, optional
 |          When to switch from the verbose to the truncated output. If the
 |          DataFrame has more than `max_cols` columns, the truncated output
 |          is used. By default, the setting in
 |          ``pandas.options.display.max_info_columns`` is used.
 |      memory_usage : bool, str, optional
 |          Specifies whether total memory usage of the DataFrame
 |          elements (including the index) should be displayed. By default,
 |          this follows the ``pandas.options.display.memory_usage`` setting.
 |
 |          True always show memory usage. False never shows memory usage.
 |          A value of 'deep' is equivalent to "True with deep introspection".
 |          Memory usage is shown in human-readable units (base-2
 |          representation). Without deep introspection a memory estimation is
 |          made based in column dtype and number of rows assuming values
 |          consume the same memory amount for corresponding dtypes. With deep
 |          memory introspection, a real memory usage calculation is performed
 |          at the cost of computational resources.
 |      show_counts : bool, optional
 |          Whether to show the non-null counts. By default, this is shown
 |          only if the DataFrame is smaller than
 |          ``pandas.options.display.max_info_rows`` and
 |          ``pandas.options.display.max_info_columns``. A value of True always
 |          shows the counts, and False never shows the counts.
 |      null_counts : bool, optional
 |          .. deprecated:: 1.2.0
 |              Use show_counts instead.
 |
 |      Returns
 |      -------
 |      None
 |          This method prints a summary of a DataFrame and returns None.
 |
 |      See Also
 |      --------
 |      DataFrame.describe: Generate descriptive statistics of DataFrame
 |          columns.
 |      DataFrame.memory_usage: Memory usage of DataFrame columns.
 |
 |      Examples
 |      --------
 |      >>> int_values = [1, 2, 3, 4, 5]
 |      >>> text_values = ['alpha', 'beta', 'gamma', 'delta', 'epsilon']
 |      >>> float_values = [0.0, 0.25, 0.5, 0.75, 1.0]
 |      >>> df = pd.DataFrame({"int_col": int_values, "text_col": text_values,
 |      ...                   "float_col": float_values})
 |      >>> df
 |          int_col text_col  float_col
 |      0        1    alpha       0.00
 |      1        2     beta       0.25
 |      2        3    gamma       0.50
 |      3        4    delta       0.75
 |      4        5  epsilon       1.00
 |      
 |      Prints information of all columns:
 |      
 |      >>> df.info(verbose=True)
 |      <class 'pandas.core.frame.DataFrame'>
 |      RangeIndex: 5 entries, 0 to 4
 |      Data columns (total 3 columns):
 |       #   Column     Non-Null Count  Dtype
 |      ---  ------     --------------  -----
 |       0   int_col    5 non-null      int64
 |       1   text_col   5 non-null      object
 |       2   float_col  5 non-null      float64
 |      dtypes: float64(1), int64(1), object(1)
 |      memory usage: 248.0+ bytes
 |      
 |      Prints a summary of columns count and its dtypes but not per column
 |      information:
 |      
 |      >>> df.info(verbose=False)
 |      <class 'pandas.core.frame.DataFrame'>
 |      RangeIndex: 5 entries, 0 to 4
 |      Columns: 3 entries, int_col to float_col
 |      dtypes: float64(1), int64(1), object(1)
 |      memory usage: 248.0+ bytes
 |      
 |      Pipe output of DataFrame.info to buffer instead of sys.stdout, get
 |      buffer content and writes to a text file:
 |      
 |      >>> import io
 |      >>> buffer = io.StringIO()
 |      >>> df.info(buf=buffer)
 |      >>> s = buffer.getvalue()
 |      >>> with open("df_info.txt", "w",
 |      ...           encoding="utf-8") as f:  # doctest: +SKIP
 |      ...     f.write(s)
 |      260
 |      
 |      The `memory_usage` parameter allows deep introspection mode, specially
 |      useful for big DataFrames and fine-tune memory optimization:
 |      
 |      >>> random_strings_array = np.random.choice(['a', 'b', 'c'], 10 ** 6)
 |      >>> df = pd.DataFrame({
 |      ...     'column_1': np.random.choice(['a', 'b', 'c'], 10 ** 6),
 |      ...     'column_2': np.random.choice(['a', 'b', 'c'], 10 ** 6),
 |      ...     'column_3': np.random.choice(['a', 'b', 'c'], 10 ** 6)
 |      ... })
 |      >>> df.info()
 |      <class 'pandas.core.frame.DataFrame'>
 |      RangeIndex: 1000000 entries, 0 to 999999
 |      Data columns (total 3 columns):
 |       #   Column    Non-Null Count    Dtype
 |      ---  ------    --------------    -----
 |       0   column_1  1000000 non-null  object
 |       1   column_2  1000000 non-null  object
 |       2   column_3  1000000 non-null  object
 |      dtypes: object(3)
 |      memory usage: 22.9+ MB
 |      
 |      >>> df.info(memory_usage='deep')
 |      <class 'pandas.core.frame.DataFrame'>
 |      RangeIndex: 1000000 entries, 0 to 999999
 |      Data columns (total 3 columns):
 |       #   Column    Non-Null Count    Dtype
 |      ---  ------    --------------    -----
 |       0   column_1  1000000 non-null  object
 |       1   column_2  1000000 non-null  object
 |       2   column_3  1000000 non-null  object
 |      dtypes: object(3)
 |      memory usage: 165.9 MB
 |  
 |  insert(self, loc, column, value, allow_duplicates: 'bool' = False) -> 'None'
 |      Insert column into DataFrame at specified location.
 |      
 |      Raises a ValueError if `column` is already contained in the DataFrame,
 |      unless `allow_duplicates` is set to True.
 |      
 |      Parameters
 |      ----------
 |      loc : int
 |          Insertion index. Must verify 0 <= loc <= len(columns).
 |      column : str, number, or hashable object
 |          Label of the inserted column.
 |      value : int, Series, or array-like
 |      allow_duplicates : bool, optional
 |      
 |      See Also
 |      --------
 |      Index.insert : Insert new item by index.
 |      
 |      Examples
 |      --------
 |      >>> df = pd.DataFrame({'col1': [1, 2], 'col2': [3, 4]})
 |      >>> df
 |         col1  col2
 |      0     1     3
 |      1     2     4
 |      >>> df.insert(1, "newcol", [99, 99])
 |      >>> df
 |         col1  newcol  col2
 |      0     1      99     3
 |      1     2      99     4
 |      >>> df.insert(0, "col1", [100, 100], allow_duplicates=True)
 |      >>> df
 |         col1  col1  newcol  col2
 |      0   100     1      99     3
 |      1   100     2      99     4
 |      
 |      Notice that pandas uses index alignment in case of `value` from type
`Series`:
 |      
 |      >>> df.insert(0, "col0", pd.Series([5, 6], index=[1, 2]))
 |      >>> df
 |         col0  col1  col1  newcol  col2
 |      0   NaN   100     1      99     3
 |      1   5.0   100     2      99     4
 |  
 |  interpolate(self: 'DataFrame', method: 'str' = 'linear', axis: 'Axis' = 0, limit:
'int | None' = None, inplace: 'bool' = False, limit_direction: 'str | None' = None,
limit_area: 'str | None' = None, downcast: 'str | None' = None, **kwargs) ->
'DataFrame | None'
 |      Fill NaN values using an interpolation method.
 |      
 |      Please note that only ``method='linear'`` is supported for
 |      DataFrame/Series with a MultiIndex.
 |      
 |      Parameters
 |      ----------
 |      method : str, default 'linear'
 |          Interpolation technique to use. One of:
 |      
 |          * 'linear': Ignore the index and treat the values as equally
 |            spaced. This is the only method supported on MultiIndexes.
 |          * 'time': Works on daily and higher resolution data to interpolate
 |            given length of interval.
 |          * 'index', 'values': use the actual numerical values of the index.
 |          * 'pad': Fill in NaNs using existing values.
 |          * 'nearest', 'zero', 'slinear', 'quadratic', 'cubic', 'spline',
 |            'barycentric', 'polynomial': Passed to
 |            `scipy.interpolate.interp1d`. These methods use the numerical
 |            values of the index.  Both 'polynomial' and 'spline' require that
 |            you also specify an `order` (int), e.g.
 |            ``df.interpolate(method='polynomial', order=5)``.
 |          * 'krogh', 'piecewise_polynomial', 'spline', 'pchip', 'akima',
 |            'cubicspline': Wrappers around the SciPy interpolation methods of
 |            similar names. See `Notes`.
 |          * 'from_derivatives': Refers to
 |            `scipy.interpolate.BPoly.from_derivatives` which
 |            replaces 'piecewise_polynomial' interpolation method in
 |            scipy 0.18.
 |      
 |      axis : {{0 or 'index', 1 or 'columns', None}}, default None
 |          Axis to interpolate along.
 |      limit : int, optional
 |          Maximum number of consecutive NaNs to fill. Must be greater than
 |          0.
 |      inplace : bool, default False
 |          Update the data in place if possible.
 |      limit_direction : {{'forward', 'backward', 'both'}}, Optional
 |          Consecutive NaNs will be filled in this direction.
 |      
 |          If limit is specified:
```

```
 |              * If 'method' is 'pad' or 'ffill', 'limit_direction' must be
'forward'.
 |              * If 'method' is 'backfill' or 'bfill', 'limit_direction' must be
 |                'backwards'.
 |
 |          If 'limit' is not specified:
 |              * If 'method' is 'backfill' or 'bfill', the default is 'backward'
 |              * else the default is 'forward'
 |
 |          .. versionchanged:: 1.1.0
 |              raises ValueError if `limit_direction` is 'forward' or 'both' and
 |                  method is 'backfill' or 'bfill'.
 |              raises ValueError if `limit_direction` is 'backward' or 'both' and
 |                  method is 'pad' or 'ffill'.
 |
 |      limit_area : {{`None`, 'inside', 'outside'}}, default None
 |          If limit is specified, consecutive NaNs will be filled with this
 |          restriction.
 |
 |          * ``None``: No fill restriction.
 |          * 'inside': Only fill NaNs surrounded by valid values
 |            (interpolate).
 |          * 'outside': Only fill NaNs outside valid values (extrapolate).
 |
 |      downcast : optional, 'infer' or None, defaults to None
 |          Downcast dtypes if possible.
 |      ``**kwargs`` : optional
 |          Keyword arguments to pass on to the interpolating function.
 |
 |      Returns
 |      -------
 |      Series or DataFrame or None
 |          Returns the same object type as the caller, interpolated at
 |          some or all ``NaN`` values or None if ``inplace=True``.
 |
 |      See Also
 |      --------
 |      fillna : Fill missing values using different methods.
 |      scipy.interpolate.Akima1DInterpolator : Piecewise cubic polynomials
 |          (Akima interpolator).
 |      scipy.interpolate.BPoly.from_derivatives : Piecewise polynomial in the
 |          Bernstein basis.
 |      scipy.interpolate.interp1d : Interpolate a 1-D function.
 |      scipy.interpolate.KroghInterpolator : Interpolate polynomial (Krogh
 |          interpolator).
 |      scipy.interpolate.PchipInterpolator : PCHIP 1-d monotonic cubic
 |          interpolation.
 |      scipy.interpolate.CubicSpline : Cubic spline data interpolator.
 |
 |      Notes
 |      -----
 |      The 'krogh', 'piecewise_polynomial', 'spline', 'pchip' and 'akima'
 |      methods are wrappers around the respective SciPy implementations of
 |      similar names. These use the actual numerical values of the index.
 |      For more information on their behavior, see the
 |      `SciPy documentation
 |      <https://docs.scipy.org/doc/scipy/reference/interpolate.html#univariate-
interpolation>`__
 |      and `SciPy tutorial
 |      <https://docs.scipy.org/doc/scipy/reference/tutorial/interpolate.html>`__.
 |
 |      Examples
 |      --------
 |      Filling in ``NaN`` in a :class:`~pandas.Series` via linear
 |      interpolation.
 |
 |      >>> s = pd.Series([0, 1, np.nan, 3])
 |      >>> s
 |      0    0.0
 |      1    1.0
 |      2    NaN
 |      3    3.0
 |      dtype: float64
 |      >>> s.interpolate()
 |      0    0.0
 |      1    1.0
 |      2    2.0
 |      3    3.0
 |      dtype: float64
 |
 |      Filling in ``NaN`` in a Series by padding, but filling at most two
 |      consecutive ``NaN`` at a time.
 |
 |      >>> s = pd.Series([np.nan, "single_one", np.nan,
 |      ...                "fill_two_more", np.nan, np.nan, np.nan,
 |      ...                4.71, np.nan])
 |      >>> s
 |      0              NaN
 |      1       single_one
 |      2              NaN
 |      3    fill_two_more
 |      4              NaN
 |      5              NaN
 |      6              NaN
 |      7             4.71
 |      8              NaN
 |      dtype: object
 |      >>> s.interpolate(method='pad', limit=2)
 |      0              NaN
 |      1       single_one
 |      2       single_one
 |      3    fill_two_more
 |      4    fill_two_more
 |      5    fill_two_more
 |      6              NaN
 |      7             4.71
 |      8             4.71
 |      dtype: object
 |
 |      Filling in ``NaN`` in a Series via polynomial interpolation or splines:
 |      Both 'polynomial' and 'spline' methods require that you also specify
 |      an ``order`` (int).
 |
 |      >>> s = pd.Series([0, 2, np.nan, 8])
 |      >>> s.interpolate(method='polynomial', order=2)
 |      0    0.000000
 |      1    2.000000
 |      2    4.666667
 |      3    8.000000
 |      dtype: float64
 |
 |      Fill the DataFrame forward (that is, going down) along each column
 |      using linear interpolation.
 |
 |      Note how the last entry in column 'a' is interpolated differently,
 |      because there is no entry after it to use for interpolation.
 |      Note how the first entry in column 'b' remains ``NaN``, because there
 |      is no entry before it to use for interpolation.
 |
 |      >>> df = pd.DataFrame([(0.0, np.nan, -1.0, 1.0),
 |      ...                    (np.nan, 2.0, np.nan, np.nan),
 |      ...                    (2.0, 3.0, np.nan, 9.0),
 |      ...                    (np.nan, 4.0, -4.0, 16.0)],
 |      ...                   columns=list('abcd'))
 |      >>> df
 |           a    b    c     d
 |      0  0.0  NaN -1.0   1.0
 |      1  NaN  2.0  NaN   NaN
 |      2  2.0  3.0  NaN   9.0
 |      3  NaN  4.0 -4.0  16.0
 |      >>> df.interpolate(method='linear', limit_direction='forward', axis=0)
 |           a    b    c     d
 |      0  0.0  NaN -1.0   1.0
 |      1  1.0  2.0 -2.0   5.0
 |      2  2.0  3.0 -3.0   9.0
 |      3  2.0  4.0 -4.0  16.0
 |
 |      Using polynomial interpolation.
 |
 |      >>> df['d'].interpolate(method='polynomial', order=2)
 |      0     1.0
 |      1     4.0
 |      2     9.0
 |      3    16.0
 |      Name: d, dtype: float64
 |
 |  isin(self, values) -> 'DataFrame'
 |      Whether each element in the DataFrame is contained in values.
 |
 |      Parameters
 |      ----------
 |      values : iterable, Series, DataFrame or dict
 |          The result will only be true at a location if all the
 |          labels match. If `values` is a Series, that's the index. If
 |          `values` is a dict, the keys must be the column names,
 |          which must match. If `values` is a DataFrame,
 |          then both the index and column labels must match.
 |
 |      Returns
 |      -------
 |      DataFrame
 |          DataFrame of booleans showing whether each element in the DataFrame
 |          is contained in values.
 |
 |      See Also
 |      --------
 |      DataFrame.eq: Equality test for DataFrame.
 |      Series.isin: Equivalent method on Series.
 |      Series.str.contains: Test if pattern or regex is contained within a
 |          string of a Series or Index.
 |
 |      Examples
 |      --------
 |      >>> df = pd.DataFrame({'num_legs': [2, 4], 'num_wings': [2, 0]},
 |      ...                   index=['falcon', 'dog'])
 |      >>> df
 |              num_legs  num_wings
 |      falcon         2          2
 |      dog            4          0
 |
 |      When ``values`` is a list check whether every value in the DataFrame
 |      is present in the list (which animals have 0 or 2 legs or wings)
 |
 |      >>> df.isin([0, 2])
 |              num_legs  num_wings
 |      falcon      True       True
 |      dog        False       True
 |
 |      When ``values`` is a dict, we can pass values to check for each
 |      column separately:
 |
 |      >>> df.isin({'num_wings': [0, 3]})
 |              num_legs  num_wings
 |      falcon     False      False
 |      dog        False       True
 |
 |      When ``values`` is a Series or DataFrame the index and column must
 |      match. Note that 'falcon' does not match based on the number of legs
 |      in df2.
 |
 |      >>> other = pd.DataFrame({'num_legs': [8, 2], 'num_wings': [0, 2]},
 |      ...                      index=['spider', 'falcon'])
 |      >>> df.isin(other)
 |              num_legs  num_wings
 |      falcon      True       True
 |      dog        False      False
 |
 |  isna(self) -> 'DataFrame'
 |      Detect missing values.
 |
 |      Return a boolean same-sized object indicating if the values are NA.
 |      NA values, such as None or :attr:`numpy.NaN`, gets mapped to True
 |      values.
 |      Everything else gets mapped to False values. Characters such as empty
 |      strings ``''`` or :attr:`numpy.inf` are not considered NA values
 |      (unless you set ``pandas.options.mode.use_inf_as_na = True``).
 |
 |      Returns
 |      -------
 |      DataFrame
 |          Mask of bool values for each element in DataFrame that
 |          indicates whether an element is an NA value.
 |
 |      See Also
 |      --------
 |      DataFrame.isnull : Alias of isna.
 |      DataFrame.notna : Boolean inverse of isna.
 |      DataFrame.dropna : Omit axes labels with missing values.
 |      isna : Top-level isna.
 |
 |      Examples
 |      --------
 |      Show which entries in a DataFrame are NA.
 |
 |      >>> df = pd.DataFrame(dict(age=[5, 6, np.NaN],
 |      ...                    born=[pd.NaT, pd.Timestamp('1939-05-27'),
 |      ...                          pd.Timestamp('1940-04-25')],
 |      ...                    name=['Alfred', 'Batman', ''],
 |      ...                    toy=[None, 'Batmobile', 'Joker']))
 |      >>> df
 |         age       born    name        toy
 |      0  5.0        NaT  Alfred       None
 |      1  6.0 1939-05-27  Batman  Batmobile
 |      2  NaN 1940-04-25              Joker
 |
 |      >>> df.isna()
 |           age   born   name    toy
 |      0  False   True  False   True
 |      1  False  False  False  False
 |      2   True  False  False  False
 |
 |      Show which entries in a Series are NA.
 |
 |      >>> ser = pd.Series([5, 6, np.NaN])
 |      >>> ser
 |      0    5.0
 |      1    6.0
 |      2    NaN
 |      dtype: float64
 |
 |      >>> ser.isna()
 |      0    False
 |      1    False
 |      2     True
 |      dtype: bool
 |
 |  isnull(self) -> 'DataFrame'
 |      Detect missing values.
 |
 |      Return a boolean same-sized object indicating if the values are NA.
 |      NA values, such as None or :attr:`numpy.NaN`, gets mapped to True
 |      values.
 |      Everything else gets mapped to False values. Characters such as empty
 |      strings ``''`` or :attr:`numpy.inf` are not considered NA values
 |      (unless you set ``pandas.options.mode.use_inf_as_na = True``).
 |
 |      Returns
 |      -------
 |      DataFrame
 |          Mask of bool values for each element in DataFrame that
 |          indicates whether an element is an NA value.
 |
 |      See Also
 |      --------
 |      DataFrame.isnull : Alias of isna.
 |      DataFrame.notna : Boolean inverse of isna.
 |      DataFrame.dropna : Omit axes labels with missing values.
 |      isna : Top-level isna.
 |
 |      Examples
 |      --------
 |      Show which entries in a DataFrame are NA.
 |
 |      >>> df = pd.DataFrame(dict(age=[5, 6, np.NaN],
 |      ...                    born=[pd.NaT, pd.Timestamp('1939-05-27'),
 |      ...                          pd.Timestamp('1940-04-25')],
 |      ...                    name=['Alfred', 'Batman', ''],
 |      ...                    toy=[None, 'Batmobile', 'Joker']))
 |      >>> df
 |         age       born    name        toy
 |      0  5.0        NaT  Alfred       None
 |      1  6.0 1939-05-27  Batman  Batmobile
 |      2  NaN 1940-04-25              Joker
 |
 |      >>> df.isna()
 |           age   born   name    toy
 |      0  False   True  False   True
 |      1  False  False  False  False
 |      2   True  False  False  False
 |
 |      Show which entries in a Series are NA.
 |
 |      >>> ser = pd.Series([5, 6, np.NaN])
 |      >>> ser
 |      0    5.0
 |      1    6.0
 |      2    NaN
 |      dtype: float64
 |
 |      >>> ser.isna()
 |      0    False
 |      1    False
 |      2     True
 |      dtype: bool
 |
 |  items(self) -> 'Iterable[tuple[Hashable, Series]]'
 |      Iterate over (column name, Series) pairs.
 |
 |      Iterates over the DataFrame columns, returning a tuple with
 |      the column name and the content as a Series.
 |
 |      Yields
 |      ------
 |      label : object
 |          The column names for the DataFrame being iterated over.
 |      content : Series
 |          The column entries belonging to each label, as a Series.
 |
 |      See Also
 |      --------
 |      DataFrame.iterrows : Iterate over DataFrame rows as
 |          (index, Series) pairs.
 |      DataFrame.itertuples : Iterate over DataFrame rows as namedtuples
 |          of the values.
 |
 |      Examples
 |      --------
 |      >>> df = pd.DataFrame({'species': ['bear', 'bear', 'marsupial'],
 |      ...                   'population': [1864, 22000, 80000]},
 |      ...                   index=['panda', 'polar', 'koala'])
 |      >>> df
 |              species   population
 |      panda   bear      1864
 |      polar   bear      22000
 |      koala   marsupial 80000
 |      >>> for label, content in df.items():
 |      ...     print(f'label: {label}')
 |      ...     print(f'content: {content}', sep='\n')
 |      ...
 |      label: species
 |      content:
 |      panda         bear
 |      polar         bear
 |      koala    marsupial
 |      Name: species, dtype: object
 |      label: population
 |      content:
 |      panda     1864
 |      polar    22000
 |      koala    80000
 |      Name: population, dtype: int64
 |
 |  iteritems(self) -> 'Iterable[tuple[Hashable, Series]]'
 |      Iterate over (column name, Series) pairs.
 |
 |      Iterates over the DataFrame columns, returning a tuple with
 |      the column name and the content as a Series.
 |
 |      Yields
 |      ------
 |      label : object
 |          The column names for the DataFrame being iterated over.
 |      content : Series
 |          The column entries belonging to each label, as a Series.
 |
 |      See Also
 |      --------
 |      DataFrame.iterrows : Iterate over DataFrame rows as
 |          (index, Series) pairs.
 |      DataFrame.itertuples : Iterate over DataFrame rows as namedtuples
 |          of the values.
 |
 |      Examples
 |      --------
 |      >>> df = pd.DataFrame({'species': ['bear', 'bear', 'marsupial'],
 |      ...                   'population': [1864, 22000, 80000]},
 |      ...                   index=['panda', 'polar', 'koala'])
 |      >>> df
 |              species   population
 |      panda   bear      1864
 |      polar   bear      22000
 |      koala   marsupial 80000
 |      >>> for label, content in df.items():
 |      ...     print(f'label: {label}')
 |      ...     print(f'content: {content}', sep='\n')
 |      ...
 |      label: species
 |      content:
 |      panda         bear
 |      polar         bear
 |      koala    marsupial
 |      Name: species, dtype: object
 |      label: population
 |      content:
 |      panda     1864
 |      polar    22000
 |      koala    80000
 |      Name: population, dtype: int64
 |
 |  iterrows(self) -> 'Iterable[tuple[Hashable, Series]]'
 |      Iterate over DataFrame rows as (index, Series) pairs.
 |
 |      Yields
 |      ------
 |      index : label or tuple of label
 |          The index of the row. A tuple for a `MultiIndex`.
 |      data : Series
 |          The data of the row as a Series.
 |
 |      See Also
 |      --------
 |      DataFrame.itertuples : Iterate over DataFrame rows as namedtuples of the
values.
 |      DataFrame.items : Iterate over (column name, Series) pairs.
 |
 |      Notes
 |      -----
 |      1. Because ``iterrows`` returns a Series for each row,
 |         it does **not** preserve dtypes across the rows (dtypes are
 |         preserved across columns for DataFrames). For example,
 |
 |         >>> df = pd.DataFrame([[1, 1.5]], columns=['int', 'float'])
 |         >>> row = next(df.iterrows())[1]
 |         >>> row
 |         int      1.0
 |         float    1.5
 |         Name: 0, dtype: float64
 |         >>> print(row['int'].dtype)
 |         float64
 |         >>> print(df['int'].dtype)
 |         int64
 |
 |         To preserve dtypes while iterating over the rows, it is better
 |         to use :meth:`itertuples` which returns namedtuples of the values
 |         and which is generally faster than ``iterrows``.
 |
 |      2. You should **never modify** something you are iterating over.
 |         This is not guaranteed to work in all cases. Depending on the
 |         data types, the iterator returns a copy and not a view, and writing
 |         to it will have no effect.
 |
 |  itertuples(self, index: 'bool' = True, name: 'str | None' = 'Pandas') ->
'Iterable[tuple[Any, ...]]'
 |      Iterate over DataFrame rows as namedtuples.
 |
 |      Parameters
 |      ----------
 |      index : bool, default True
 |          If True, return the index as the first element of the tuple.
 |      name : str or None, default "Pandas"
 |          The name of the returned namedtuples or None to return regular
 |          tuples.
 |
 |      Returns
 |      -------
 |      iterator
 |          An object to iterate over namedtuples for each row in the
 |          DataFrame with the first field possibly being the index and
 |          following fields being the column values.
 |
 |      See Also
 |      --------
 |      DataFrame.iterrows : Iterate over DataFrame rows as (index, Series)
 |          pairs.
 |      DataFrame.items : Iterate over (column name, Series) pairs.
 |
 |      Notes
 |      -----
 |      The column names will be renamed to positional names if they are
 |      invalid Python identifiers, repeated, or start with an underscore.
 |      On python versions < 3.7 regular tuples are returned for DataFrames
 |      with a large number of columns (>254).
 |
 |      Examples
 |      --------
 |      >>> df = pd.DataFrame({'num_legs': [4, 2], 'num_wings': [0, 2]},
 |      ...                   index=['dog', 'hawk'])
 |      >>> df
 |            num_legs  num_wings
 |      dog          4          0
 |      hawk         2          2
 |      >>> for row in df.itertuples():
 |      ...     print(row)
 |      ...
 |      Pandas(Index='dog', num_legs=4, num_wings=0)
 |      Pandas(Index='hawk', num_legs=2, num_wings=2)
 |
 |      By setting the `index` parameter to False we can remove the index
 |      as the first element of the tuple:
 |
 |      >>> for row in df.itertuples(index=False):
 |      ...     print(row)
 |      ...
 |      Pandas(num_legs=4, num_wings=0)
 |      Pandas(num_legs=2, num_wings=2)
 |
 |      With the `name` parameter set we set a custom name for the yielded
 |      namedtuples:
 |
 |      >>> for row in df.itertuples(name='Animal'):
 |      ...     print(row)
 |      ...
 |      Animal(Index='dog', num_legs=4, num_wings=0)
 |      Animal(Index='hawk', num_legs=2, num_wings=2)
 |
 |  join(self, other: 'FrameOrSeriesUnion', on: 'IndexLabel | None' = None, how:
'str' = 'left', lsuffix: 'str' = '', rsuffix: 'str' = '', sort: 'bool' = False) ->
'DataFrame'
 |      Join columns of another DataFrame.
 |
 |      Join columns with `other` DataFrame either on index or on a key
 |      column. Efficiently join multiple DataFrame objects by index at once by
 |      passing a list.
 |
 |      Parameters
 |      ----------
 |      other : DataFrame, Series, or list of DataFrame
 |          Index should be similar to one of the columns in this one. If a
 |          Series is passed, its name attribute must be set, and that will be
 |          used as the column name in the resulting joined DataFrame.
 |      on : str, list of str, or array-like, optional
 |          Column or index level name(s) in the caller to join on the index
 |          in `other`, otherwise joins index-on-index. If multiple
 |          values given, the `other` DataFrame must have a MultiIndex. Can
 |          pass an array as the join key if it is not already contained in
 |          the calling DataFrame. Like an Excel VLOOKUP operation.
 |      how : {'left', 'right', 'outer', 'inner'}, default 'left'
 |          How to handle the operation of the two objects.
 |
 |          * left: use calling frame's index (or column if on is specified)
 |          * right: use `other`'s index.
 |          * outer: form union of calling frame's index (or column if on is
 |            specified) with `other`'s index, and sort it.
 |            lexicographically.
 |          * inner: form intersection of calling frame's index (or column if
 |            on is specified) with `other`'s index, preserving the order
 |            of the calling's one.
 |      lsuffix : str, default ''
 |          Suffix to use from left frame's overlapping columns.
 |      rsuffix : str, default ''
 |          Suffix to use from right frame's overlapping columns.
 |      sort : bool, default False
 |          Order result DataFrame lexicographically by the join key. If False,
 |          the order of the join key depends on the join type (how keyword).
 |
 |      Returns
 |      -------
 |      DataFrame
 |          A dataframe containing columns from both the caller and `other`.
 |
 |      See Also
 |      --------
 |      DataFrame.merge : For column(s)-on-column(s) operations.
 |
 |      Notes
 |      -----
 |      Parameters `on`, `lsuffix`, and `rsuffix` are not supported when
 |      passing a list of `DataFrame` objects.
 |
 |      Support for specifying index levels as the `on` parameter was added
 |      in version 0.23.0.
 |
 |      Examples
 |      --------
 |      >>> df = pd.DataFrame({'key': ['K0', 'K1', 'K2', 'K3', 'K4', 'K5'],
 |      ...                    'A': ['A0', 'A1', 'A2', 'A3', 'A4', 'A5']})
 |
 |      >>> df
 |        key   A
 |      0  K0  A0
 |      1  K1  A1
 |      2  K2  A2
 |      3  K3  A3
 |      4  K4  A4
 |      5  K5  A5
 |
 |      >>> other = pd.DataFrame({'key': ['K0', 'K1', 'K2'],
 |      ...                       'B': ['B0', 'B1', 'B2']})
 |
 |      >>> other
 |        key   B
 |      0  K0  B0
 |      1  K1  B1
 |      2  K2  B2
 |      
 |      Join DataFrames using their indexes.
 |      
 |      >>> df.join(other, lsuffix='_caller', rsuffix='_other')
 |        key_caller   A key_other    B
 |      0         K0  A0        K0   B0
 |      1         K1  A1        K1   B1
 |      2         K2  A2        K2   B2
 |      3         K3  A3       NaN  NaN
 |      4         K4  A4       NaN  NaN
 |      5         K5  A5       NaN  NaN
 |      
 |      If we want to join using the key columns, we need to set key to be
 |      the index in both `df` and `other`. The joined DataFrame will have
 |      key as its index.
 |      
 |      >>> df.set_index('key').join(other.set_index('key'))
 |            A    B
 |      key
 |      K0   A0   B0
 |      K1   A1   B1
 |      K2   A2   B2
 |      K3   A3  NaN
 |      K4   A4  NaN
 |      K5   A5  NaN
 |      
 |      Another option to join using the key columns is to use the `on`
 |      parameter. DataFrame.join always uses `other`'s index but we can use
 |      any column in `df`. This method preserves the original DataFrame's
 |      index in the result.
 |      
 |      >>> df.join(other.set_index('key'), on='key')
 |        key   A    B
 |      0  K0  A0   B0
 |      1  K1  A1   B1
 |      2  K2  A2   B2
 |      3  K3  A3  NaN
 |      4  K4  A4  NaN
 |      5  K5  A5  NaN
 |  
 |  kurt(self, axis=None, skipna=None, level=None, numeric_only=None, **kwargs)
 |      Return unbiased kurtosis over requested axis.
 |      
 |      Kurtosis obtained using Fisher's definition of
 |      kurtosis (kurtosis of normal == 0.0). Normalized by N-1.
 |      
 |      Parameters
 |      ----------
 |      axis : {index (0), columns (1)}
 |          Axis for the function to be applied on.
 |      skipna : bool, default True
 |          Exclude NA/null values when computing the result.
 |      level : int or level name, default None
 |          If the axis is a MultiIndex (hierarchical), count along a
 |          particular level, collapsing into a Series.
 |      numeric_only : bool, default None
 |          Include only float, int, boolean columns. If None, will attempt to use
 |          everything, then use only numeric data. Not implemented for Series.
 |      **kwargs
 |          Additional keyword arguments to be passed to the function.
 |      
 |      Returns
 |      -------
 |      Series or DataFrame (if level specified)
 |  
 |  kurtosis = kurt(self, axis=None, skipna=None, level=None, numeric_only=None,
**kwargs)
 |  
 |  le(self, other, axis='columns', level=None)
 |      Get Less than or equal to of dataframe and other, element-wise (binary
operator `le`).
 |      
 |      Among flexible wrappers (`eq`, `ne`, `le`, `lt`, `ge`, `gt`) to comparison
 |      operators.
 |      
 |      Equivalent to `==`, `!=`, `<=`, `<`, `>=`, `>` with support to choose axis
 |      (rows or columns) and level for comparison.
 |      
 |      Parameters
 |      ----------
 |      other : scalar, sequence, Series, or DataFrame
 |          Any single or multiple element data structure, or list-like object.
 |      axis : {0 or 'index', 1 or 'columns'}, default 'columns'
 |          Whether to compare by the index (0 or 'index') or columns
 |          (1 or 'columns').
 |      level : int or label
 |          Broadcast across a level, matching Index values on the passed
 |          MultiIndex level.
 |
 |      Returns
 |      -------
 |      DataFrame of bool
 |          Result of the comparison.
 |
 |      See Also
 |      --------
 |      DataFrame.eq : Compare DataFrames for equality elementwise.
 |      DataFrame.ne : Compare DataFrames for inequality elementwise.
 |      DataFrame.le : Compare DataFrames for less than inequality
 |          or equality elementwise.
 |      DataFrame.lt : Compare DataFrames for strictly less than
 |          inequality elementwise.
 |      DataFrame.ge : Compare DataFrames for greater than inequality
 |          or equality elementwise.
 |      DataFrame.gt : Compare DataFrames for strictly greater than
 |          inequality elementwise.
 |
 |      Notes
 |      -----
 |      Mismatched indices will be unioned together.
 |      `NaN` values are considered different (i.e. `NaN` != `NaN`).
 |
 |      Examples
 |      --------
 |      >>> df = pd.DataFrame({'cost': [250, 150, 100],
 |      ...                    'revenue': [100, 250, 300]},
 |      ...                   index=['A', 'B', 'C'])
 |      >>> df
 |         cost  revenue
 |      A   250      100
 |      B   150      250
 |      C   100      300
 |
 |      Comparison with a scalar, using either the operator or method:
 |
 |      >>> df == 100
 |          cost  revenue
 |      A  False     True
 |      B  False    False
 |      C   True    False
 |
 |      >>> df.eq(100)
 |          cost  revenue
 |      A  False     True
 |      B  False    False
 |      C   True    False
 |
 |      When `other` is a :class:`Series`, the columns of a DataFrame are aligned
 |      with the index of `other` and broadcast:
 |
 |      >>> df != pd.Series([100, 250], index=["cost", "revenue"])
 |          cost  revenue
 |      A   True     True
 |      B   True    False
 |      C  False     True
 |
 |      Use the method to control the broadcast axis:
 |
 |      >>> df.ne(pd.Series([100, 300], index=["A", "D"]), axis='index')
 |         cost  revenue
 |      A  True    False
 |      B  True     True
 |      C  True     True
 |      D  True     True
 |
 |      When comparing to an arbitrary sequence, the number of columns must
 |      match the number elements in `other`:
 |
 |      >>> df == [250, 100]
 |          cost  revenue
 |      A   True     True
 |      B  False    False
 |      C  False    False
 |
 |      Use the method to control the axis:
 |
 |      >>> df.eq([250, 250, 100], axis='index')
 |          cost  revenue
 |      A   True    False
```

```
 |      B  False     True
 |      C   True    False
 |
 |      Compare to a DataFrame of different shape.
 |
 |      >>> other = pd.DataFrame({'revenue': [300, 250, 100, 150]},
 |      ...                      index=['A', 'B', 'C', 'D'])
 |      >>> other
 |         revenue
 |      A      300
 |      B      250
 |      C      100
 |      D      150
 |
 |      >>> df.gt(other)
 |          cost  revenue
 |      A  False    False
 |      B  False    False
 |      C  False     True
 |      D  False    False
 |
 |      Compare to a MultiIndex by level.
 |
 |      >>> df_multindex = pd.DataFrame({'cost': [250, 150, 100, 150, 300, 220],
 |      ...                              'revenue': [100, 250, 300, 200, 175, 225]},
 |      ...                             index=[['Q1', 'Q1', 'Q1', 'Q2', 'Q2', 'Q2'],
 |      ...                                    ['A', 'B', 'C', 'A', 'B', 'C']])
 |      >>> df_multindex
 |            cost  revenue
 |      Q1 A   250      100
 |         B   150      250
 |         C   100      300
 |      Q2 A   150      200
 |         B   300      175
 |         C   220      225
 |
 |      >>> df.le(df_multindex, level=1)
 |             cost  revenue
 |      Q1 A   True     True
 |         B   True     True
 |         C   True     True
 |      Q2 A  False     True
 |         B   True    False
 |         C   True    False
 |
 |  lookup(self, row_labels: 'Sequence[IndexLabel]', col_labels:
'Sequence[IndexLabel]') -> 'np.ndarray'
 |      Label-based "fancy indexing" function for DataFrame.
 |      Given equal-length arrays of row and column labels, return an
 |      array of the values corresponding to each (row, col) pair.
 |
 |      .. deprecated:: 1.2.0
 |          DataFrame.lookup is deprecated,
 |          use DataFrame.melt and DataFrame.loc instead.
 |          For further details see
 |          :ref:`Looking up values by index/column labels <indexing.lookup>`.
 |
 |      Parameters
 |      ----------
 |      row_labels : sequence
 |          The row labels to use for lookup.
 |      col_labels : sequence
 |          The column labels to use for lookup.
 |
 |      Returns
 |      -------
 |      numpy.ndarray
 |          The found values.
 |
 |  lt(self, other, axis='columns', level=None)
 |      Get Less than of dataframe and other, element-wise (binary operator `lt`).
 |
 |      Among flexible wrappers (`eq`, `ne`, `le`, `lt`, `ge`, `gt`) to comparison
 |      operators.
 |
 |      Equivalent to `==`, `!=`, `<=`, `<`, `>=`, `>` with support to choose axis
 |      (rows or columns) and level for comparison.
 |
 |      Parameters
 |      ----------
 |      other : scalar, sequence, Series, or DataFrame
 |          Any single or multiple element data structure, or list-like object.
 |      axis : {0 or 'index', 1 or 'columns'}, default 'columns'
 |          Whether to compare by the index (0 or 'index') or columns
 |          (1 or 'columns').
 |      level : int or label
 |          Broadcast across a level, matching Index values on the passed
 |          MultiIndex level.
 |
 |      Returns
 |      -------
 |      DataFrame of bool
 |          Result of the comparison.
 |
 |      See Also
 |      --------
 |      DataFrame.eq : Compare DataFrames for equality elementwise.
 |      DataFrame.ne : Compare DataFrames for inequality elementwise.
 |      DataFrame.le : Compare DataFrames for less than inequality
 |          or equality elementwise.
 |      DataFrame.lt : Compare DataFrames for strictly less than
 |          inequality elementwise.
 |      DataFrame.ge : Compare DataFrames for greater than inequality
 |          or equality elementwise.
 |      DataFrame.gt : Compare DataFrames for strictly greater than
 |          inequality elementwise.
 |
 |      Notes
 |      -----
 |      Mismatched indices will be unioned together.
 |      `NaN` values are considered different (i.e. `NaN` != `NaN`).
 |
 |      Examples
 |      --------
 |      >>> df = pd.DataFrame({'cost': [250, 150, 100],
 |      ...                    'revenue': [100, 250, 300]},
 |      ...                   index=['A', 'B', 'C'])
 |      >>> df
 |         cost  revenue
 |      A   250      100
 |      B   150      250
 |      C   100      300
 |
 |      Comparison with a scalar, using either the operator or method:
 |
 |      >>> df == 100
 |          cost  revenue
 |      A  False     True
 |      B  False    False
 |      C   True    False
 |
 |      >>> df.eq(100)
 |          cost  revenue
 |      A  False     True
 |      B  False    False
 |      C   True    False
 |
 |      When `other` is a :class:`Series`, the columns of a DataFrame are aligned
 |      with the index of `other` and broadcast:
 |
 |      >>> df != pd.Series([100, 250], index=["cost", "revenue"])
 |          cost  revenue
 |      A   True     True
 |      B   True    False
 |      C  False     True
 |
 |      Use the method to control the broadcast axis:
 |
 |      >>> df.ne(pd.Series([100, 300], index=["A", "D"]), axis='index')
 |         cost  revenue
 |      A  True    False
 |      B  True     True
 |      C  True     True
 |      D  True     True
 |
 |      When comparing to an arbitrary sequence, the number of columns must
 |      match the number elements in `other`:
 |
 |      >>> df == [250, 100]
 |          cost  revenue
 |      A   True     True
 |      B  False    False
 |      C  False    False
 |
 |      Use the method to control the axis:
 |
 |      >>> df.eq([250, 250, 100], axis='index')
 |          cost  revenue
 |      A   True    False
 |      B  False     True
 |      C   True    False
 |
 |      Compare to a DataFrame of different shape.
 |
 |      >>> other = pd.DataFrame({'revenue': [300, 250, 100, 150]},
 |      ...                      index=['A', 'B', 'C', 'D'])
 |      >>> other
 |         revenue
 |      A      300
 |      B      250
 |      C      100
 |      D      150
 |
 |      >>> df.gt(other)
 |          cost  revenue
 |      A  False    False
 |      B  False    False
 |      C  False     True
 |      D  False    False
 |
 |      Compare to a MultiIndex by level.
 |
 |      >>> df_multiindex = pd.DataFrame({'cost': [250, 150, 100, 150, 300, 220],
 |      ...                              'revenue': [100, 250, 300, 200, 175, 225]},
 |      ...                             index=[['Q1', 'Q1', 'Q1', 'Q2', 'Q2', 'Q2'],
 |      ...                                    ['A', 'B', 'C', 'A', 'B', 'C']])
 |      >>> df_multiindex
 |            cost  revenue
 |      Q1 A   250      100
 |         B   150      250
 |         C   100      300
 |      Q2 A   150      200
 |         B   300      175
 |         C   220      225
 |
 |      >>> df.le(df_multiindex, level=1)
 |             cost  revenue
 |      Q1 A   True     True
 |         B   True     True
 |         C   True     True
 |      Q2 A  False     True
 |         B   True    False
 |         C   True    False
 |
 |  mad(self, axis=None, skipna=None, level=None)
 |      Return the mean absolute deviation of the values over the requested axis.
 |
 |      Parameters
 |      ----------
 |      axis : {index (0), columns (1)}
 |          Axis for the function to be applied on.
 |      skipna : bool, default None
 |          Exclude NA/null values when computing the result.
 |      level : int or level name, default None
 |          If the axis is a MultiIndex (hierarchical), count along a
 |          particular level, collapsing into a Series.
 |
 |      Returns
 |      -------
 |      Series or DataFrame (if level specified)
 |
 |  mask(self, cond, other=nan, inplace=False, axis=None, level=None, errors='raise',
try_cast=<no_default>)
 |      Replace values where the condition is True.
 |
 |      Parameters
 |      ----------
 |      cond : bool Series/DataFrame, array-like, or callable
 |          Where `cond` is False, keep the original value. Where
 |          True, replace with corresponding value from `other`.
 |          If `cond` is callable, it is computed on the Series/DataFrame and
 |          should return boolean Series/DataFrame or array. The callable must
 |          not change input Series/DataFrame (though pandas doesn't check it).
 |      other : scalar, Series/DataFrame, or callable
 |          Entries where `cond` is True are replaced with
 |          corresponding value from `other`.
 |          If other is callable, it is computed on the Series/DataFrame and
 |          should return scalar or Series/DataFrame. The callable must not
 |          change input Series/DataFrame (though pandas doesn't check it).
 |      inplace : bool, default False
 |          Whether to perform the operation in place on the data.
 |      axis : int, default None
 |          Alignment axis if needed.
 |      level : int, default None
 |          Alignment level if needed.
 |      errors : str, {'raise', 'ignore'}, default 'raise'
 |          Note that currently this parameter won't affect
 |          the results and will always coerce to a suitable dtype.
 |
 |          - 'raise' : allow exceptions to be raised.
 |          - 'ignore' : suppress exceptions. On error return original object.
```

```
|
|      try_cast : bool, default None
|          Try to cast the result back to the input type (if possible).
|
|          .. deprecated:: 1.3.0
|              Manually cast back if necessary.
|
|      Returns
|      -------
|      Same type as caller or None if ``inplace=True``.
|
|      See Also
|      --------
|      :func:`DataFrame.where` : Return an object of same shape as
|          self.
|
|      Notes
|      -----
|      The mask method is an application of the if-then idiom. For each
|      element in the calling DataFrame, if ``cond`` is ``False`` the
|      element is used; otherwise the corresponding element from the DataFrame
|      ``other`` is used.
|
|      The signature for :func:`DataFrame.where` differs from
|      :func:`numpy.where`. Roughly ``df1.where(m, df2)`` is equivalent to
|      ``np.where(m, df1, df2)``.
|
|      For further details and examples see the ``mask`` documentation in
|      :ref:`indexing <indexing.where_mask>`.
|
|      Examples
|      --------
|      >>> s = pd.Series(range(5))
|      >>> s.where(s > 0)
|      0    NaN
|      1    1.0
|      2    2.0
|      3    3.0
|      4    4.0
|      dtype: float64
|      >>> s.mask(s > 0)
|      0    0.0
|      1    NaN
|      2    NaN
|      3    NaN
|      4    NaN
|      dtype: float64
|
|      >>> s.where(s > 1, 10)
|      0    10
|      1    10
|      2    2
|      3    3
|      4    4
|      dtype: int64
|      >>> s.mask(s > 1, 10)
|      0     0
|      1     1
|      2    10
|      3    10
|      4    10
|      dtype: int64
|
|      >>> df = pd.DataFrame(np.arange(10).reshape(-1, 2), columns=['A', 'B'])
|      >>> df
|         A  B
|      0  0  1
|      1  2  3
|      2  4  5
|      3  6  7
|      4  8  9
|      >>> m = df % 3 == 0
|      >>> df.where(m, -df)
|         A  B
|      0  0 -1
|      1 -2  3
|      2 -4 -5
|      3  6 -7
|      4 -8  9
|      >>> df.where(m, -df) == np.where(m, df, -df)
|            A     B
|      0  True  True
|      1  True  True
|      2  True  True
|      3  True  True
|      4  True  True
|      >>> df.where(m, -df) == df.mask(~m, -df)
```

```
 |             A     B
 |       0  True  True
 |       1  True  True
 |       2  True  True
 |       3  True  True
 |       4  True  True
 |
 |   max(self, axis=None, skipna=None, level=None, numeric_only=None, **kwargs)
 |       Return the maximum of the values over the requested axis.
 |
 |       If you want the *index* of the maximum, use ``idxmax``. This is the
equivalent of the ``numpy.ndarray`` method ``argmax``.
 |
 |       Parameters
 |       ----------
 |       axis : {index (0), columns (1)}
 |           Axis for the function to be applied on.
 |       skipna : bool, default True
 |           Exclude NA/null values when computing the result.
 |       level : int or level name, default None
 |           If the axis is a MultiIndex (hierarchical), count along a
 |           particular level, collapsing into a Series.
 |       numeric_only : bool, default None
 |           Include only float, int, boolean columns. If None, will attempt to use
 |           everything, then use only numeric data. Not implemented for Series.
 |       **kwargs
 |           Additional keyword arguments to be passed to the function.
 |
 |       Returns
 |       -------
 |       Series or DataFrame (if level specified)
 |
 |       See Also
 |       --------
 |       Series.sum : Return the sum.
 |       Series.min : Return the minimum.
 |       Series.max : Return the maximum.
 |       Series.idxmin : Return the index of the minimum.
 |       Series.idxmax : Return the index of the maximum.
 |       DataFrame.sum : Return the sum over the requested axis.
 |       DataFrame.min : Return the minimum over the requested axis.
 |       DataFrame.max : Return the maximum over the requested axis.
 |       DataFrame.idxmin : Return the index of the minimum over the requested axis.
 |       DataFrame.idxmax : Return the index of the maximum over the requested axis.
 |
 |       Examples
 |       --------
 |       >>> idx = pd.MultiIndex.from_arrays([
 |       ...     ['warm', 'warm', 'cold', 'cold'],
 |       ...     ['dog', 'falcon', 'fish', 'spider']],
 |       ...     names=['blooded', 'animal'])
 |       >>> s = pd.Series([4, 2, 0, 8], name='legs', index=idx)
 |       >>> s
 |       blooded  animal
 |       warm     dog       4
 |                falcon    2
 |       cold     fish      0
 |                spider    8
 |       Name: legs, dtype: int64
 |
 |       >>> s.max()
 |       8
 |
 |   mean(self, axis=None, skipna=None, level=None, numeric_only=None, **kwargs)
 |       Return the mean of the values over the requested axis.
 |
 |       Parameters
 |       ----------
 |       axis : {index (0), columns (1)}
 |           Axis for the function to be applied on.
 |       skipna : bool, default True
 |           Exclude NA/null values when computing the result.
 |       level : int or level name, default None
 |           If the axis is a MultiIndex (hierarchical), count along a
 |           particular level, collapsing into a Series.
 |       numeric_only : bool, default None
 |           Include only float, int, boolean columns. If None, will attempt to use
 |           everything, then use only numeric data. Not implemented for Series.
 |       **kwargs
 |           Additional keyword arguments to be passed to the function.
 |
 |       Returns
 |       -------
 |       Series or DataFrame (if level specified)
 |
 |   median(self, axis=None, skipna=None, level=None, numeric_only=None, **kwargs)
 |       Return the median of the values over the requested axis.
 |
 |      Parameters
 |      ----------
 |      axis : {index (0), columns (1)}
 |          Axis for the function to be applied on.
 |      skipna : bool, default True
 |          Exclude NA/null values when computing the result.
 |      level : int or level name, default None
 |          If the axis is a MultiIndex (hierarchical), count along a
 |          particular level, collapsing into a Series.
 |      numeric_only : bool, default None
 |          Include only float, int, boolean columns. If None, will attempt to use
 |          everything, then use only numeric data. Not implemented for Series.
 |      **kwargs
 |          Additional keyword arguments to be passed to the function.
 |
 |      Returns
 |      -------
 |      Series or DataFrame (if level specified)
 |
 |  melt(self, id_vars=None, value_vars=None, var_name=None, value_name='value',
col_level: 'Level | None' = None, ignore_index: 'bool' = True) -> 'DataFrame'
 |      Unpivot a DataFrame from wide to long format, optionally leaving identifiers
set.
 |
 |      This function is useful to massage a DataFrame into a format where one
 |      or more columns are identifier variables (`id_vars`), while all other
 |      columns, considered measured variables (`value_vars`), are "unpivoted" to
 |      the row axis, leaving just two non-identifier columns, 'variable' and
 |      'value'.
 |
 |      Parameters
 |      ----------
 |      id_vars : tuple, list, or ndarray, optional
 |          Column(s) to use as identifier variables.
 |      value_vars : tuple, list, or ndarray, optional
 |          Column(s) to unpivot. If not specified, uses all columns that
 |          are not set as `id_vars`.
 |      var_name : scalar
 |          Name to use for the 'variable' column. If None it uses
 |          ``frame.columns.name`` or 'variable'.
 |      value_name : scalar, default 'value'
 |          Name to use for the 'value' column.
 |      col_level : int or str, optional
 |          If columns are a MultiIndex then use this level to melt.
 |      ignore_index : bool, default True
 |          If True, original index is ignored. If False, the original index is
retained.
 |          Index labels will be repeated as necessary.
 |
 |          .. versionadded:: 1.1.0
 |
 |      Returns
 |      -------
 |      DataFrame
 |          Unpivoted DataFrame.
 |
 |      See Also
 |      --------
 |      melt : Identical method.
 |      pivot_table : Create a spreadsheet-style pivot table as a DataFrame.
 |      DataFrame.pivot : Return reshaped DataFrame organized
 |          by given index / column values.
 |      DataFrame.explode : Explode a DataFrame from list-like
 |              columns to long format.
 |
 |      Examples
 |      --------
 |      >>> df = pd.DataFrame({'A': {0: 'a', 1: 'b', 2: 'c'},
 |      ...                    'B': {0: 1, 1: 3, 2: 5},
 |      ...                    'C': {0: 2, 1: 4, 2: 6}})
 |      >>> df
 |         A  B  C
 |      0  a  1  2
 |      1  b  3  4
 |      2  c  5  6
 |
 |      >>> df.melt(id_vars=['A'], value_vars=['B'])
 |         A variable  value
 |      0  a        B      1
 |      1  b        B      3
 |      2  c        B      5
 |
 |      >>> df.melt(id_vars=['A'], value_vars=['B', 'C'])
 |         A variable  value
 |      0  a        B      1
 |      1  b        B      3
 |      2  c        B      5
 |      3  a        C      2
 |      4  b        C      4
 |      5  c        C      6
 |
 |      The names of 'variable' and 'value' columns can be customized:
 |
 |      >>> df.melt(id_vars=['A'], value_vars=['B'],
 |      ...         var_name='myVarname', value_name='myValname')
 |         A myVarname  myValname
 |      0  a         B          1
 |      1  b         B          3
 |      2  c         B          5
 |
 |      Original index values can be kept around:
 |
 |      >>> df.melt(id_vars=['A'], value_vars=['B', 'C'], ignore_index=False)
 |         A variable  value
 |      0  a        B      1
 |      1  b        B      3
 |      2  c        B      5
 |      0  a        C      2
 |      1  b        C      4
 |      2  c        C      6
 |
 |      If you have multi-index columns:
 |
 |      >>> df.columns = [list('ABC'), list('DEF')]
 |      >>> df
 |         A  B  C
 |         D  E  F
 |      0  a  1  2
 |      1  b  3  4
 |      2  c  5  6
 |
 |      >>> df.melt(col_level=0, id_vars=['A'], value_vars=['B'])
 |         A variable  value
 |      0  a        B      1
 |      1  b        B      3
 |      2  c        B      5
 |
 |      >>> df.melt(id_vars=[('A', 'D')], value_vars=[('B', 'E')])
 |        (A, D) variable_0 variable_1  value
 |      0      a          B          E      1
 |      1      b          B          E      3
 |      2      c          B          E      5
 |
 |  memory_usage(self, index: 'bool' = True, deep: 'bool' = False) -> 'Series'
 |      Return the memory usage of each column in bytes.
 |
 |      The memory usage can optionally include the contribution of
 |      the index and elements of `object` dtype.
 |
 |      This value is displayed in `DataFrame.info` by default. This can be
 |      suppressed by setting ``pandas.options.display.memory_usage`` to False.
 |
 |      Parameters
 |      ----------
 |      index : bool, default True
 |          Specifies whether to include the memory usage of the DataFrame's
 |          index in returned Series. If ``index=True``, the memory usage of
 |          the index is the first item in the output.
 |      deep : bool, default False
 |          If True, introspect the data deeply by interrogating
 |          `object` dtypes for system-level memory consumption, and include
 |          it in the returned values.
 |
 |      Returns
 |      -------
 |      Series
 |          A Series whose index is the original column names and whose values
 |          is the memory usage of each column in bytes.
 |
 |      See Also
 |      --------
 |      numpy.ndarray.nbytes : Total bytes consumed by the elements of an
 |          ndarray.
 |      Series.memory_usage : Bytes consumed by a Series.
 |      Categorical : Memory-efficient array for string values with
 |          many repeated values.
 |      DataFrame.info : Concise summary of a DataFrame.
 |
 |      Examples
 |      --------
 |      >>> dtypes = ['int64', 'float64', 'complex128', 'object', 'bool']
 |      >>> data = dict([(t, np.ones(shape=5000, dtype=int).astype(t))
 |      ...              for t in dtypes])
```

```
 |      >>> df = pd.DataFrame(data)
 |      >>> df.head()
 |         int64  float64            complex128  object  bool
 |      0      1      1.0              1.0+0.0j       1  True
 |      1      1      1.0              1.0+0.0j       1  True
 |      2      1      1.0              1.0+0.0j       1  True
 |      3      1      1.0              1.0+0.0j       1  True
 |      4      1      1.0              1.0+0.0j       1  True
 |
 |      >>> df.memory_usage()
 |      Index           128
 |      int64         40000
 |      float64       40000
 |      complex128    80000
 |      object        40000
 |      bool           5000
 |      dtype: int64
 |
 |      >>> df.memory_usage(index=False)
 |      int64         40000
 |      float64       40000
 |      complex128    80000
 |      object        40000
 |      bool           5000
 |      dtype: int64
 |
 |      The memory footprint of `object` dtype columns is ignored by default:
 |
 |      >>> df.memory_usage(deep=True)
 |      Index            128
 |      int64          40000
 |      float64        40000
 |      complex128     80000
 |      object        180000
 |      bool            5000
 |      dtype: int64
 |
 |      Use a Categorical for efficient storage of an object-dtype column with
 |      many repeated values.
 |
 |      >>> df['object'].astype('category').memory_usage(deep=True)
 |      5244
 |
 |  merge(self, right: 'FrameOrSeriesUnion', how: 'str' = 'inner', on: 'IndexLabel |
None' = None, left_on: 'IndexLabel | None' = None, right_on: 'IndexLabel | None' =
None, left_index: 'bool' = False, right_index: 'bool' = False, sort: 'bool' = False,
suffixes: 'Suffixes' = ('_x', '_y'), copy: 'bool' = True, indicator: 'bool' = False,
validate: 'str | None' = None) -> 'DataFrame'
 |      Merge DataFrame or named Series objects with a database-style join.
 |
 |      A named Series object is treated as a DataFrame with a single named column.
 |
 |      The join is done on columns or indexes. If joining columns on
 |      columns, the DataFrame indexes *will be ignored*. Otherwise if joining
indexes
 |      on indexes or indexes on a column or columns, the index will be passed on.
 |      When performing a cross merge, no column specifications to merge on are
 |      allowed.
 |
 |      Parameters
 |      ----------
 |      right : DataFrame or named Series
 |          Object to merge with.
 |      how : {'left', 'right', 'outer', 'inner', 'cross'}, default 'inner'
 |          Type of merge to be performed.
 |
 |          * left: use only keys from left frame, similar to a SQL left outer join;
 |            preserve key order.
 |          * right: use only keys from right frame, similar to a SQL right outer
join;
 |            preserve key order.
 |          * outer: use union of keys from both frames, similar to a SQL full outer
 |            join; sort keys lexicographically.
 |          * inner: use intersection of keys from both frames, similar to a SQL
inner
 |            join; preserve the order of the left keys.
 |          * cross: creates the cartesian product from both frames, preserves the
order
 |            of the left keys.
 |
 |            .. versionadded:: 1.2.0
 |
 |      on : label or list
 |          Column or index level names to join on. These must be found in both
 |          DataFrames. If `on` is None and not merging on indexes then this defaults
 |          to the intersection of the columns in both DataFrames.
 |      left_on : label or list, or array-like
 |          Column or index level names to join on in the left DataFrame. Can also
 |          be an array or list of arrays of the length of the left DataFrame.
 |          These arrays are treated as if they are columns.
 |      right_on : label or list, or array-like
 |          Column or index level names to join on in the right DataFrame. Can also
 |          be an array or list of arrays of the length of the right DataFrame.
 |          These arrays are treated as if they are columns.
 |      left_index : bool, default False
 |          Use the index from the left DataFrame as the join key(s). If it is a
 |          MultiIndex, the number of keys in the other DataFrame (either the index
 |          or a number of columns) must match the number of levels.
 |      right_index : bool, default False
 |          Use the index from the right DataFrame as the join key. Same caveats as
 |          left_index.
 |      sort : bool, default False
 |          Sort the join keys lexicographically in the result DataFrame. If False,
 |          the order of the join keys depends on the join type (how keyword).
 |      suffixes : list-like, default is ("_x", "_y")
 |          A length-2 sequence where each element is optionally a string
 |          indicating the suffix to add to overlapping column names in
 |          `left` and `right` respectively. Pass a value of `None` instead
 |          of a string to indicate that the column name from `left` or
 |          `right` should be left as-is, with no suffix. At least one of the
 |          values must not be None.
 |      copy : bool, default True
 |          If False, avoid copy if possible.
 |      indicator : bool or str, default False
 |          If True, adds a column to the output DataFrame called "_merge" with
 |          information on the source of each row. The column can be given a
different
 |          name by providing a string argument. The column will have a Categorical
 |          type with the value of "left_only" for observations whose merge key only
 |          appears in the left DataFrame, "right_only" for observations
 |          whose merge key only appears in the right DataFrame, and "both"
 |          if the observation's merge key is found in both DataFrames.
 |
 |      validate : str, optional
 |          If specified, checks if merge is of specified type.
 |
 |          * "one_to_one" or "1:1": check if merge keys are unique in both
 |            left and right datasets.
 |          * "one_to_many" or "1:m": check if merge keys are unique in left
 |            dataset.
 |          * "many_to_one" or "m:1": check if merge keys are unique in right
 |            dataset.
 |          * "many_to_many" or "m:m": allowed, but does not result in checks.
 |
 |      Returns
 |      -------
 |      DataFrame
 |          A DataFrame of the two merged objects.
 |
 |      See Also
 |      --------
 |      merge_ordered : Merge with optional filling/interpolation.
 |      merge_asof : Merge on nearest keys.
 |      DataFrame.join : Similar method using indices.
 |
 |      Notes
 |      -----
 |      Support for specifying index levels as the `on`, `left_on`, and
 |      `right_on` parameters was added in version 0.23.0
 |      Support for merging named Series objects was added in version 0.24.0
 |
 |      Examples
 |      --------
 |      >>> df1 = pd.DataFrame({'lkey': ['foo', 'bar', 'baz', 'foo'],
 |      ...                     'value': [1, 2, 3, 5]})
 |      >>> df2 = pd.DataFrame({'rkey': ['foo', 'bar', 'baz', 'foo'],
 |      ...                     'value': [5, 6, 7, 8]})
 |      >>> df1
 |          lkey value
 |      0   foo      1
 |      1   bar      2
 |      2   baz      3
 |      3   foo      5
 |      >>> df2
 |          rkey value
 |      0   foo      5
 |      1   bar      6
 |      2   baz      7
 |      3   foo      8
 |
 |      Merge df1 and df2 on the lkey and rkey columns. The value columns have
 |      the default suffixes, _x and _y, appended.
 |
 |      >>> df1.merge(df2, left_on='lkey', right_on='rkey')
 |        lkey  value_x rkey  value_y
 |      0  foo        1  foo        5
 |      1  foo        1  foo        8
 |      2  foo        5  foo        5
 |      3  foo        5  foo        8
 |      4  bar        2  bar        6
 |      5  baz        3  baz        7
 |
 |      Merge DataFrames df1 and df2 with specified left and right suffixes
 |      appended to any overlapping columns.
 |
 |      >>> df1.merge(df2, left_on='lkey', right_on='rkey',
 |      ...           suffixes=('_left', '_right'))
 |        lkey  value_left rkey  value_right
 |      0  foo           1  foo            5
 |      1  foo           1  foo            8
 |      2  foo           5  foo            5
 |      3  foo           5  foo            8
 |      4  bar           2  bar            6
 |      5  baz           3  baz            7
 |
 |      Merge DataFrames df1 and df2, but raise an exception if the DataFrames have
 |      any overlapping columns.
 |
 |      >>> df1.merge(df2, left_on='lkey', right_on='rkey', suffixes=(False, False))
 |      Traceback (most recent call last):
 |      ...
 |      ValueError: columns overlap but no suffix specified:
 |          Index(['value'], dtype='object')
 |
 |      >>> df1 = pd.DataFrame({'a': ['foo', 'bar'], 'b': [1, 2]})
 |      >>> df2 = pd.DataFrame({'a': ['foo', 'baz'], 'c': [3, 4]})
 |      >>> df1
 |            a  b
 |      0   foo  1
 |      1   bar  2
 |      >>> df2
 |            a  c
 |      0   foo  3
 |      1   baz  4
 |
 |      >>> df1.merge(df2, how='inner', on='a')
 |            a  b  c
 |      0   foo  1  3
 |
 |      >>> df1.merge(df2, how='left', on='a')
 |            a  b  c
 |      0   foo  1  3.0
 |      1   bar  2  NaN
 |
 |      >>> df1 = pd.DataFrame({'left': ['foo', 'bar']})
 |      >>> df2 = pd.DataFrame({'right': [7, 8]})
 |      >>> df1
 |          left
 |      0   foo
 |      1   bar
 |      >>> df2
 |          right
 |      0   7
 |      1   8
 |
 |      >>> df1.merge(df2, how='cross')
 |         left  right
 |      0   foo      7
 |      1   foo      8
 |      2   bar      7
 |      3   bar      8
 |
 |  min(self, axis=None, skipna=None, level=None, numeric_only=None, **kwargs)
 |      Return the minimum of the values over the requested axis.
 |
 |      If you want the *index* of the minimum, use ``idxmin``. This is the
equivalent of the ``numpy.ndarray`` method ``argmin``.
 |
 |      Parameters
 |      ----------
 |      axis : {index (0), columns (1)}
 |          Axis for the function to be applied on.
 |      skipna : bool, default True
 |          Exclude NA/null values when computing the result.
 |      level : int or level name, default None
 |          If the axis is a MultiIndex (hierarchical), count along a
 |          particular level, collapsing into a Series.
 |      numeric_only : bool, default None
 |          Include only float, int, boolean columns. If None, will attempt to use
 |          everything, then use only numeric data. Not implemented for Series.
 |      **kwargs
 |          Additional keyword arguments to be passed to the function.
 |
 |      Returns
 |      -------
 |      Series or DataFrame (if level specified)
 |
 |      See Also
 |      --------
 |      Series.sum : Return the sum.
 |      Series.min : Return the minimum.
 |      Series.max : Return the maximum.
 |      Series.idxmin : Return the index of the minimum.
 |      Series.idxmax : Return the index of the maximum.
 |      DataFrame.sum : Return the sum over the requested axis.
 |      DataFrame.min : Return the minimum over the requested axis.
 |      DataFrame.max : Return the maximum over the requested axis.
 |      DataFrame.idxmin : Return the index of the minimum over the requested axis.
 |      DataFrame.idxmax : Return the index of the maximum over the requested axis.
 |
 |      Examples
 |      --------
 |      >>> idx = pd.MultiIndex.from_arrays([
 |      ...     ['warm', 'warm', 'cold', 'cold'],
 |      ...     ['dog', 'falcon', 'fish', 'spider']],
 |      ...     names=['blooded', 'animal'])
 |      >>> s = pd.Series([4, 2, 0, 8], name='legs', index=idx)
 |      >>> s
 |      blooded  animal
 |      warm     dog       4
 |               falcon    2
 |      cold     fish      0
 |               spider    8
 |      Name: legs, dtype: int64
 |
 |      >>> s.min()
 |      0
 |
 |  mod(self, other, axis='columns', level=None, fill_value=None)
 |      Get Modulo of dataframe and other, element-wise (binary operator `mod`).
 |
 |      Equivalent to ``dataframe % other``, but with support to substitute a
fill_value
 |      for missing data in one of the inputs. With reverse version, `rmod`.
 |
 |      Among flexible wrappers (`add`, `sub`, `mul`, `div`, `mod`, `pow`) to
 |      arithmetic operators: `+`, `-`, `*`, `/`, `//`, `%`, `**`.
 |
 |      Parameters
 |      ----------
 |      other : scalar, sequence, Series, or DataFrame
 |          Any single or multiple element data structure, or list-like object.
 |      axis : {0 or 'index', 1 or 'columns'}
 |          Whether to compare by the index (0 or 'index') or columns
 |          (1 or 'columns'). For Series input, axis to match Series index on.
 |      level : int or label
 |          Broadcast across a level, matching Index values on the
 |          passed MultiIndex level.
 |      fill_value : float or None, default None
 |          Fill existing missing (NaN) values, and any new element needed for
 |          successful DataFrame alignment, with this value before computation.
 |          If data in both corresponding DataFrame locations is missing
 |          the result will be missing.
 |
 |      Returns
 |      -------
 |      DataFrame
 |          Result of the arithmetic operation.
 |
 |      See Also
 |      --------
 |      DataFrame.add : Add DataFrames.
 |      DataFrame.sub : Subtract DataFrames.
 |      DataFrame.mul : Multiply DataFrames.
 |      DataFrame.div : Divide DataFrames (float division).
 |      DataFrame.truediv : Divide DataFrames (float division).
 |      DataFrame.floordiv : Divide DataFrames (integer division).
 |      DataFrame.mod : Calculate modulo (remainder after division).
 |      DataFrame.pow : Calculate exponential power.
 |
 |      Notes
 |      -----
 |      Mismatched indices will be unioned together.
 |
 |      Examples
 |      --------
 |      >>> df = pd.DataFrame({'angles': [0, 3, 4],
 |      ...                    'degrees': [360, 180, 360]},
 |      ...                   index=['circle', 'triangle', 'rectangle'])
 |      >>> df
 |                 angles  degrees
 |      circle          0      360
 |      triangle        3      180
 |      rectangle       4      360
 |
 |      Add a scalar with operator version which return the same
 |      results.
 |
 |      >>> df + 1
 |                 angles  degrees
 |      circle          1      361
 |      triangle        4      181
 |      rectangle       5      361
 |
 |      >>> df.add(1)
 |                 angles  degrees
 |      circle          1      361
 |      triangle        4      181
 |      rectangle       5      361
 |
 |      Divide by constant with reverse version.
 |
 |      >>> df.div(10)
 |                 angles  degrees
 |      circle        0.0     36.0
 |      triangle      0.3     18.0
 |      rectangle     0.4     36.0
 |
 |      >>> df.rdiv(10)
 |                   angles   degrees
 |      circle          inf  0.027778
 |      triangle   3.333333  0.055556
 |      rectangle  2.500000  0.027778
 |
 |      Subtract a list and Series by axis with operator version.
 |
 |      >>> df - [1, 2]
 |                 angles  degrees
 |      circle         -1      358
 |      triangle        2      178
 |      rectangle       3      358
 |
 |      >>> df.sub([1, 2], axis='columns')
 |                 angles  degrees
 |      circle         -1      358
 |      triangle        2      178
 |      rectangle       3      358
 |
 |      >>> df.sub(pd.Series([1, 1, 1], index=['circle', 'triangle', 'rectangle']),
 |      ...        axis='index')
 |                 angles  degrees
 |      circle         -1      359
 |      triangle        2      179
 |      rectangle       3      359
 |
 |      Multiply a DataFrame of different shape with operator version.
 |
 |      >>> other = pd.DataFrame({'angles': [0, 3, 4]},
 |      ...                      index=['circle', 'triangle', 'rectangle'])
 |      >>> other
 |                 angles
 |      circle          0
 |      triangle        3
 |      rectangle       4
 |
 |      >>> df * other
 |                 angles  degrees
 |      circle          0      NaN
 |      triangle        9      NaN
 |      rectangle      16      NaN
 |
 |      >>> df.mul(other, fill_value=0)
 |                 angles  degrees
 |      circle          0      0.0
 |      triangle        9      0.0
 |      rectangle      16      0.0
 |
 |      Divide by a MultiIndex by level.
 |
 |      >>> df_multiindex = pd.DataFrame({'angles': [0, 3, 4, 4, 5, 6],
 |      ...                              'degrees': [360, 180, 360, 360, 540, 720]},
 |      ...                             index=[['A', 'A', 'A', 'B', 'B', 'B'],
 |      ...                                    ['circle', 'triangle', 'rectangle',
 |      ...                                     'square', 'pentagon', 'hexagon']])
 |      >>> df_multiindex
```

```
 |                   angles  degrees
 |      A circle          0      360
 |        triangle        3      180
 |        rectangle       4      360
 |      B square          4      360
 |        pentagon        5      540
 |        hexagon         6      720
 |
 |      >>> df.div(df_multindex, level=1, fill_value=0)
 |                   angles  degrees
 |      A circle        NaN      1.0
 |        triangle      1.0      1.0
 |        rectangle     1.0      1.0
 |      B square        0.0      0.0
 |        pentagon      0.0      0.0
 |        hexagon       0.0      0.0
 |
 |  mode(self, axis: 'Axis' = 0, numeric_only: 'bool' = False, dropna: 'bool' = True)
-> 'DataFrame'
 |      Get the mode(s) of each element along the selected axis.
 |
 |      The mode of a set of values is the value that appears most often.
 |      It can be multiple values.
 |
 |      Parameters
 |      ----------
 |      axis : {0 or 'index', 1 or 'columns'}, default 0
 |          The axis to iterate over while searching for the mode:
 |
 |          * 0 or 'index' : get mode of each column
 |          * 1 or 'columns' : get mode of each row.
 |
 |      numeric_only : bool, default False
 |          If True, only apply to numeric columns.
 |      dropna : bool, default True
 |          Don't consider counts of NaN/NaT.
 |
 |      Returns
 |      -------
 |      DataFrame
 |          The modes of each column or row.
 |
 |      See Also
 |      --------
 |      Series.mode : Return the highest frequency value in a Series.
 |      Series.value_counts : Return the counts of values in a Series.
 |
 |      Examples
 |      --------
 |      >>> df = pd.DataFrame([('bird', 2, 2),
 |      ...                    ('mammal', 4, np.nan),
 |      ...                    ('arthropod', 8, 0),
 |      ...                    ('bird', 2, np.nan)],
 |      ...                   index=('falcon', 'horse', 'spider', 'ostrich'),
 |      ...                   columns=('species', 'legs', 'wings'))
 |      >>> df
 |                 species  legs  wings
 |      falcon        bird     2    2.0
 |      horse       mammal     4    NaN
 |      spider   arthropod     8    0.0
 |      ostrich       bird     2    NaN
 |
 |      By default, missing values are not considered, and the mode of wings
 |      are both 0 and 2. Because the resulting DataFrame has two rows,
 |      the second row of ``species`` and ``legs`` contains ``NaN``.
 |
 |      >>> df.mode()
 |        species  legs  wings
 |      0    bird   2.0    0.0
 |      1     NaN   NaN    2.0
 |
 |      Setting ``dropna=False`` ``NaN`` values are considered and they can be
 |      the mode (like for wings).
 |
 |      >>> df.mode(dropna=False)
 |        species  legs  wings
 |      0    bird     2    NaN
 |
 |      Setting ``numeric_only=True``, only the mode of numeric columns is
 |      computed, and columns of other types are ignored.
 |
 |      >>> df.mode(numeric_only=True)
 |         legs  wings
 |      0   2.0    0.0
 |      1   NaN    2.0
 |
 |      To compute the mode over columns and not rows, use the axis parameter:
```

```
 |
 |      >>> df.mode(axis='columns', numeric_only=True)
 |                 0    1
 |      falcon   2.0  NaN
 |      horse    4.0  NaN
 |      spider   0.0  8.0
 |      ostrich  2.0  NaN
 |
 |  mul(self, other, axis='columns', level=None, fill_value=None)
 |      Get Multiplication of dataframe and other, element-wise (binary operator
`mul`).
 |
 |      Equivalent to ``dataframe * other``, but with support to substitute a
fill_value
 |      for missing data in one of the inputs. With reverse version, `rmul`.
 |
 |      Among flexible wrappers (`add`, `sub`, `mul`, `div`, `mod`, `pow`) to
 |      arithmetic operators: `+`, `-`, `*`, `/`, `//`, `%`, `**`.
 |
 |      Parameters
 |      ----------
 |      other : scalar, sequence, Series, or DataFrame
 |          Any single or multiple element data structure, or list-like object.
 |      axis : {0 or 'index', 1 or 'columns'}
 |          Whether to compare by the index (0 or 'index') or columns
 |          (1 or 'columns'). For Series input, axis to match Series index on.
 |      level : int or label
 |          Broadcast across a level, matching Index values on the
 |          passed MultiIndex level.
 |      fill_value : float or None, default None
 |          Fill existing missing (NaN) values, and any new element needed for
 |          successful DataFrame alignment, with this value before computation.
 |          If data in both corresponding DataFrame locations is missing
 |          the result will be missing.
 |
 |      Returns
 |      -------
 |      DataFrame
 |          Result of the arithmetic operation.
 |
 |      See Also
 |      --------
 |      DataFrame.add : Add DataFrames.
 |      DataFrame.sub : Subtract DataFrames.
 |      DataFrame.mul : Multiply DataFrames.
 |      DataFrame.div : Divide DataFrames (float division).
 |      DataFrame.truediv : Divide DataFrames (float division).
 |      DataFrame.floordiv : Divide DataFrames (integer division).
 |      DataFrame.mod : Calculate modulo (remainder after division).
 |      DataFrame.pow : Calculate exponential power.
 |
 |      Notes
 |      -----
 |      Mismatched indices will be unioned together.
 |
 |      Examples
 |      --------
 |      >>> df = pd.DataFrame({'angles': [0, 3, 4],
 |      ...                    'degrees': [360, 180, 360]},
 |      ...                   index=['circle', 'triangle', 'rectangle'])
 |      >>> df
 |                 angles  degrees
 |      circle          0      360
 |      triangle        3      180
 |      rectangle       4      360
 |
 |      Add a scalar with operator version which return the same
 |      results.
 |
 |      >>> df + 1
 |                 angles  degrees
 |      circle          1      361
 |      triangle        4      181
 |      rectangle       5      361
 |
 |      >>> df.add(1)
 |                 angles  degrees
 |      circle          1      361
 |      triangle        4      181
 |      rectangle       5      361
 |
 |      Divide by constant with reverse version.
 |
 |      >>> df.div(10)
 |                 angles  degrees
 |      circle        0.0     36.0
 |      triangle      0.3     18.0
 |      rectangle     0.4     36.0
 |
 |      >>> df.rdiv(10)
 |                   angles   degrees
 |      circle          inf  0.027778
 |      triangle   3.333333  0.055556
 |      rectangle  2.500000  0.027778
 |
 |      Subtract a list and Series by axis with operator version.
 |
 |      >>> df - [1, 2]
 |                 angles  degrees
 |      circle         -1      358
 |      triangle        2      178
 |      rectangle       3      358
 |
 |      >>> df.sub([1, 2], axis='columns')
 |                 angles  degrees
 |      circle         -1      358
 |      triangle        2      178
 |      rectangle       3      358
 |
 |      >>> df.sub(pd.Series([1, 1, 1], index=['circle', 'triangle', 'rectangle']),
 |      ...        axis='index')
 |                 angles  degrees
 |      circle         -1      359
 |      triangle        2      179
 |      rectangle       3      359
 |
 |      Multiply a DataFrame of different shape with operator version.
 |
 |      >>> other = pd.DataFrame({'angles': [0, 3, 4]},
 |      ...                      index=['circle', 'triangle', 'rectangle'])
 |      >>> other
 |                 angles
 |      circle          0
 |      triangle        3
 |      rectangle       4
 |
 |      >>> df * other
 |                 angles  degrees
 |      circle          0      NaN
 |      triangle        9      NaN
 |      rectangle      16      NaN
 |
 |      >>> df.mul(other, fill_value=0)
 |                 angles  degrees
 |      circle          0      0.0
 |      triangle        9      0.0
 |      rectangle      16      0.0
 |
 |      Divide by a MultiIndex by level.
 |
 |      >>> df_multindex = pd.DataFrame({'angles': [0, 3, 4, 4, 5, 6],
 |      ...                              'degrees': [360, 180, 360, 360, 540, 720]},
 |      ...                             index=[['A', 'A', 'A', 'B', 'B', 'B'],
 |      ...                                    ['circle', 'triangle', 'rectangle',
 |      ...                                     'square', 'pentagon', 'hexagon']])
 |      >>> df_multindex
 |                   angles  degrees
 |      A circle          0      360
 |        triangle        3      180
 |        rectangle       4      360
 |      B square          4      360
 |        pentagon        5      540
 |        hexagon         6      720
 |
 |      >>> df.div(df_multindex, level=1, fill_value=0)
 |                   angles  degrees
 |      A circle        NaN      1.0
 |        triangle      1.0      1.0
 |        rectangle     1.0      1.0
 |      B square        0.0      0.0
 |        pentagon      0.0      0.0
 |        hexagon       0.0      0.0
 |
 |  multiply = mul(self, other, axis='columns', level=None, fill_value=None)
 |
 |  ne(self, other, axis='columns', level=None)
 |      Get Not equal to of dataframe and other, element-wise (binary operator `ne`).
 |
 |      Among flexible wrappers (`eq`, `ne`, `le`, `lt`, `ge`, `gt`) to comparison
 |      operators.
 |
 |      Equivalent to `==`, `!=`, `<=`, `<`, `>=`, `>` with support to choose axis
 |      (rows or columns) and level for comparison.
 |
 |      Parameters
 |      ----------
 |      other : scalar, sequence, Series, or DataFrame
 |          Any single or multiple element data structure, or list-like object.
 |      axis : {0 or 'index', 1 or 'columns'}, default 'columns'
 |          Whether to compare by the index (0 or 'index') or columns
 |          (1 or 'columns').
 |      level : int or label
 |          Broadcast across a level, matching Index values on the passed
 |          MultiIndex level.
 |
 |      Returns
 |      -------
 |      DataFrame of bool
 |          Result of the comparison.
 |
 |      See Also
 |      --------
 |      DataFrame.eq : Compare DataFrames for equality elementwise.
 |      DataFrame.ne : Compare DataFrames for inequality elementwise.
 |      DataFrame.le : Compare DataFrames for less than inequality
 |          or equality elementwise.
 |      DataFrame.lt : Compare DataFrames for strictly less than
 |          inequality elementwise.
 |      DataFrame.ge : Compare DataFrames for greater than inequality
 |          or equality elementwise.
 |      DataFrame.gt : Compare DataFrames for strictly greater than
 |          inequality elementwise.
 |
 |      Notes
 |      -----
 |      Mismatched indices will be unioned together.
 |      `NaN` values are considered different (i.e. `NaN` != `NaN`).
 |
 |      Examples
 |      --------
 |      >>> df = pd.DataFrame({'cost': [250, 150, 100],
 |      ...                    'revenue': [100, 250, 300]},
 |      ...                   index=['A', 'B', 'C'])
 |      >>> df
 |         cost  revenue
 |      A   250      100
 |      B   150      250
 |      C   100      300
 |
 |      Comparison with a scalar, using either the operator or method:
 |
 |      >>> df == 100
 |          cost  revenue
 |      A  False     True
 |      B  False    False
 |      C   True    False
 |
 |      >>> df.eq(100)
 |          cost  revenue
 |      A  False     True
 |      B  False    False
 |      C   True    False
 |
 |      When `other` is a :class:`Series`, the columns of a DataFrame are aligned
 |      with the index of `other` and broadcast:
 |
 |      >>> df != pd.Series([100, 250], index=["cost", "revenue"])
 |          cost  revenue
 |      A   True     True
 |      B   True    False
 |      C  False     True
 |
 |      Use the method to control the broadcast axis:
 |
 |      >>> df.ne(pd.Series([100, 300], index=["A", "D"]), axis='index')
 |         cost  revenue
 |      A  True    False
 |      B  True     True
 |      C  True     True
 |      D  True     True
 |
 |      When comparing to an arbitrary sequence, the number of columns must
 |      match the number elements in `other`:
 |
 |      >>> df == [250, 100]
 |          cost  revenue
 |      A   True     True
 |      B  False    False
 |      C  False    False
 |
 |      Use the method to control the axis:
```

```
|
|      >>> df.eq([250, 250, 100], axis='index')
|          cost  revenue
|      A   True    False
|      B  False     True
|      C   True    False
|
|      Compare to a DataFrame of different shape.
|
|      >>> other = pd.DataFrame({'revenue': [300, 250, 100, 150]},
|      ...                      index=['A', 'B', 'C', 'D'])
|      >>> other
|         revenue
|      A      300
|      B      250
|      C      100
|      D      150
|
|      >>> df.gt(other)
|          cost  revenue
|      A  False    False
|      B  False    False
|      C  False     True
|      D  False    False
|
|      Compare to a MultiIndex by level.
|
|      >>> df_multindex = pd.DataFrame({'cost': [250, 150, 100, 150, 300, 220],
|      ...                              'revenue': [100, 250, 300, 200, 175, 225]},
|      ...                             index=[['Q1', 'Q1', 'Q1', 'Q2', 'Q2', 'Q2'],
|      ...                                    ['A', 'B', 'C', 'A', 'B', 'C']])
|      >>> df_multindex
|            cost  revenue
|      Q1 A   250      100
|         B   150      250
|         C   100      300
|      Q2 A   150      200
|         B   300      175
|         C   220      225
|
|      >>> df.le(df_multindex, level=1)
|             cost  revenue
|      Q1 A   True     True
|         B   True     True
|         C   True     True
|      Q2 A  False     True
|         B   True    False
|         C   True    False
|
|  nlargest(self, n, columns, keep: 'str' = 'first') -> 'DataFrame'
|      Return the first `n` rows ordered by `columns` in descending order.
|
|      Return the first `n` rows with the largest values in `columns`, in
|      descending order. The columns that are not specified are returned as
|      well, but not used for ordering.
|
|      This method is equivalent to
|      ``df.sort_values(columns, ascending=False).head(n)``, but more
|      performant.
|
|      Parameters
|      ----------
|      n : int
|          Number of rows to return.
|      columns : label or list of labels
|          Column label(s) to order by.
|      keep : {'first', 'last', 'all'}, default 'first'
|          Where there are duplicate values:
|
|          - `first` : prioritize the first occurrence(s)
|          - `last` : prioritize the last occurrence(s)
|          - ``all`` : do not drop any duplicates, even it means
|                      selecting more than `n` items.
|
|      Returns
|      -------
|      DataFrame
|          The first `n` rows ordered by the given columns in descending
|          order.
|
|      See Also
|      --------
|      DataFrame.nsmallest : Return the first `n` rows ordered by `columns` in
|          ascending order.
|      DataFrame.sort_values : Sort DataFrame by the values.
|      DataFrame.head : Return the first `n` rows without re-ordering.
|
|      Notes
|      -----
|      This function cannot be used with all column types. For example, when
|      specifying columns with `object` or `category` dtypes, ``TypeError`` is
|      raised.
|
|      Examples
|      --------
|      >>> df = pd.DataFrame({'population': [59000000, 65000000, 434000,
|      ...                                   434000, 434000, 337000, 11300,
|      ...                                   11300, 11300],
|      ...                    'GDP': [1937894, 2583560 , 12011, 4520, 12128,
|      ...                            17036, 182, 38, 311],
|      ...                    'alpha-2': ["IT", "FR", "MT", "MV", "BN",
|      ...                                "IS", "NR", "TV", "AI"]},
|      ...                   index=["Italy", "France", "Malta",
|      ...                          "Maldives", "Brunei", "Iceland",
|      ...                          "Nauru", "Tuvalu", "Anguilla"])
|      >>> df
|                population      GDP alpha-2
|      Italy       59000000  1937894      IT
|      France      65000000  2583560      FR
|      Malta         434000    12011      MT
|      Maldives      434000     4520      MV
|      Brunei        434000    12128      BN
|      Iceland       337000    17036      IS
|      Nauru          11300      182      NR
|      Tuvalu         11300       38      TV
|      Anguilla       11300      311      AI
|
|      In the following example, we will use ``nlargest`` to select the three
|      rows having the largest values in column "population".
|
|      >>> df.nlargest(3, 'population')
|              population      GDP alpha-2
|      France    65000000  2583560      FR
|      Italy     59000000  1937894      IT
|      Malta       434000    12011      MT
|
|      When using ``keep='last'``, ties are resolved in reverse order:
|
|      >>> df.nlargest(3, 'population', keep='last')
|              population      GDP alpha-2
|      France    65000000  2583560      FR
|      Italy     59000000  1937894      IT
|      Brunei      434000    12128      BN
|
|      When using ``keep='all'``, all duplicate items are maintained:
|
|      >>> df.nlargest(3, 'population', keep='all')
|                population      GDP alpha-2
|      France      65000000  2583560      FR
|      Italy       59000000  1937894      IT
|      Malta         434000    12011      MT
|      Maldives      434000     4520      MV
|      Brunei        434000    12128      BN
|
|      To order by the largest values in column "population" and then "GDP",
|      we can specify multiple columns like in the next example.
|
|      >>> df.nlargest(3, ['population', 'GDP'])
|              population      GDP alpha-2
|      France    65000000  2583560      FR
|      Italy     59000000  1937894      IT
|      Brunei      434000    12128      BN
|
|  notna(self) -> 'DataFrame'
|      Detect existing (non-missing) values.
|
|      Return a boolean same-sized object indicating if the values are not NA.
|      Non-missing values get mapped to True. Characters such as empty
|      strings ``''`` or :attr:`numpy.inf` are not considered NA values
|      (unless you set ``pandas.options.mode.use_inf_as_na = True``).
|      NA values, such as None or :attr:`numpy.NaN`, get mapped to False
|      values.
|
|      Returns
|      -------
|      DataFrame
|          Mask of bool values for each element in DataFrame that
|          indicates whether an element is not an NA value.
|
|      See Also
|      --------
|      DataFrame.notnull : Alias of notna.
|      DataFrame.isna : Boolean inverse of notna.
|      DataFrame.dropna : Omit axes labels with missing values.
```

```
 |      notna : Top-level notna.
 |      
 |      Examples
 |      --------
 |      Show which entries in a DataFrame are not NA.
 |      
 |      >>> df = pd.DataFrame(dict(age=[5, 6, np.NaN],
 |      ...                    born=[pd.NaT, pd.Timestamp('1939-05-27'),
 |      ...                          pd.Timestamp('1940-04-25')],
 |      ...                    name=['Alfred', 'Batman', ''],
 |      ...                    toy=[None, 'Batmobile', 'Joker']))
 |      >>> df
 |         age       born    name        toy
 |      0  5.0        NaT  Alfred       None
 |      1  6.0 1939-05-27  Batman  Batmobile
 |      2  NaN 1940-04-25              Joker
 |      
 |      >>> df.notna()
 |           age   born  name    toy
 |      0   True  False  True  False
 |      1   True   True  True   True
 |      2  False   True  True   True
 |      
 |      Show which entries in a Series are not NA.
 |      
 |      >>> ser = pd.Series([5, 6, np.NaN])
 |      >>> ser
 |      0    5.0
 |      1    6.0
 |      2    NaN
 |      dtype: float64
 |      
 |      >>> ser.notna()
 |      0     True
 |      1     True
 |      2    False
 |      dtype: bool
 |  
 |  notnull(self) -> 'DataFrame'
 |      Detect existing (non-missing) values.
 |      
 |      Return a boolean same-sized object indicating if the values are not NA.
 |      Non-missing values get mapped to True. Characters such as empty
 |      strings ``''`` or :attr:`numpy.inf` are not considered NA values
 |      (unless you set ``pandas.options.mode.use_inf_as_na = True``).
 |      NA values, such as None or :attr:`numpy.NaN`, get mapped to False
 |      values.
 |      
 |      Returns
 |      -------
 |      DataFrame
 |          Mask of bool values for each element in DataFrame that
 |          indicates whether an element is not an NA value.
 |      
 |      See Also
 |      --------
 |      DataFrame.notnull : Alias of notna.
 |      DataFrame.isna : Boolean inverse of notna.
 |      DataFrame.dropna : Omit axes labels with missing values.
 |      notna : Top-level notna.
 |      
 |      Examples
 |      --------
 |      Show which entries in a DataFrame are not NA.
 |      
 |      >>> df = pd.DataFrame(dict(age=[5, 6, np.NaN],
 |      ...                    born=[pd.NaT, pd.Timestamp('1939-05-27'),
 |      ...                          pd.Timestamp('1940-04-25')],
 |      ...                    name=['Alfred', 'Batman', ''],
 |      ...                    toy=[None, 'Batmobile', 'Joker']))
 |      >>> df
 |         age       born    name        toy
 |      0  5.0        NaT  Alfred       None
 |      1  6.0 1939-05-27  Batman  Batmobile
 |      2  NaN 1940-04-25              Joker
 |      
 |      >>> df.notna()
 |           age   born  name    toy
 |      0   True  False  True  False
 |      1   True   True  True   True
 |      2  False   True  True   True
 |      
 |      Show which entries in a Series are not NA.
 |      
 |      >>> ser = pd.Series([5, 6, np.NaN])
 |      >>> ser
 |      0    5.0
```

```
|      1    6.0
|      2    NaN
|      dtype: float64
|
|      >>> ser.notna()
|      0     True
|      1     True
|      2    False
|      dtype: bool
|
|  nsmallest(self, n, columns, keep: 'str' = 'first') -> 'DataFrame'
|      Return the first `n` rows ordered by `columns` in ascending order.
|
|      Return the first `n` rows with the smallest values in `columns`, in
|      ascending order. The columns that are not specified are returned as
|      well, but not used for ordering.
|
|      This method is equivalent to
|      ``df.sort_values(columns, ascending=True).head(n)``, but more
|      performant.
|
|      Parameters
|      ----------
|      n : int
|          Number of items to retrieve.
|      columns : list or str
|          Column name or names to order by.
|      keep : {'first', 'last', 'all'}, default 'first'
|          Where there are duplicate values:
|
|          - ``first`` : take the first occurrence.
|          - ``last`` : take the last occurrence.
|          - ``all`` : do not drop any duplicates, even it means
|            selecting more than `n` items.
|
|      Returns
|      -------
|      DataFrame
|
|      See Also
|      --------
|      DataFrame.nlargest : Return the first `n` rows ordered by `columns` in
|          descending order.
|      DataFrame.sort_values : Sort DataFrame by the values.
|      DataFrame.head : Return the first `n` rows without re-ordering.
|
|      Examples
|      --------
|      >>> df = pd.DataFrame({'population': [59000000, 65000000, 434000,
|      ...                                   434000, 434000, 337000, 337000,
|      ...                                   11300, 11300],
|      ...                    'GDP': [1937894, 2583560 , 12011, 4520, 12128,
|      ...                            17036, 182, 38, 311],
|      ...                    'alpha-2': ["IT", "FR", "MT", "MV", "BN",
|      ...                                "IS", "NR", "TV", "AI"]},
|      ...                   index=["Italy", "France", "Malta",
|      ...                          "Maldives", "Brunei", "Iceland",
|      ...                          "Nauru", "Tuvalu", "Anguilla"])
|      >>> df
|                population      GDP alpha-2
|      Italy       59000000  1937894      IT
|      France      65000000  2583560      FR
|      Malta         434000    12011      MT
|      Maldives      434000     4520      MV
|      Brunei        434000    12128      BN
|      Iceland       337000    17036      IS
|      Nauru         337000      182      NR
|      Tuvalu         11300       38      TV
|      Anguilla       11300      311      AI
|
|      In the following example, we will use ``nsmallest`` to select the
|      three rows having the smallest values in column "population".
|
|      >>> df.nsmallest(3, 'population')
|                population    GDP alpha-2
|      Tuvalu         11300     38      TV
|      Anguilla       11300    311      AI
|      Iceland       337000  17036      IS
|
|      When using ``keep='last'``, ties are resolved in reverse order:
|
|      >>> df.nsmallest(3, 'population', keep='last')
|                population  GDP alpha-2
|      Anguilla       11300  311      AI
|      Tuvalu         11300   38      TV
|      Nauru         337000  182      NR
|
|      When using ``keep='all'``, all duplicate items are maintained:
|
|      >>> df.nsmallest(3, 'population', keep='all')
|                population    GDP alpha-2
|      Tuvalu         11300     38      TV
|      Anguilla       11300    311      AI
|      Iceland       337000  17036      IS
|      Nauru         337000    182      NR
|
|      To order by the smallest values in column "population" and then "GDP", we can
|      specify multiple columns like in the next example.
|
|      >>> df.nsmallest(3, ['population', 'GDP'])
|                population  GDP alpha-2
|      Tuvalu         11300   38      TV
|      Anguilla       11300  311      AI
|      Nauru         337000  182      NR
|
|  nunique(self, axis: 'Axis' = 0, dropna: 'bool' = True) -> 'Series'
|      Count number of distinct elements in specified axis.
|
|      Return Series with number of distinct elements. Can ignore NaN
|      values.
|
|      Parameters
|      ----------
|      axis : {0 or 'index', 1 or 'columns'}, default 0
|          The axis to use. 0 or 'index' for row-wise, 1 or 'columns' for
|          column-wise.
|      dropna : bool, default True
|          Don't include NaN in the counts.
|
|      Returns
|      -------
|      Series
|
|      See Also
|      --------
|      Series.nunique: Method nunique for Series.
|      DataFrame.count: Count non-NA cells for each column or row.
|
|      Examples
|      --------
|      >>> df = pd.DataFrame({'A': [4, 5, 6], 'B': [4, 1, 1]})
|      >>> df.nunique()
|      A    3
|      B    2
|      dtype: int64
|
|      >>> df.nunique(axis=1)
|      0    1
|      1    2
|      2    2
|      dtype: int64
|
|  pivot(self, index=None, columns=None, values=None) -> 'DataFrame'
|      Return reshaped DataFrame organized by given index / column values.
|
|      Reshape data (produce a "pivot" table) based on column values. Uses
|      unique values from specified `index` / `columns` to form axes of the
|      resulting DataFrame. This function does not support data
|      aggregation, multiple values will result in a MultiIndex in the
|      columns. See the :ref:`User Guide <reshaping>` for more on reshaping.
|
|      Parameters
|      ----------
|      index : str or object or a list of str, optional
|          Column to use to make new frame's index. If None, uses
|          existing index.
|
|          .. versionchanged:: 1.1.0
|             Also accept list of index names.
|
|      columns : str or object or a list of str
|          Column to use to make new frame's columns.
|
|          .. versionchanged:: 1.1.0
|             Also accept list of columns names.
|
|      values : str, object or a list of the previous, optional
|          Column(s) to use for populating new frame's values. If not
|          specified, all remaining columns will be used and the result will
|          have hierarchically indexed columns.
|
|      Returns
|      -------
|      DataFrame
|          Returns reshaped DataFrame.
|
|      Raises
|      ------
|      ValueError:
|          When there are any `index`, `columns` combinations with multiple
|          values. `DataFrame.pivot_table` when you need to aggregate.
|
|      See Also
|      --------
|      DataFrame.pivot_table : Generalization of pivot that can handle
|          duplicate values for one index/column pair.
|      DataFrame.unstack : Pivot based on the index values instead of a
|          column.
|      wide_to_long : Wide panel to long format. Less flexible but more
|          user-friendly than melt.
|
|      Notes
|      -----
|      For finer-tuned control, see hierarchical indexing documentation along
|      with the related stack/unstack methods.
|
|      Examples
|      --------
|      >>> df = pd.DataFrame({'foo': ['one', 'one', 'one', 'two', 'two',
|      ...                            'two'],
|      ...                    'bar': ['A', 'B', 'C', 'A', 'B', 'C'],
|      ...                    'baz': [1, 2, 3, 4, 5, 6],
|      ...                    'zoo': ['x', 'y', 'z', 'q', 'w', 't']})
|      >>> df
|          foo   bar  baz  zoo
|      0   one   A    1    x
|      1   one   B    2    y
|      2   one   C    3    z
|      3   two   A    4    q
|      4   two   B    5    w
|      5   two   C    6    t
|
|      >>> df.pivot(index='foo', columns='bar', values='baz')
|      bar  A   B   C
|      foo
|      one  1   2   3
|      two  4   5   6
|
|      >>> df.pivot(index='foo', columns='bar')['baz']
|      bar  A   B   C
|      foo
|      one  1   2   3
|      two  4   5   6
|
|      >>> df.pivot(index='foo', columns='bar', values=['baz', 'zoo'])
|            baz       zoo
|      bar   A  B  C   A  B  C
|      foo
|      one   1  2  3   x  y  z
|      two   4  5  6   q  w  t
|
|      You could also assign a list of column names or a list of index names.
|
|      >>> df = pd.DataFrame({
|      ...        "lev1": [1, 1, 1, 2, 2, 2],
|      ...        "lev2": [1, 1, 2, 1, 1, 2],
|      ...        "lev3": [1, 2, 1, 2, 1, 2],
|      ...        "lev4": [1, 2, 3, 4, 5, 6],
|      ...        "values": [0, 1, 2, 3, 4, 5]})
|      >>> df
|          lev1 lev2 lev3 lev4 values
|      0   1    1    1    1    0
|      1   1    1    2    2    1
|      2   1    2    1    3    2
|      3   2    1    2    4    3
|      4   2    1    1    5    4
|      5   2    2    2    6    5
|
|      >>> df.pivot(index="lev1", columns=["lev2", "lev3"],values="values")
|      lev2    1         2
|      lev3    1    2    1    2
|      lev1
|      1     0.0  1.0  2.0  NaN
|      2     4.0  3.0  NaN  5.0
|
|      >>> df.pivot(index=["lev1", "lev2"], columns=["lev3"],values="values")
|            lev3    1    2
|      lev1  lev2
|         1     1  0.0  1.0
|               2  2.0  NaN
|         2     1  4.0  3.0
```

```
 |                2  NaN  5.0
 |
 |      A ValueError is raised if there are any duplicates.
 |
 |      >>> df = pd.DataFrame({"foo": ['one', 'one', 'two', 'two'],
 |      ...                    "bar": ['A', 'A', 'B', 'C'],
 |      ...                    "baz": [1, 2, 3, 4]})
 |      >>> df
 |         foo bar  baz
 |      0  one   A    1
 |      1  one   A    2
 |      2  two   B    3
 |      3  two   C    4
 |
 |      Notice that the first two rows are the same for our `index`
 |      and `columns` arguments.
 |
 |      >>> df.pivot(index='foo', columns='bar', values='baz')
 |      Traceback (most recent call last):
 |         ...
 |      ValueError: Index contains duplicate entries, cannot reshape
 |
 |  pivot_table(self, values=None, index=None, columns=None, aggfunc='mean',
fill_value=None, margins=False, dropna=True, margins_name='All', observed=False,
sort=True) -> 'DataFrame'
 |      Create a spreadsheet-style pivot table as a DataFrame.
 |
 |      The levels in the pivot table will be stored in MultiIndex objects
 |      (hierarchical indexes) on the index and columns of the result DataFrame.
 |
 |      Parameters
 |      ----------
 |      values : column to aggregate, optional
 |      index : column, Grouper, array, or list of the previous
 |          If an array is passed, it must be the same length as the data. The
 |          list can contain any of the other types (except list).
 |          Keys to group by on the pivot table index.  If an array is passed,
 |          it is being used as the same manner as column values.
 |      columns : column, Grouper, array, or list of the previous
 |          If an array is passed, it must be the same length as the data. The
 |          list can contain any of the other types (except list).
 |          Keys to group by on the pivot table column.  If an array is passed,
 |          it is being used as the same manner as column values.
 |      aggfunc : function, list of functions, dict, default numpy.mean
 |          If list of functions passed, the resulting pivot table will have
 |          hierarchical columns whose top level are the function names
 |          (inferred from the function objects themselves)
 |          If dict is passed, the key is column to aggregate and value
 |          is function or list of functions.
 |      fill_value : scalar, default None
 |          Value to replace missing values with (in the resulting pivot table,
 |          after aggregation).
 |      margins : bool, default False
 |          Add all row / columns (e.g. for subtotal / grand totals).
 |      dropna : bool, default True
 |          Do not include columns whose entries are all NaN.
 |      margins_name : str, default 'All'
 |          Name of the row / column that will contain the totals
 |          when margins is True.
 |      observed : bool, default False
 |          This only applies if any of the groupers are Categoricals.
 |          If True: only show observed values for categorical groupers.
 |          If False: show all values for categorical groupers.
 |
 |          .. versionchanged:: 0.25.0
 |
 |      sort : bool, default True
 |          Specifies if the result should be sorted.
 |
 |          .. versionadded:: 1.3.0
 |
 |      Returns
 |      -------
 |      DataFrame
 |          An Excel style pivot table.
 |
 |      See Also
 |      --------
 |      DataFrame.pivot : Pivot without aggregation that can handle
 |          non-numeric data.
 |      DataFrame.melt: Unpivot a DataFrame from wide to long format,
 |          optionally leaving identifiers set.
 |      wide_to_long : Wide panel to long format. Less flexible but more
 |          user-friendly than melt.
 |
 |      Examples
 |      --------
 |      >>> df = pd.DataFrame({"A": ["foo", "foo", "foo", "foo", "foo",
 |      ...                          "bar", "bar", "bar", "bar"],
 |      ...                    "B": ["one", "one", "one", "two", "two",
 |      ...                          "one", "one", "two", "two"],
 |      ...                    "C": ["small", "large", "large", "small",
 |      ...                          "small", "large", "small", "small",
 |      ...                          "large"],
 |      ...                    "D": [1, 2, 2, 3, 3, 4, 5, 6, 7],
 |      ...                    "E": [2, 4, 5, 5, 6, 6, 8, 9, 9]})
 |      >>> df
 |           A    B      C  D  E
 |      0  foo  one  small  1  2
 |      1  foo  one  large  2  4
 |      2  foo  one  large  2  5
 |      3  foo  two  small  3  5
 |      4  foo  two  small  3  6
 |      5  bar  one  large  4  6
 |      6  bar  one  small  5  8
 |      7  bar  two  small  6  9
 |      8  bar  two  large  7  9
 |
 |      This first example aggregates values by taking the sum.
 |
 |      >>> table = pd.pivot_table(df, values='D', index=['A', 'B'],
 |      ...                     columns=['C'], aggfunc=np.sum)
 |      >>> table
 |      C        large  small
 |      A   B
 |      bar one    4.0    5.0
 |          two    7.0    6.0
 |      foo one    4.0    1.0
 |          two    NaN    6.0
 |
 |      We can also fill missing values using the `fill_value` parameter.
 |
 |      >>> table = pd.pivot_table(df, values='D', index=['A', 'B'],
 |      ...                     columns=['C'], aggfunc=np.sum, fill_value=0)
 |      >>> table
 |      C        large  small
 |      A   B
 |      bar one      4      5
 |          two      7      6
 |      foo one      4      1
 |          two      0      6
 |
 |      The next example aggregates by taking the mean across multiple columns.
 |
 |      >>> table = pd.pivot_table(df, values=['D', 'E'], index=['A', 'C'],
 |      ...                     aggfunc={'D': np.mean,
 |      ...                              'E': np.mean})
 |      >>> table
 |                      D         E
 |      A   C
 |      bar large  5.500000  7.500000
 |          small  5.500000  8.500000
 |      foo large  2.000000  4.500000
 |          small  2.333333  4.333333
 |
 |      We can also calculate multiple types of aggregations for any given
 |      value column.
 |
 |      >>> table = pd.pivot_table(df, values=['D', 'E'], index=['A', 'C'],
 |      ...                     aggfunc={'D': np.mean,
 |      ...                              'E': [min, max, np.mean]})
 |      >>> table
 |                      D    E
 |                  mean  max      mean  min
 |      A   C
 |      bar large  5.500000  9.0  7.500000  6.0
 |          small  5.500000  9.0  8.500000  8.0
 |      foo large  2.000000  5.0  4.500000  4.0
 |          small  2.333333  6.0  4.333333  2.0
 |
 |  pop(self, item: 'Hashable') -> 'Series'
 |      Return item and drop from frame. Raise KeyError if not found.
 |
 |      Parameters
 |      ----------
 |      item : label
 |          Label of column to be popped.
 |
 |      Returns
 |      -------
 |      Series
 |
 |      Examples
 |      --------
 |      >>> df = pd.DataFrame([('falcon', 'bird', 389.0),
 |      ...                    ('parrot', 'bird', 24.0),
 |      ...                    ('lion', 'mammal', 80.5),
 |      ...                    ('monkey', 'mammal', np.nan)],
 |      ...                   columns=('name', 'class', 'max_speed'))
 |      >>> df
 |           name   class  max_speed
 |      0  falcon    bird      389.0
 |      1  parrot    bird       24.0
 |      2    lion  mammal       80.5
 |      3  monkey  mammal        NaN
 |      
 |      >>> df.pop('class')
 |      0      bird
 |      1      bird
 |      2    mammal
 |      3    mammal
 |      Name: class, dtype: object
 |      
 |      >>> df
 |           name  max_speed
 |      0  falcon      389.0
 |      1  parrot       24.0
 |      2    lion       80.5
 |      3  monkey        NaN
 |  
 |  pow(self, other, axis='columns', level=None, fill_value=None)
 |      Get Exponential power of dataframe and other, element-wise (binary operator
`pow`).
 |      
 |      Equivalent to ``dataframe ** other``, but with support to substitute a
fill_value
 |      for missing data in one of the inputs. With reverse version, `rpow`.
 |      
 |      Among flexible wrappers (`add`, `sub`, `mul`, `div`, `mod`, `pow`) to
 |      arithmetic operators: `+`, `-`, `*`, `/`, `//`, `%`, `**`.
 |      
 |      Parameters
 |      ----------
 |      other : scalar, sequence, Series, or DataFrame
 |          Any single or multiple element data structure, or list-like object.
 |      axis : {0 or 'index', 1 or 'columns'}
 |          Whether to compare by the index (0 or 'index') or columns
 |          (1 or 'columns'). For Series input, axis to match Series index on.
 |      level : int or label
 |          Broadcast across a level, matching Index values on the
 |          passed MultiIndex level.
 |      fill_value : float or None, default None
 |          Fill existing missing (NaN) values, and any new element needed for
 |          successful DataFrame alignment, with this value before computation.
 |          If data in both corresponding DataFrame locations is missing
 |          the result will be missing.
 |      
 |      Returns
 |      -------
 |      DataFrame
 |          Result of the arithmetic operation.
 |      
 |      See Also
 |      --------
 |      DataFrame.add : Add DataFrames.
 |      DataFrame.sub : Subtract DataFrames.
 |      DataFrame.mul : Multiply DataFrames.
 |      DataFrame.div : Divide DataFrames (float division).
 |      DataFrame.truediv : Divide DataFrames (float division).
 |      DataFrame.floordiv : Divide DataFrames (integer division).
 |      DataFrame.mod : Calculate modulo (remainder after division).
 |      DataFrame.pow : Calculate exponential power.
 |      
 |      Notes
 |      -----
 |      Mismatched indices will be unioned together.
 |      
 |      Examples
 |      --------
 |      >>> df = pd.DataFrame({'angles': [0, 3, 4],
 |      ...                    'degrees': [360, 180, 360]},
 |      ...                   index=['circle', 'triangle', 'rectangle'])
 |      >>> df
 |                 angles  degrees
 |      circle          0      360
 |      triangle        3      180
 |      rectangle       4      360
 |      
 |      Add a scalar with operator version which return the same
 |      results.
 |      
 |      >>> df + 1
 |                 angles  degrees
 |      circle          1      361
 |      triangle        4      181
 |      rectangle       5      361
 |
 |      >>> df.add(1)
 |                 angles  degrees
 |      circle          1      361
 |      triangle        4      181
 |      rectangle       5      361
 |
 |      Divide by constant with reverse version.
 |
 |      >>> df.div(10)
 |                 angles  degrees
 |      circle        0.0     36.0
 |      triangle      0.3     18.0
 |      rectangle     0.4     36.0
 |
 |      >>> df.rdiv(10)
 |                   angles   degrees
 |      circle          inf  0.027778
 |      triangle   3.333333  0.055556
 |      rectangle  2.500000  0.027778
 |
 |      Subtract a list and Series by axis with operator version.
 |
 |      >>> df - [1, 2]
 |                 angles  degrees
 |      circle         -1      358
 |      triangle        2      178
 |      rectangle       3      358
 |
 |      >>> df.sub([1, 2], axis='columns')
 |                 angles  degrees
 |      circle         -1      358
 |      triangle        2      178
 |      rectangle       3      358
 |
 |      >>> df.sub(pd.Series([1, 1, 1], index=['circle', 'triangle', 'rectangle']),
 |      ...        axis='index')
 |                 angles  degrees
 |      circle         -1      359
 |      triangle        2      179
 |      rectangle       3      359
 |
 |      Multiply a DataFrame of different shape with operator version.
 |
 |      >>> other = pd.DataFrame({'angles': [0, 3, 4]},
 |      ...                      index=['circle', 'triangle', 'rectangle'])
 |      >>> other
 |                 angles
 |      circle          0
 |      triangle        3
 |      rectangle       4
 |
 |      >>> df * other
 |                 angles  degrees
 |      circle          0      NaN
 |      triangle        9      NaN
 |      rectangle      16      NaN
 |
 |      >>> df.mul(other, fill_value=0)
 |                 angles  degrees
 |      circle          0      0.0
 |      triangle        9      0.0
 |      rectangle      16      0.0
 |
 |      Divide by a MultiIndex by level.
 |
 |      >>> df_multindex = pd.DataFrame({'angles': [0, 3, 4, 4, 5, 6],
 |      ...                              'degrees': [360, 180, 360, 360, 540, 720]},
 |      ...                             index=[['A', 'A', 'A', 'B', 'B', 'B'],
 |      ...                                    ['circle', 'triangle', 'rectangle',
 |      ...                                     'square', 'pentagon', 'hexagon']])
 |      >>> df_multindex
 |                   angles  degrees
 |      A circle          0      360
 |        triangle        3      180
 |        rectangle       4      360
 |      B square          4      360
 |        pentagon        5      540
 |        hexagon         6      720
 |
 |      >>> df.div(df_multindex, level=1, fill_value=0)
 |                   angles  degrees
```

```
 |      A circle        NaN      1.0
 |        triangle      1.0      1.0
 |        rectangle     1.0      1.0
 |      B square        0.0      0.0
 |        pentagon      0.0      0.0
 |        hexagon       0.0      0.0
 |
 |  prod(self, axis=None, skipna=None, level=None, numeric_only=None, min_count=0,
**kwargs)
 |      Return the product of the values over the requested axis.
 |
 |      Parameters
 |      ----------
 |      axis : {index (0), columns (1)}
 |          Axis for the function to be applied on.
 |      skipna : bool, default True
 |          Exclude NA/null values when computing the result.
 |      level : int or level name, default None
 |          If the axis is a MultiIndex (hierarchical), count along a
 |          particular level, collapsing into a Series.
 |      numeric_only : bool, default None
 |          Include only float, int, boolean columns. If None, will attempt to use
 |          everything, then use only numeric data. Not implemented for Series.
 |      min_count : int, default 0
 |          The required number of valid values to perform the operation. If fewer
than
 |          ``min_count`` non-NA values are present the result will be NA.
 |      **kwargs
 |          Additional keyword arguments to be passed to the function.
 |
 |      Returns
 |      -------
 |      Series or DataFrame (if level specified)
 |
 |      See Also
 |      --------
 |      Series.sum : Return the sum.
 |      Series.min : Return the minimum.
 |      Series.max : Return the maximum.
 |      Series.idxmin : Return the index of the minimum.
 |      Series.idxmax : Return the index of the maximum.
 |      DataFrame.sum : Return the sum over the requested axis.
 |      DataFrame.min : Return the minimum over the requested axis.
 |      DataFrame.max : Return the maximum over the requested axis.
 |      DataFrame.idxmin : Return the index of the minimum over the requested axis.
 |      DataFrame.idxmax : Return the index of the maximum over the requested axis.
 |
 |      Examples
 |      --------
 |      By default, the product of an empty or all-NA Series is ``1``
 |
 |      >>> pd.Series([], dtype="float64").prod()
 |      1.0
 |
 |      This can be controlled with the ``min_count`` parameter
 |
 |      >>> pd.Series([], dtype="float64").prod(min_count=1)
 |      nan
 |
 |      Thanks to the ``skipna`` parameter, ``min_count`` handles all-NA and
 |      empty series identically.
 |
 |      >>> pd.Series([np.nan]).prod()
 |      1.0
 |
 |      >>> pd.Series([np.nan]).prod(min_count=1)
 |      nan
 |
 |  product = prod(self, axis=None, skipna=None, level=None, numeric_only=None,
min_count=0, **kwargs)
 |
 |  quantile(self, q=0.5, axis: 'Axis' = 0, numeric_only: 'bool' = True,
interpolation: 'str' = 'linear')
 |      Return values at the given quantile over requested axis.
 |
 |      Parameters
 |      ----------
 |      q : float or array-like, default 0.5 (50% quantile)
 |          Value between 0 <= q <= 1, the quantile(s) to compute.
 |      axis : {0, 1, 'index', 'columns'}, default 0
 |          Equals 0 or 'index' for row-wise, 1 or 'columns' for column-wise.
 |      numeric_only : bool, default True
 |          If False, the quantile of datetime and timedelta data will be
 |          computed as well.
 |      interpolation : {'linear', 'lower', 'higher', 'midpoint', 'nearest'}
 |          This optional parameter specifies the interpolation method to use,
 |          when the desired quantile lies between two data points `i` and `j`:
```

```
 |
 |          * linear: `i + (j - i) * fraction`, where `fraction` is the
 |            fractional part of the index surrounded by `i` and `j`.
 |          * lower: `i`.
 |          * higher: `j`.
 |          * nearest: `i` or `j` whichever is nearest.
 |          * midpoint: (`i` + `j`) / 2.
 |
 |      Returns
 |      -------
 |      Series or DataFrame
 |
 |          If ``q`` is an array, a DataFrame will be returned where the
 |            index is ``q``, the columns are the columns of self, and the
 |            values are the quantiles.
 |          If ``q`` is a float, a Series will be returned where the
 |            index is the columns of self and the values are the quantiles.
 |
 |      See Also
 |      --------
 |      core.window.Rolling.quantile: Rolling quantile.
 |      numpy.percentile: Numpy function to compute the percentile.
 |
 |      Examples
 |      --------
 |      >>> df = pd.DataFrame(np.array([[1, 1], [2, 10], [3, 100], [4, 100]]),
 |      ...                   columns=['a', 'b'])
 |      >>> df.quantile(.1)
 |      a    1.3
 |      b    3.7
 |      Name: 0.1, dtype: float64
 |      >>> df.quantile([.1, .5])
 |             a     b
 |      0.1  1.3   3.7
 |      0.5  2.5  55.0
 |
 |      Specifying `numeric_only=False` will also compute the quantile of
 |      datetime and timedelta data.
 |
 |      >>> df = pd.DataFrame({'A': [1, 2],
 |      ...                    'B': [pd.Timestamp('2010'),
 |      ...                          pd.Timestamp('2011')],
 |      ...                    'C': [pd.Timedelta('1 days'),
 |      ...                          pd.Timedelta('2 days')]})
 |      >>> df.quantile(0.5, numeric_only=False)
 |      A                   1.5
 |      B    2010-07-02 12:00:00
 |      C        1 days 12:00:00
 |      Name: 0.5, dtype: object
 |
 |  query(self, expr: 'str', inplace: 'bool' = False, **kwargs)
 |      Query the columns of a DataFrame with a boolean expression.
 |
 |      Parameters
 |      ----------
 |      expr : str
 |          The query string to evaluate.
 |
 |          You can refer to variables
 |          in the environment by prefixing them with an '@' character like
 |          ``@a + b``.
 |
 |          You can refer to column names that are not valid Python variable names
 |          by surrounding them in backticks. Thus, column names containing spaces
 |          or punctuations (besides underscores) or starting with digits must be
 |          surrounded by backticks. (For example, a column named "Area (cm^2)" would
 |          be referenced as ```Area (cm^2)```). Column names which are Python
keywords
 |          (like "list", "for", "import", etc) cannot be used.
 |
 |          For example, if one of your columns is called ``a a`` and you want
 |          to sum it with ``b``, your query should be ```a a` + b``.
 |
 |          .. versionadded:: 0.25.0
 |              Backtick quoting introduced.
 |
 |          .. versionadded:: 1.0.0
 |              Expanding functionality of backtick quoting for more than only
spaces.
 |
 |      inplace : bool
 |          Whether the query should modify the data in place or return
 |          a modified copy.
 |      **kwargs
 |          See the documentation for :func:`eval` for complete details
 |          on the keyword arguments accepted by :meth:`DataFrame.query`.
 |
```

```
 |      Returns
 |      -------
 |      DataFrame or None
 |          DataFrame resulting from the provided query expression or
 |          None if ``inplace=True``.
 |
 |      See Also
 |      --------
 |      eval : Evaluate a string describing operations on
 |          DataFrame columns.
 |      DataFrame.eval : Evaluate a string describing operations on
 |          DataFrame columns.
 |
 |      Notes
 |      -----
 |      The result of the evaluation of this expression is first passed to
 |      :attr:`DataFrame.loc` and if that fails because of a
 |      multidimensional key (e.g., a DataFrame) then the result will be passed
 |      to :meth:`DataFrame.__getitem__`.
 |
 |      This method uses the top-level :func:`eval` function to
 |      evaluate the passed query.
 |
 |      The :meth:`~pandas.DataFrame.query` method uses a slightly
 |      modified Python syntax by default. For example, the ``&`` and ``|``
 |      (bitwise) operators have the precedence of their boolean cousins,
 |      :keyword:`and` and :keyword:`or`. This *is* syntactically valid Python,
 |      however the semantics are different.
 |
 |      You can change the semantics of the expression by passing the keyword
 |      argument ``parser='python'``. This enforces the same semantics as
 |      evaluation in Python space. Likewise, you can pass ``engine='python'``
 |      to evaluate an expression using Python itself as a backend. This is not
 |      recommended as it is inefficient compared to using ``numexpr`` as the
 |      engine.
 |
 |      The :attr:`DataFrame.index` and
 |      :attr:`DataFrame.columns` attributes of the
 |      :class:`~pandas.DataFrame` instance are placed in the query namespace
 |      by default, which allows you to treat both the index and columns of the
 |      frame as a column in the frame.
 |      The identifier ``index`` is used for the frame index; you can also
 |      use the name of the index to identify it in a query. Please note that
 |      Python keywords may not be used as identifiers.
 |
 |      For further details and examples see the ``query`` documentation in
 |      :ref:`indexing <indexing.query>`.
 |
 |      *Backtick quoted variables*
 |
 |      Backtick quoted variables are parsed as literal Python code and
 |      are converted internally to a Python valid identifier.
 |      This can lead to the following problems.
 |
 |      During parsing a number of disallowed characters inside the backtick
 |      quoted string are replaced by strings that are allowed as a Python
identifier.
 |      These characters include all operators in Python, the space character, the
 |      question mark, the exclamation mark, the dollar sign, and the euro sign.
 |      For other characters that fall outside the ASCII range (U+0001..U+007F)
 |      and those that are not further specified in PEP 3131,
 |      the query parser will raise an error.
 |      This excludes whitespace different than the space character,
 |      but also the hashtag (as it is used for comments) and the backtick
 |      itself (backtick can also not be escaped).
 |
 |      In a special case, quotes that make a pair around a backtick can
 |      confuse the parser.
 |      For example, ```it's` > `that's``` will raise an error,
 |      as it forms a quoted string (``'s > `that'``) with a backtick inside.
 |
 |      See also the Python documentation about lexical analysis
 |      (https://docs.python.org/3/reference/lexical_analysis.html)
 |      in combination with the source code in
:mod:`pandas.core.computation.parsing`.
 |
 |      Examples
 |      --------
 |      >>> df = pd.DataFrame({'A': range(1, 6),
 |      ...                    'B': range(10, 0, -2),
 |      ...                    'C C': range(10, 5, -1)})
 |      >>> df
 |         A   B  C C
 |      0  1  10   10
 |      1  2   8    9
 |      2  3   6    8
 |      3  4   4    7
```

```
 |      4  5   2    6
 |      >>> df.query('A > B')
 |         A  B  C C
 |      4  5  2    6
 |
 |      The previous expression is equivalent to
 |
 |      >>> df[df.A > df.B]
 |         A  B  C C
 |      4  5  2    6
 |
 |      For columns with spaces in their name, you can use backtick quoting.
 |
 |      >>> df.query('B == `C C`')
 |         A   B  C C
 |      0  1  10   10
 |
 |      The previous expression is equivalent to
 |
 |      >>> df[df.B == df['C C']]
 |         A   B  C C
 |      0  1  10   10
 |
 |  radd(self, other, axis='columns', level=None, fill_value=None)
 |      Get Addition of dataframe and other, element-wise (binary operator `radd`).
 |
 |      Equivalent to ``other + dataframe``, but with support to substitute a
fill_value
 |      for missing data in one of the inputs. With reverse version, `add`.
 |
 |      Among flexible wrappers (`add`, `sub`, `mul`, `div`, `mod`, `pow`) to
 |      arithmetic operators: `+`, `-`, `*`, `/`, `//`, `%`, `**`.
 |
 |      Parameters
 |      ----------
 |      other : scalar, sequence, Series, or DataFrame
 |          Any single or multiple element data structure, or list-like object.
 |      axis : {0 or 'index', 1 or 'columns'}
 |          Whether to compare by the index (0 or 'index') or columns
 |          (1 or 'columns'). For Series input, axis to match Series index on.
 |      level : int or label
 |          Broadcast across a level, matching Index values on the
 |          passed MultiIndex level.
 |      fill_value : float or None, default None
 |          Fill existing missing (NaN) values, and any new element needed for
 |          successful DataFrame alignment, with this value before computation.
 |          If data in both corresponding DataFrame locations is missing
 |          the result will be missing.
 |
 |      Returns
 |      -------
 |      DataFrame
 |          Result of the arithmetic operation.
 |
 |      See Also
 |      --------
 |      DataFrame.add : Add DataFrames.
 |      DataFrame.sub : Subtract DataFrames.
 |      DataFrame.mul : Multiply DataFrames.
 |      DataFrame.div : Divide DataFrames (float division).
 |      DataFrame.truediv : Divide DataFrames (float division).
 |      DataFrame.floordiv : Divide DataFrames (integer division).
 |      DataFrame.mod : Calculate modulo (remainder after division).
 |      DataFrame.pow : Calculate exponential power.
 |
 |      Notes
 |      -----
 |      Mismatched indices will be unioned together.
 |
 |      Examples
 |      --------
 |      >>> df = pd.DataFrame({'angles': [0, 3, 4],
 |      ...                    'degrees': [360, 180, 360]},
 |      ...                   index=['circle', 'triangle', 'rectangle'])
 |      >>> df
 |                 angles  degrees
 |      circle          0      360
 |      triangle        3      180
 |      rectangle       4      360
 |
 |      Add a scalar with operator version which return the same
 |      results.
 |
 |      >>> df + 1
 |                 angles  degrees
 |      circle          1      361
 |      triangle        4      181
 |      rectangle       5      361
 |
 |      >>> df.add(1)
 |                 angles  degrees
 |      circle          1      361
 |      triangle        4      181
 |      rectangle       5      361
 |
 |      Divide by constant with reverse version.
 |
 |      >>> df.div(10)
 |                 angles  degrees
 |      circle        0.0     36.0
 |      triangle      0.3     18.0
 |      rectangle     0.4     36.0
 |
 |      >>> df.rdiv(10)
 |                   angles   degrees
 |      circle          inf  0.027778
 |      triangle   3.333333  0.055556
 |      rectangle  2.500000  0.027778
 |
 |      Subtract a list and Series by axis with operator version.
 |
 |      >>> df - [1, 2]
 |                 angles  degrees
 |      circle         -1      358
 |      triangle        2      178
 |      rectangle       3      358
 |
 |      >>> df.sub([1, 2], axis='columns')
 |                 angles  degrees
 |      circle         -1      358
 |      triangle        2      178
 |      rectangle       3      358
 |
 |      >>> df.sub(pd.Series([1, 1, 1], index=['circle', 'triangle', 'rectangle']),
 |      ...        axis='index')
 |                 angles  degrees
 |      circle         -1      359
 |      triangle        2      179
 |      rectangle       3      359
 |
 |      Multiply a DataFrame of different shape with operator version.
 |
 |      >>> other = pd.DataFrame({'angles': [0, 3, 4]},
 |      ...                      index=['circle', 'triangle', 'rectangle'])
 |      >>> other
 |                 angles
 |      circle          0
 |      triangle        3
 |      rectangle       4
 |
 |      >>> df * other
 |                 angles  degrees
 |      circle          0      NaN
 |      triangle        9      NaN
 |      rectangle      16      NaN
 |
 |      >>> df.mul(other, fill_value=0)
 |                 angles  degrees
 |      circle          0      0.0
 |      triangle        9      0.0
 |      rectangle      16      0.0
 |
 |      Divide by a MultiIndex by level.
 |
 |      >>> df_multindex = pd.DataFrame({'angles': [0, 3, 4, 4, 5, 6],
 |      ...                              'degrees': [360, 180, 360, 360, 540, 720]},
 |      ...                             index=[['A', 'A', 'A', 'B', 'B', 'B'],
 |      ...                                    ['circle', 'triangle', 'rectangle',
 |      ...                                     'square', 'pentagon', 'hexagon']])
 |      >>> df_multindex
 |                   angles  degrees
 |      A circle          0      360
 |        triangle        3      180
 |        rectangle       4      360
 |      B square          4      360
 |        pentagon        5      540
 |        hexagon         6      720
 |
 |      >>> df.div(df_multindex, level=1, fill_value=0)
 |                   angles  degrees
 |      A circle        NaN      1.0
 |        triangle      1.0      1.0
 |        rectangle     1.0      1.0
 |      B square        0.0      0.0
```

```
 |        pentagon      0.0      0.0
 |        hexagon       0.0      0.0
 |  
 |  rdiv = rtruediv(self, other, axis='columns', level=None, fill_value=None)
 |  
 |  reindex(self, labels=None, index=None, columns=None, axis=None, method=None,
copy=True, level=None, fill_value=nan, limit=None, tolerance=None)
 |      Conform Series/DataFrame to new index with optional filling logic.
 |      
 |      Places NA/NaN in locations having no value in the previous index. A new
object
 |      is produced unless the new index is equivalent to the current one and
 |      ``copy=False``.
 |      
 |      Parameters
 |      ----------
 |      
 |      keywords for axes : array-like, optional
 |          New labels / index to conform to, should be specified using
 |          keywords. Preferably an Index object to avoid duplicating data.
 |      
 |      method : {None, 'backfill'/'bfill', 'pad'/'ffill', 'nearest'}
 |          Method to use for filling holes in reindexed DataFrame.
 |          Please note: this is only applicable to DataFrames/Series with a
 |          monotonically increasing/decreasing index.
 |      
 |          * None (default): don't fill gaps
 |          * pad / ffill: Propagate last valid observation forward to next
 |            valid.
 |          * backfill / bfill: Use next valid observation to fill gap.
 |          * nearest: Use nearest valid observations to fill gap.
 |      
 |      copy : bool, default True
 |          Return a new object, even if the passed indexes are the same.
 |      level : int or name
 |          Broadcast across a level, matching Index values on the
 |          passed MultiIndex level.
 |      fill_value : scalar, default np.NaN
 |          Value to use for missing values. Defaults to NaN, but can be any
 |          "compatible" value.
 |      limit : int, default None
 |          Maximum number of consecutive elements to forward or backward fill.
 |      tolerance : optional
 |          Maximum distance between original and new labels for inexact
 |          matches. The values of the index at the matching locations most
 |          satisfy the equation ``abs(index[indexer] - target) <= tolerance``.
 |      
 |          Tolerance may be a scalar value, which applies the same tolerance
 |          to all values, or list-like, which applies variable tolerance per
 |          element. List-like includes list, tuple, array, Series, and must be
 |          the same size as the index and its dtype must exactly match the
 |          index's type.
 |      
 |      Returns
 |      -------
 |      Series/DataFrame with changed index.
 |      
 |      See Also
 |      --------
 |      DataFrame.set_index : Set row labels.
 |      DataFrame.reset_index : Remove row labels or move them to new columns.
 |      DataFrame.reindex_like : Change to same indices as other DataFrame.
 |      
 |      Examples
 |      --------
 |      ``DataFrame.reindex`` supports two calling conventions
 |      
 |      * ``(index=index_labels, columns=column_labels, ...)``
 |      * ``(labels, axis={'index', 'columns'}, ...)``
 |      
 |      We *highly* recommend using keyword arguments to clarify your
 |      intent.
 |      
 |      Create a dataframe with some fictional data.
 |      
 |      >>> index = ['Firefox', 'Chrome', 'Safari', 'IE10', 'Konqueror']
 |      >>> df = pd.DataFrame({'http_status': [200, 200, 404, 404, 301],
 |      ...                   'response_time': [0.04, 0.02, 0.07, 0.08, 1.0]},
 |      ...                   index=index)
 |      >>> df
 |                 http_status  response_time
 |      Firefox            200           0.04
 |      Chrome             200           0.02
 |      Safari             404           0.07
 |      IE10               404           0.08
 |      Konqueror          301           1.00
 |      
 |      Create a new index and reindex the dataframe. By default
 |      values in the new index that do not have corresponding
 |      records in the dataframe are assigned ``NaN``.
 |
 |      >>> new_index = ['Safari', 'Iceweasel', 'Comodo Dragon', 'IE10',
 |      ...              'Chrome']
 |      >>> df.reindex(new_index)
 |                     http_status  response_time
 |      Safari               404.0           0.07
 |      Iceweasel              NaN            NaN
 |      Comodo Dragon          NaN            NaN
 |      IE10                 404.0           0.08
 |      Chrome               200.0           0.02
 |
 |      We can fill in the missing values by passing a value to
 |      the keyword ``fill_value``. Because the index is not monotonically
 |      increasing or decreasing, we cannot use arguments to the keyword
 |      ``method`` to fill the ``NaN`` values.
 |
 |      >>> df.reindex(new_index, fill_value=0)
 |                     http_status  response_time
 |      Safari                 404           0.07
 |      Iceweasel                0           0.00
 |      Comodo Dragon            0           0.00
 |      IE10                   404           0.08
 |      Chrome                 200           0.02
 |
 |      >>> df.reindex(new_index, fill_value='missing')
 |                    http_status response_time
 |      Safari                404          0.07
 |      Iceweasel         missing       missing
 |      Comodo Dragon     missing       missing
 |      IE10                  404          0.08
 |      Chrome                200          0.02
 |
 |      We can also reindex the columns.
 |
 |      >>> df.reindex(columns=['http_status', 'user_agent'])
 |                 http_status  user_agent
 |      Firefox            200         NaN
 |      Chrome             200         NaN
 |      Safari             404         NaN
 |      IE10               404         NaN
 |      Konqueror          301         NaN
 |
 |      Or we can use "axis-style" keyword arguments
 |
 |      >>> df.reindex(['http_status', 'user_agent'], axis="columns")
 |                 http_status  user_agent
 |      Firefox            200         NaN
 |      Chrome             200         NaN
 |      Safari             404         NaN
 |      IE10               404         NaN
 |      Konqueror          301         NaN
 |
 |      To further illustrate the filling functionality in
 |      ``reindex``, we will create a dataframe with a
 |      monotonically increasing index (for example, a sequence
 |      of dates).
 |
 |      >>> date_index = pd.date_range('1/1/2010', periods=6, freq='D')
 |      >>> df2 = pd.DataFrame({"prices": [100, 101, np.nan, 100, 89, 88]},
 |      ...                    index=date_index)
 |      >>> df2
 |                  prices
 |      2010-01-01   100.0
 |      2010-01-02   101.0
 |      2010-01-03     NaN
 |      2010-01-04   100.0
 |      2010-01-05    89.0
 |      2010-01-06    88.0
 |
 |      Suppose we decide to expand the dataframe to cover a wider
 |      date range.
 |
 |      >>> date_index2 = pd.date_range('12/29/2009', periods=10, freq='D')
 |      >>> df2.reindex(date_index2)
 |                  prices
 |      2009-12-29     NaN
 |      2009-12-30     NaN
 |      2009-12-31     NaN
 |      2010-01-01   100.0
 |      2010-01-02   101.0
 |      2010-01-03     NaN
 |      2010-01-04   100.0
 |      2010-01-05    89.0
 |      2010-01-06    88.0
```

```
 |      2010-01-07     NaN
 |
 |      The index entries that did not have a value in the original data frame
 |      (for example, '2009-12-29') are by default filled with ``NaN``.
 |      If desired, we can fill in the missing values using one of several
 |      options.
 |
 |      For example, to back-propagate the last valid value to fill the ``NaN``
 |      values, pass ``bfill`` as an argument to the ``method`` keyword.
 |
 |      >>> df2.reindex(date_index2, method='bfill')
 |                  prices
 |      2009-12-29   100.0
 |      2009-12-30   100.0
 |      2009-12-31   100.0
 |      2010-01-01   100.0
 |      2010-01-02   101.0
 |      2010-01-03     NaN
 |      2010-01-04   100.0
 |      2010-01-05    89.0
 |      2010-01-06    88.0
 |      2010-01-07     NaN
 |
 |      Please note that the ``NaN`` value present in the original dataframe
 |      (at index value 2010-01-03) will not be filled by any of the
 |      value propagation schemes. This is because filling while reindexing
 |      does not look at dataframe values, but only compares the original and
 |      desired indexes. If you do want to fill in the ``NaN`` values present
 |      in the original dataframe, use the ``fillna()`` method.
 |
 |      See the :ref:`user guide <basics.reindexing>` for more.
 |
 |  rename(self, mapper=None, index=None, columns=None, axis=None, copy=True,
inplace=False, level=None, errors='ignore')
 |      Alter axes labels.
 |
 |      Function / dict values must be unique (1-to-1). Labels not contained in
 |      a dict / Series will be left as-is. Extra labels listed don't throw an
 |      error.
 |
 |      See the :ref:`user guide <basics.rename>` for more.
 |
 |      Parameters
 |      ----------
 |      mapper : dict-like or function
 |          Dict-like or function transformations to apply to
 |          that axis' values. Use either ``mapper`` and ``axis`` to
 |          specify the axis to target with ``mapper``, or ``index`` and
 |          ``columns``.
 |      index : dict-like or function
 |          Alternative to specifying axis (``mapper, axis=0``
 |          is equivalent to ``index=mapper``).
 |      columns : dict-like or function
 |          Alternative to specifying axis (``mapper, axis=1``
 |          is equivalent to ``columns=mapper``).
 |      axis : {0 or 'index', 1 or 'columns'}, default 0
 |          Axis to target with ``mapper``. Can be either the axis name
 |          ('index', 'columns') or number (0, 1). The default is 'index'.
 |      copy : bool, default True
 |          Also copy underlying data.
 |      inplace : bool, default False
 |          Whether to return a new DataFrame. If True then value of copy is
 |          ignored.
 |      level : int or level name, default None
 |          In case of a MultiIndex, only rename labels in the specified
 |          level.
 |      errors : {'ignore', 'raise'}, default 'ignore'
 |          If 'raise', raise a `KeyError` when a dict-like `mapper`, `index`,
 |          or `columns` contains labels that are not present in the Index
 |          being transformed.
 |          If 'ignore', existing keys will be renamed and extra keys will be
 |          ignored.
 |
 |      Returns
 |      -------
 |      DataFrame or None
 |          DataFrame with the renamed axis labels or None if ``inplace=True``.
 |
 |      Raises
 |      ------
 |      KeyError
 |          If any of the labels is not found in the selected axis and
 |          "errors='raise'".
 |
 |      See Also
 |      --------
 |      DataFrame.rename_axis : Set the name of the axis.
 |
 |      Examples
 |      --------
 |      ``DataFrame.rename`` supports two calling conventions
 |
 |      * ``(index=index_mapper, columns=columns_mapper, ...)``
 |      * ``(mapper, axis={'index', 'columns'}, ...)``
 |
 |      We *highly* recommend using keyword arguments to clarify your
 |      intent.
 |
 |      Rename columns using a mapping:
 |
 |      >>> df = pd.DataFrame({"A": [1, 2, 3], "B": [4, 5, 6]})
 |      >>> df.rename(columns={"A": "a", "B": "c"})
 |         a  c
 |      0  1  4
 |      1  2  5
 |      2  3  6
 |
 |      Rename index using a mapping:
 |
 |      >>> df.rename(index={0: "x", 1: "y", 2: "z"})
 |         A  B
 |      x  1  4
 |      y  2  5
 |      z  3  6
 |
 |      Cast index labels to a different type:
 |
 |      >>> df.index
 |      RangeIndex(start=0, stop=3, step=1)
 |      >>> df.rename(index=str).index
 |      Index(['0', '1', '2'], dtype='object')
 |
 |      >>> df.rename(columns={"A": "a", "B": "b", "C": "c"}, errors="raise")
 |      Traceback (most recent call last):
 |      KeyError: ['C'] not found in axis
 |
 |      Using axis-style parameters:
 |
 |      >>> df.rename(str.lower, axis='columns')
 |         a  b
 |      0  1  4
 |      1  2  5
 |      2  3  6
 |
 |      >>> df.rename({1: 2, 2: 4}, axis='index')
 |         A  B
 |      0  1  4
 |      2  2  5
 |      4  3  6
 |
 |  reorder_levels(self, order: 'Sequence[Axis]', axis: 'Axis' = 0) -> 'DataFrame'
 |      Rearrange index levels using input order. May not drop or duplicate levels.
 |
 |      Parameters
 |      ----------
 |      order : list of int or list of str
 |          List representing new level order. Reference level by number
 |          (position) or by key (label).
 |      axis : {0 or 'index', 1 or 'columns'}, default 0
 |          Where to reorder levels.
 |
 |      Returns
 |      -------
 |      DataFrame
 |
 |  replace(self, to_replace=None, value=None, inplace: 'bool' = False, limit=None,
regex: 'bool' = False, method: 'str' = 'pad')
 |      Replace values given in `to_replace` with `value`.
 |
 |      Values of the DataFrame are replaced with other values dynamically.
 |
 |      This differs from updating with ``.loc`` or ``.iloc``, which require
 |      you to specify a location to update with some value.
 |
 |      Parameters
 |      ----------
 |      to_replace : str, regex, list, dict, Series, int, float, or None
 |          How to find the values that will be replaced.
 |
 |          * numeric, str or regex:
 |
 |              - numeric: numeric values equal to `to_replace` will be
 |                  replaced with `value`
 |              - str: string exactly matching `to_replace` will be replaced
 |                 with `value`
 |             - regex: regexs matching `to_replace` will be replaced with
 |                 `value`
 |
 |         * list of str, regex, or numeric:
 |
 |             - First, if `to_replace` and `value` are both lists, they
 |                 **must** be the same length.
 |             - Second, if ``regex=True`` then all of the strings in **both**
 |                 lists will be interpreted as regexs otherwise they will match
 |                 directly. This doesn't matter much for `value` since there
 |                 are only a few possible substitution regexes you can use.
 |             - str, regex and numeric rules apply as above.
 |
 |         * dict:
 |
 |             - Dicts can be used to specify different replacement values
 |                 for different existing values. For example,
 |                 ``{'a': 'b', 'y': 'z'}`` replaces the value 'a' with 'b' and
 |                 'y' with 'z'. To use a dict in this way the `value`
 |                 parameter should be `None`.
 |             - For a DataFrame a dict can specify that different values
 |                 should be replaced in different columns. For example,
 |                 ``{'a': 1, 'b': 'z'}`` looks for the value 1 in column 'a'
 |                 and the value 'z' in column 'b' and replaces these values
 |                 with whatever is specified in `value`. The `value` parameter
 |                 should not be ``None`` in this case. You can treat this as a
 |                 special case of passing two lists except that you are
 |                 specifying the column to search in.
 |             - For a DataFrame nested dictionaries, e.g.,
 |                 ``{'a': {'b': np.nan}}``, are read as follows: look in column
 |                 'a' for the value 'b' and replace it with NaN. The `value`
 |                 parameter should be ``None`` to use a nested dict in this
 |                 way. You can nest regular expressions as well. Note that
 |                 column names (the top-level dictionary keys in a nested
 |                 dictionary) **cannot** be regular expressions.
 |
 |         * None:
 |
 |             - This means that the `regex` argument must be a string,
 |                 compiled regular expression, or list, dict, ndarray or
 |                 Series of such elements. If `value` is also ``None`` then
 |                 this **must** be a nested dictionary or Series.
 |
 |         See the examples section for examples of each of these.
 |     value : scalar, dict, list, str, regex, default None
 |         Value to replace any values matching `to_replace` with.
 |         For a DataFrame a dict of values can be used to specify which
 |         value to use for each column (columns not in the dict will not be
 |         filled). Regular expressions, strings and lists or dicts of such
 |         objects are also allowed.
 |
 |     inplace : bool, default False
 |         If True, performs operation inplace and returns None.
 |     limit : int, default None
 |         Maximum size gap to forward or backward fill.
 |     regex : bool or same types as `to_replace`, default False
 |         Whether to interpret `to_replace` and/or `value` as regular
 |         expressions. If this is ``True`` then `to_replace` *must* be a
 |         string. Alternatively, this could be a regular expression or a
 |         list, dict, or array of regular expressions in which case
 |         `to_replace` must be ``None``.
 |     method : {'pad', 'ffill', 'bfill', `None`}
 |         The method to use when for replacement, when `to_replace` is a
 |         scalar, list or tuple and `value` is ``None``.
 |
 |         .. versionchanged:: 0.23.0
 |             Added to DataFrame.
 |
 |     Returns
 |     -------
 |     DataFrame
 |         Object after replacement.
 |
 |     Raises
 |     ------
 |     AssertionError
 |         * If `regex` is not a ``bool`` and `to_replace` is not
 |             ``None``.
 |
 |     TypeError
 |         * If `to_replace` is not a scalar, array-like, ``dict``, or ``None``
 |         * If `to_replace` is a ``dict`` and `value` is not a ``list``,
 |             ``dict``, ``ndarray``, or ``Series``
 |         * If `to_replace` is ``None`` and `regex` is not compilable
 |             into a regular expression or is a list, dict, ndarray, or
 |             Series.
 |          * When replacing multiple ``bool`` or ``datetime64`` objects and
 |              the arguments to `to_replace` does not match the type of the
 |              value being replaced
 |
 |      ValueError
 |          * If a ``list`` or an ``ndarray`` is passed to `to_replace` and
 |              `value` but they are not the same length.
 |
 |      See Also
 |      --------
 |      DataFrame.fillna : Fill NA values.
 |      DataFrame.where : Replace values based on boolean condition.
 |      Series.str.replace : Simple string replacement.
 |
 |      Notes
 |      -----
 |      * Regex substitution is performed under the hood with ``re.sub``. The
 |          rules for substitution for ``re.sub`` are the same.
 |      * Regular expressions will only substitute on strings, meaning you
 |          cannot provide, for example, a regular expression matching floating
 |          point numbers and expect the columns in your frame that have a
 |          numeric dtype to be matched. However, if those floating point
 |          numbers *are* strings, then you can do this.
 |      * This method has *a lot* of options. You are encouraged to experiment
 |          and play with this method to gain intuition about how it works.
 |      * When dict is used as the `to_replace` value, it is like
 |          key(s) in the dict are the to_replace part and
 |          value(s) in the dict are the value parameter.
 |
 |      Examples
 |      --------
 |
 |      **Scalar `to_replace` and `value`**
 |
 |      >>> s = pd.Series([0, 1, 2, 3, 4])
 |      >>> s.replace(0, 5)
 |      0    5
 |      1    1
 |      2    2
 |      3    3
 |      4    4
 |      dtype: int64
 |
 |      >>> df = pd.DataFrame({'A': [0, 1, 2, 3, 4],
 |      ...                    'B': [5, 6, 7, 8, 9],
 |      ...                    'C': ['a', 'b', 'c', 'd', 'e']})
 |      >>> df.replace(0, 5)
 |          A  B  C
 |      0  5  5  a
 |      1  1  6  b
 |      2  2  7  c
 |      3  3  8  d
 |      4  4  9  e
 |
 |      **List-like `to_replace`**
 |
 |      >>> df.replace([0, 1, 2, 3], 4)
 |          A  B  C
 |      0  4  5  a
 |      1  4  6  b
 |      2  4  7  c
 |      3  4  8  d
 |      4  4  9  e
 |
 |      >>> df.replace([0, 1, 2, 3], [4, 3, 2, 1])
 |          A  B  C
 |      0  4  5  a
 |      1  3  6  b
 |      2  2  7  c
 |      3  1  8  d
 |      4  4  9  e
 |
 |      >>> s.replace([1, 2], method='bfill')
 |      0    0
 |      1    3
 |      2    3
 |      3    3
 |      4    4
 |      dtype: int64
 |
 |      **dict-like `to_replace`**
 |
 |      >>> df.replace({0: 10, 1: 100})
 |              A  B  C
 |      0   10  5  a
 |      1  100  6  b
 |      2    2  7  c
 |      3    3  8  d
 |      4    4  9  e
 |
 |      >>> df.replace({'A': 0, 'B': 5}, 100)
 |              A    B  C
 |      0  100  100  a
 |      1    1    6  b
 |      2    2    7  c
 |      3    3    8  d
 |      4    4    9  e
 |
 |      >>> df.replace({'A': {0: 100, 4: 400}})
 |              A  B  C
 |      0  100  5  a
 |      1    1  6  b
 |      2    2  7  c
 |      3    3  8  d
 |      4  400  9  e
 |
 |      **Regular expression `to_replace`**
 |
 |      >>> df = pd.DataFrame({'A': ['bat', 'foo', 'bait'],
 |      ...                    'B': ['abc', 'bar', 'xyz']})
 |      >>> df.replace(to_replace=r'^ba.$', value='new', regex=True)
 |              A    B
 |      0   new  abc
 |      1   foo  new
 |      2  bait  xyz
 |
 |      >>> df.replace({'A': r'^ba.$'}, {'A': 'new'}, regex=True)
 |              A    B
 |      0   new  abc
 |      1   foo  bar
 |      2  bait  xyz
 |
 |      >>> df.replace(regex=r'^ba.$', value='new')
 |              A    B
 |      0   new  abc
 |      1   foo  new
 |      2  bait  xyz
 |
 |      >>> df.replace(regex={r'^ba.$': 'new', 'foo': 'xyz'})
 |              A    B
 |      0   new  abc
 |      1   xyz  new
 |      2  bait  xyz
 |
 |      >>> df.replace(regex=[r'^ba.$', 'foo'], value='new')
 |              A    B
 |      0   new  abc
 |      1   new  new
 |      2  bait  xyz
 |
 |      Compare the behavior of ``s.replace({'a': None})`` and
 |      ``s.replace('a', None)`` to understand the peculiarities
 |      of the `to_replace` parameter:
 |
 |      >>> s = pd.Series([10, 'a', 'a', 'b', 'a'])
 |
 |      When one uses a dict as the `to_replace` value, it is like the
 |      value(s) in the dict are equal to the `value` parameter.
 |      ``s.replace({'a': None})`` is equivalent to
 |      ``s.replace(to_replace={'a': None}, value=None, method=None)``:
 |
 |      >>> s.replace({'a': None})
 |      0      10
 |      1    None
 |      2    None
 |      3       b
 |      4    None
 |      dtype: object
 |
 |      When ``value=None`` and `to_replace` is a scalar, list or
 |      tuple, `replace` uses the method parameter (default 'pad') to do the
 |      replacement. So this is why the 'a' values are being replaced by 10
 |      in rows 1 and 2 and 'b' in row 4 in this case.
 |      The command ``s.replace('a', None)`` is actually equivalent to
 |      ``s.replace(to_replace='a', value=None, method='pad')``:
 |
 |      >>> s.replace('a', None)
 |      0    10
 |      1    10
 |      2    10
 |      3     b
 |      4     b
 |      dtype: object
 |
 |  resample(self, rule, axis=0, closed: 'str | None' = None, label: 'str | None' =
None, convention: 'str' = 'start', kind: 'str | None' = None, loffset=None, base:
'int | None' = None, on=None, level=None, origin: 'str | TimestampConvertibleTypes' =
'start_day', offset: 'TimedeltaConvertibleTypes | None' = None) -> 'Resampler'
 |      Resample time-series data.
 |
 |      Convenience method for frequency conversion and resampling of time series.
 |      The object must have a datetime-like index (`DatetimeIndex`, `PeriodIndex`,
 |      or `TimedeltaIndex`), or the caller must pass the label of a datetime-like
 |      series/index to the ``on``/``level`` keyword parameter.
 |
 |      Parameters
 |      ----------
 |      rule : DateOffset, Timedelta or str
 |          The offset string or object representing target conversion.
 |      axis : {0 or 'index', 1 or 'columns'}, default 0
 |          Which axis to use for up- or down-sampling. For `Series` this
 |          will default to 0, i.e. along the rows. Must be
 |          `DatetimeIndex`, `TimedeltaIndex` or `PeriodIndex`.
 |      closed : {'right', 'left'}, default None
 |          Which side of bin interval is closed. The default is 'left'
 |          for all frequency offsets except for 'M', 'A', 'Q', 'BM',
 |          'BA', 'BQ', and 'W' which all have a default of 'right'.
 |      label : {'right', 'left'}, default None
 |          Which bin edge label to label bucket with. The default is 'left'
 |          for all frequency offsets except for 'M', 'A', 'Q', 'BM',
 |          'BA', 'BQ', and 'W' which all have a default of 'right'.
 |      convention : {'start', 'end', 's', 'e'}, default 'start'
 |          For `PeriodIndex` only, controls whether to use the start or
 |          end of `rule`.
 |      kind : {'timestamp', 'period'}, optional, default None
 |          Pass 'timestamp' to convert the resulting index to a
 |          `DateTimeIndex` or 'period' to convert it to a `PeriodIndex`.
 |          By default the input representation is retained.
 |      loffset : timedelta, default None
 |          Adjust the resampled time labels.
 |
 |          .. deprecated:: 1.1.0
 |              You should add the loffset to the `df.index` after the resample.
 |              See below.
 |
 |      base : int, default 0
 |          For frequencies that evenly subdivide 1 day, the "origin" of the
 |          aggregated intervals. For example, for '5min' frequency, base could
 |          range from 0 through 4. Defaults to 0.
 |
 |          .. deprecated:: 1.1.0
 |              The new arguments that you should use are 'offset' or 'origin'.
 |
 |      on : str, optional
 |          For a DataFrame, column to use instead of index for resampling.
 |          Column must be datetime-like.
 |      level : str or int, optional
 |          For a MultiIndex, level (name or number) to use for
 |          resampling. `level` must be datetime-like.
 |      origin : {'epoch', 'start', 'start_day', 'end', 'end_day'}, Timestamp
 |          or str, default 'start_day'
 |          The timestamp on which to adjust the grouping. The timezone of origin
 |          must match the timezone of the index.
 |          If a timestamp is not used, these values are also supported:
 |
 |          - 'epoch': `origin` is 1970-01-01
 |          - 'start': `origin` is the first value of the timeseries
 |          - 'start_day': `origin` is the first day at midnight of the timeseries
 |
 |          .. versionadded:: 1.1.0
 |
 |          - 'end': `origin` is the last value of the timeseries
 |          - 'end_day': `origin` is the ceiling midnight of the last day
 |
 |          .. versionadded:: 1.3.0
 |
 |      offset : Timedelta or str, default is None
 |          An offset timedelta added to the origin.
 |
 |          .. versionadded:: 1.1.0
 |
 |      Returns
 |      -------
 |      pandas.core.Resampler
 |          :class:`~pandas.core.Resampler` object.
 |
 |      See Also
 |      --------
 |      Series.resample : Resample a Series.
 |      DataFrame.resample : Resample a DataFrame.
 |      groupby : Group DataFrame by mapping, function, label, or list of labels.
 |      asfreq : Reindex a DataFrame with the given frequency without grouping.
 |      
 |      Notes
 |      -----
 |      See the `user guide
 |      <https://pandas.pydata.org/pandas-
docs/stable/user_guide/timeseries.html#resampling>`__
 |      for more.
 |      
 |      To learn more about the offset strings, please see `this link
 |      <https://pandas.pydata.org/pandas-
docs/stable/user_guide/timeseries.html#dateoffset-objects>`__.
 |      
 |      Examples
 |      --------
 |      Start by creating a series with 9 one minute timestamps.
 |      
 |      >>> index = pd.date_range('1/1/2000', periods=9, freq='T')
 |      >>> series = pd.Series(range(9), index=index)
 |      >>> series
 |      2000-01-01 00:00:00    0
 |      2000-01-01 00:01:00    1
 |      2000-01-01 00:02:00    2
 |      2000-01-01 00:03:00    3
 |      2000-01-01 00:04:00    4
 |      2000-01-01 00:05:00    5
 |      2000-01-01 00:06:00    6
 |      2000-01-01 00:07:00    7
 |      2000-01-01 00:08:00    8
 |      Freq: T, dtype: int64
 |      
 |      Downsample the series into 3 minute bins and sum the values
 |      of the timestamps falling into a bin.
 |      
 |      >>> series.resample('3T').sum()
 |      2000-01-01 00:00:00     3
 |      2000-01-01 00:03:00    12
 |      2000-01-01 00:06:00    21
 |      Freq: 3T, dtype: int64
 |      
 |      Downsample the series into 3 minute bins as above, but label each
 |      bin using the right edge instead of the left. Please note that the
 |      value in the bucket used as the label is not included in the bucket,
 |      which it labels. For example, in the original series the
 |      bucket ``2000-01-01 00:03:00`` contains the value 3, but the summed
 |      value in the resampled bucket with the label ``2000-01-01 00:03:00``
 |      does not include 3 (if it did, the summed value would be 6, not 3).
 |      To include this value close the right side of the bin interval as
 |      illustrated in the example below this one.
 |      
 |      >>> series.resample('3T', label='right').sum()
 |      2000-01-01 00:03:00     3
 |      2000-01-01 00:06:00    12
 |      2000-01-01 00:09:00    21
 |      Freq: 3T, dtype: int64
 |      
 |      Downsample the series into 3 minute bins as above, but close the right
 |      side of the bin interval.
 |      
 |      >>> series.resample('3T', label='right', closed='right').sum()
 |      2000-01-01 00:00:00     0
 |      2000-01-01 00:03:00     6
 |      2000-01-01 00:06:00    15
 |      2000-01-01 00:09:00    15
 |      Freq: 3T, dtype: int64
 |      
 |      Upsample the series into 30 second bins.
 |      
 |      >>> series.resample('30S').asfreq()[0:5]   # Select first 5 rows
 |      2000-01-01 00:00:00   0.0
 |      2000-01-01 00:00:30   NaN
 |      2000-01-01 00:01:00   1.0
 |      2000-01-01 00:01:30   NaN
 |      2000-01-01 00:02:00   2.0
 |      Freq: 30S, dtype: float64
 |      
 |      Upsample the series into 30 second bins and fill the ``NaN``
 |      values using the ``pad`` method.
 |      
 |      >>> series.resample('30S').pad()[0:5]
 |      2000-01-01 00:00:00    0
 |      2000-01-01 00:00:30    0
 |      2000-01-01 00:01:00    1
 |      2000-01-01 00:01:30    1
 |      2000-01-01 00:02:00    2
 |      Freq: 30S, dtype: int64
 |      
 |      Upsample the series into 30 second bins and fill the
 |      ``NaN`` values using the ``bfill`` method.
 |
 |      >>> series.resample('30S').bfill()[0:5]
 |      2000-01-01 00:00:00    0
 |      2000-01-01 00:00:30    1
 |      2000-01-01 00:01:00    1
 |      2000-01-01 00:01:30    2
 |      2000-01-01 00:02:00    2
 |      Freq: 30S, dtype: int64
 |
 |      Pass a custom function via ``apply``
 |
 |      >>> def custom_resampler(arraylike):
 |      ...     return np.sum(arraylike) + 5
 |      ...
 |      >>> series.resample('3T').apply(custom_resampler)
 |      2000-01-01 00:00:00     8
 |      2000-01-01 00:03:00    17
 |      2000-01-01 00:06:00    26
 |      Freq: 3T, dtype: int64
 |
 |      For a Series with a PeriodIndex, the keyword `convention` can be
 |      used to control whether to use the start or end of `rule`.
 |
 |      Resample a year by quarter using 'start' `convention`. Values are
 |      assigned to the first quarter of the period.
 |
 |      >>> s = pd.Series([1, 2], index=pd.period_range('2012-01-01',
 |      ...                                             freq='A',
 |      ...                                             periods=2))
 |      >>> s
 |      2012    1
 |      2013    2
 |      Freq: A-DEC, dtype: int64
 |      >>> s.resample('Q', convention='start').asfreq()
 |      2012Q1    1.0
 |      2012Q2    NaN
 |      2012Q3    NaN
 |      2012Q4    NaN
 |      2013Q1    2.0
 |      2013Q2    NaN
 |      2013Q3    NaN
 |      2013Q4    NaN
 |      Freq: Q-DEC, dtype: float64
 |
 |      Resample quarters by month using 'end' `convention`. Values are
 |      assigned to the last month of the period.
 |
 |      >>> q = pd.Series([1, 2, 3, 4], index=pd.period_range('2018-01-01',
 |      ...                                                   freq='Q',
 |      ...                                                   periods=4))
 |      >>> q
 |      2018Q1    1
 |      2018Q2    2
 |      2018Q3    3
 |      2018Q4    4
 |      Freq: Q-DEC, dtype: int64
 |      >>> q.resample('M', convention='end').asfreq()
 |      2018-03    1.0
 |      2018-04    NaN
 |      2018-05    NaN
 |      2018-06    2.0
 |      2018-07    NaN
 |      2018-08    NaN
 |      2018-09    3.0
 |      2018-10    NaN
 |      2018-11    NaN
 |      2018-12    4.0
 |      Freq: M, dtype: float64
 |
 |      For DataFrame objects, the keyword `on` can be used to specify the
 |      column instead of the index for resampling.
 |
 |      >>> d = {'price': [10, 11, 9, 13, 14, 18, 17, 19],
 |      ...      'volume': [50, 60, 40, 100, 50, 100, 40, 50]}
 |      >>> df = pd.DataFrame(d)
 |      >>> df['week_starting'] = pd.date_range('01/01/2018',
 |      ...                                     periods=8,
 |      ...                                     freq='W')
 |      >>> df
 |         price  volume week_starting
 |      0     10      50    2018-01-07
 |      1     11      60    2018-01-14
 |      2      9      40    2018-01-21
 |      3     13     100    2018-01-28
 |      4     14      50    2018-02-04
```

```
 |      5     18     100    2018-02-11
 |      6     17      40    2018-02-18
 |      7     19      50    2018-02-25
 |      >>> df.resample('M', on='week_starting').mean()
 |                     price  volume
 |      week_starting
 |      2018-01-31     10.75    62.5
 |      2018-02-28     17.00    60.0
 |
 |      For a DataFrame with MultiIndex, the keyword `level` can be used to
 |      specify on which level the resampling needs to take place.
 |
 |      >>> days = pd.date_range('1/1/2000', periods=4, freq='D')
 |      >>> d2 = {'price': [10, 11, 9, 13, 14, 18, 17, 19],
 |      ...       'volume': [50, 60, 40, 100, 50, 100, 40, 50]}
 |      >>> df2 = pd.DataFrame(
 |      ...     d2,
 |      ...     index=pd.MultiIndex.from_product(
 |      ...         [days, ['morning', 'afternoon']]
 |      ...     )
 |      ... )
 |      >>> df2
 |                            price  volume
 |      2000-01-01 morning       10      50
 |                 afternoon     11      60
 |      2000-01-02 morning        9      40
 |                 afternoon     13     100
 |      2000-01-03 morning       14      50
 |                 afternoon     18     100
 |      2000-01-04 morning       17      40
 |                 afternoon     19      50
 |      >>> df2.resample('D', level=0).sum()
 |                  price  volume
 |      2000-01-01     21     110
 |      2000-01-02     22     140
 |      2000-01-03     32     150
 |      2000-01-04     36      90
 |
 |      If you want to adjust the start of the bins based on a fixed timestamp:
 |
 |      >>> start, end = '2000-10-01 23:30:00', '2000-10-02 00:30:00'
 |      >>> rng = pd.date_range(start, end, freq='7min')
 |      >>> ts = pd.Series(np.arange(len(rng)) * 3, index=rng)
 |      >>> ts
 |      2000-10-01 23:30:00     0
 |      2000-10-01 23:37:00     3
 |      2000-10-01 23:44:00     6
 |      2000-10-01 23:51:00     9
 |      2000-10-01 23:58:00    12
 |      2000-10-02 00:05:00    15
 |      2000-10-02 00:12:00    18
 |      2000-10-02 00:19:00    21
 |      2000-10-02 00:26:00    24
 |      Freq: 7T, dtype: int64
 |
 |      >>> ts.resample('17min').sum()
 |      2000-10-01 23:14:00     0
 |      2000-10-01 23:31:00     9
 |      2000-10-01 23:48:00    21
 |      2000-10-02 00:05:00    54
 |      2000-10-02 00:22:00    24
 |      Freq: 17T, dtype: int64
 |
 |      >>> ts.resample('17min', origin='epoch').sum()
 |      2000-10-01 23:18:00     0
 |      2000-10-01 23:35:00    18
 |      2000-10-01 23:52:00    27
 |      2000-10-02 00:09:00    39
 |      2000-10-02 00:26:00    24
 |      Freq: 17T, dtype: int64
 |
 |      >>> ts.resample('17min', origin='2000-01-01').sum()
 |      2000-10-01 23:24:00     3
 |      2000-10-01 23:41:00    15
 |      2000-10-01 23:58:00    45
 |      2000-10-02 00:15:00    45
 |      Freq: 17T, dtype: int64
 |
 |      If you want to adjust the start of the bins with an `offset` Timedelta, the
two
 |      following lines are equivalent:
 |
 |      >>> ts.resample('17min', origin='start').sum()
 |      2000-10-01 23:30:00     9
 |      2000-10-01 23:47:00    21
 |      2000-10-02 00:04:00    54
 |      2000-10-02 00:21:00    24
 |      Freq: 17T, dtype: int64
 |
 |      >>> ts.resample('17min', offset='23h30min').sum()
 |      2000-10-01 23:30:00     9
 |      2000-10-01 23:47:00    21
 |      2000-10-02 00:04:00    54
 |      2000-10-02 00:21:00    24
 |      Freq: 17T, dtype: int64
 |
 |      If you want to take the largest Timestamp as the end of the bins:
 |
 |      >>> ts.resample('17min', origin='end').sum()
 |      2000-10-01 23:35:00     0
 |      2000-10-01 23:52:00    18
 |      2000-10-02 00:09:00    27
 |      2000-10-02 00:26:00    63
 |      Freq: 17T, dtype: int64
 |
 |      In contrast with the `start_day`, you can use `end_day` to take the ceiling
 |      midnight of the largest Timestamp as the end of the bins and drop the bins
 |      not containing data:
 |
 |      >>> ts.resample('17min', origin='end_day').sum()
 |      2000-10-01 23:38:00     3
 |      2000-10-01 23:55:00    15
 |      2000-10-02 00:12:00    45
 |      2000-10-02 00:29:00    45
 |      Freq: 17T, dtype: int64
 |
 |      To replace the use of the deprecated `base` argument, you can now use
`offset`,
 |      in this example it is equivalent to have `base=2`:
 |
 |      >>> ts.resample('17min', offset='2min').sum()
 |      2000-10-01 23:16:00     0
 |      2000-10-01 23:33:00     9
 |      2000-10-01 23:50:00    36
 |      2000-10-02 00:07:00    39
 |      2000-10-02 00:24:00    24
 |      Freq: 17T, dtype: int64
 |
 |      To replace the use of the deprecated `loffset` argument:
 |
 |      >>> from pandas.tseries.frequencies import to_offset
 |      >>> loffset = '19min'
 |      >>> ts_out = ts.resample('17min').sum()
 |      >>> ts_out.index = ts_out.index + to_offset(loffset)
 |      >>> ts_out
 |      2000-10-01 23:33:00     0
 |      2000-10-01 23:50:00     9
 |      2000-10-02 00:07:00    21
 |      2000-10-02 00:24:00    54
 |      2000-10-02 00:41:00    24
 |      Freq: 17T, dtype: int64
 |
 |  reset_index(self, level: 'Hashable | Sequence[Hashable] | None' = None, drop:
'bool' = False, inplace: 'bool' = False, col_level: 'Hashable' = 0, col_fill:
'Hashable' = '') -> 'DataFrame | None'
 |      Reset the index, or a level of it.
 |
 |      Reset the index of the DataFrame, and use the default one instead.
 |      If the DataFrame has a MultiIndex, this method can remove one or more
 |      levels.
 |
 |      Parameters
 |      ----------
 |      level : int, str, tuple, or list, default None
 |          Only remove the given levels from the index. Removes all levels by
 |          default.
 |      drop : bool, default False
 |          Do not try to insert index into dataframe columns. This resets
 |          the index to the default integer index.
 |      inplace : bool, default False
 |          Modify the DataFrame in place (do not create a new object).
 |      col_level : int or str, default 0
 |          If the columns have multiple levels, determines which level the
 |          labels are inserted into. By default it is inserted into the first
 |          level.
 |      col_fill : object, default ''
 |          If the columns have multiple levels, determines how the other
 |          levels are named. If None then the index name is repeated.
 |
 |      Returns
 |      -------
 |      DataFrame or None
 |          DataFrame with the new index or None if ``inplace=True``.
 |
 |      See Also
 |      --------
 |      DataFrame.set_index : Opposite of reset_index.
 |      DataFrame.reindex : Change to new indices or expand indices.
 |      DataFrame.reindex_like : Change to same indices as other DataFrame.
 |      
 |      Examples
 |      --------
 |      >>> df = pd.DataFrame([('bird', 389.0),
 |      ...                    ('bird', 24.0),
 |      ...                    ('mammal', 80.5),
 |      ...                    ('mammal', np.nan)],
 |      ...                   index=['falcon', 'parrot', 'lion', 'monkey'],
 |      ...                   columns=('class', 'max_speed'))
 |      >>> df
 |               class  max_speed
 |      falcon    bird      389.0
 |      parrot    bird       24.0
 |      lion    mammal       80.5
 |      monkey  mammal        NaN
 |      
 |      When we reset the index, the old index is added as a column, and a
 |      new sequential index is used:
 |      
 |      >>> df.reset_index()
 |          index   class  max_speed
 |      0  falcon    bird      389.0
 |      1  parrot    bird       24.0
 |      2    lion  mammal       80.5
 |      3  monkey  mammal        NaN
 |      
 |      We can use the `drop` parameter to avoid the old index being added as
 |      a column:
 |      
 |      >>> df.reset_index(drop=True)
 |          class  max_speed
 |      0    bird      389.0
 |      1    bird       24.0
 |      2  mammal       80.5
 |      3  mammal        NaN
 |      
 |      You can also use `reset_index` with `MultiIndex`.
 |      
 |      >>> index = pd.MultiIndex.from_tuples([('bird', 'falcon'),
 |      ...                                    ('bird', 'parrot'),
 |      ...                                    ('mammal', 'lion'),
 |      ...                                    ('mammal', 'monkey')],
 |      ...                                   names=['class', 'name'])
 |      >>> columns = pd.MultiIndex.from_tuples([('speed', 'max'),
 |      ...                                      ('species', 'type')])
 |      >>> df = pd.DataFrame([(389.0, 'fly'),
 |      ...                    ( 24.0, 'fly'),
 |      ...                    ( 80.5, 'run'),
 |      ...                    (np.nan, 'jump')],
 |      ...                   index=index,
 |      ...                   columns=columns)
 |      >>> df
 |                     speed species
 |                       max    type
 |      class  name
 |      bird   falcon  389.0     fly
 |             parrot   24.0     fly
 |      mammal lion     80.5     run
 |             monkey    NaN    jump
 |      
 |      If the index has multiple levels, we can reset a subset of them:
 |      
 |      >>> df.reset_index(level='class')
 |               class  speed species
 |                        max    type
 |      name
 |      falcon    bird  389.0     fly
 |      parrot    bird   24.0     fly
 |      lion    mammal   80.5     run
 |      monkey  mammal    NaN    jump
 |      
 |      If we are not dropping the index, by default, it is placed in the top
 |      level. We can place it in another level:
 |      
 |      >>> df.reset_index(level='class', col_level=1)
 |                      speed species
 |               class    max    type
 |      name
 |      falcon    bird  389.0     fly
 |      parrot    bird   24.0     fly
 |      lion    mammal   80.5     run
 |      monkey  mammal    NaN    jump
 |
 |      When the index is inserted under another level, we can specify under
 |      which one with the parameter `col_fill`:
 |
 |      >>> df.reset_index(level='class', col_level=1, col_fill='species')
 |                    species  speed species
 |                      class    max    type
 |      name
 |      falcon           bird  389.0     fly
 |      parrot           bird   24.0     fly
 |      lion           mammal   80.5     run
 |      monkey         mammal    NaN    jump
 |
 |      If we specify a nonexistent level for `col_fill`, it is created:
 |
 |      >>> df.reset_index(level='class', col_level=1, col_fill='genus')
 |                      genus  speed species
 |                      class    max    type
 |      name
 |      falcon           bird  389.0     fly
 |      parrot           bird   24.0     fly
 |      lion           mammal   80.5     run
 |      monkey         mammal    NaN    jump
 |
 |  rfloordiv(self, other, axis='columns', level=None, fill_value=None)
 |      Get Integer division of dataframe and other, element-wise (binary operator
`rfloordiv`).
 |
 |      Equivalent to ``other // dataframe``, but with support to substitute a
fill_value
 |      for missing data in one of the inputs. With reverse version, `floordiv`.
 |
 |      Among flexible wrappers (`add`, `sub`, `mul`, `div`, `mod`, `pow`) to
 |      arithmetic operators: `+`, `-`, `*`, `/`, `//`, `%`, `**`.
 |
 |      Parameters
 |      ----------
 |      other : scalar, sequence, Series, or DataFrame
 |          Any single or multiple element data structure, or list-like object.
 |      axis : {0 or 'index', 1 or 'columns'}
 |          Whether to compare by the index (0 or 'index') or columns
 |          (1 or 'columns'). For Series input, axis to match Series index on.
 |      level : int or label
 |          Broadcast across a level, matching Index values on the
 |          passed MultiIndex level.
 |      fill_value : float or None, default None
 |          Fill existing missing (NaN) values, and any new element needed for
 |          successful DataFrame alignment, with this value before computation.
 |          If data in both corresponding DataFrame locations is missing
 |          the result will be missing.
 |
 |      Returns
 |      -------
 |      DataFrame
 |          Result of the arithmetic operation.
 |
 |      See Also
 |      --------
 |      DataFrame.add : Add DataFrames.
 |      DataFrame.sub : Subtract DataFrames.
 |      DataFrame.mul : Multiply DataFrames.
 |      DataFrame.div : Divide DataFrames (float division).
 |      DataFrame.truediv : Divide DataFrames (float division).
 |      DataFrame.floordiv : Divide DataFrames (integer division).
 |      DataFrame.mod : Calculate modulo (remainder after division).
 |      DataFrame.pow : Calculate exponential power.
 |
 |      Notes
 |      -----
 |      Mismatched indices will be unioned together.
 |
 |      Examples
 |      --------
 |      >>> df = pd.DataFrame({'angles': [0, 3, 4],
 |      ...                    'degrees': [360, 180, 360]},
 |      ...                   index=['circle', 'triangle', 'rectangle'])
 |      >>> df
 |                 angles  degrees
 |      circle          0      360
 |      triangle        3      180
 |      rectangle       4      360
 |
 |      Add a scalar with operator version which return the same
 |      results.
 |
 |      >>> df + 1
 |                 angles  degrees
 |      circle          1      361
 |      triangle        4      181
 |      rectangle       5      361
 |
 |      >>> df.add(1)
 |                 angles  degrees
 |      circle          1      361
 |      triangle        4      181
 |      rectangle       5      361
 |
 |      Divide by constant with reverse version.
 |
 |      >>> df.div(10)
 |                 angles  degrees
 |      circle        0.0     36.0
 |      triangle      0.3     18.0
 |      rectangle     0.4     36.0
 |
 |      >>> df.rdiv(10)
 |                   angles   degrees
 |      circle          inf  0.027778
 |      triangle   3.333333  0.055556
 |      rectangle  2.500000  0.027778
 |
 |      Subtract a list and Series by axis with operator version.
 |
 |      >>> df - [1, 2]
 |                 angles  degrees
 |      circle         -1      358
 |      triangle        2      178
 |      rectangle       3      358
 |
 |      >>> df.sub([1, 2], axis='columns')
 |                 angles  degrees
 |      circle         -1      358
 |      triangle        2      178
 |      rectangle       3      358
 |
 |      >>> df.sub(pd.Series([1, 1, 1], index=['circle', 'triangle', 'rectangle']),
 |      ...        axis='index')
 |                 angles  degrees
 |      circle         -1      359
 |      triangle        2      179
 |      rectangle       3      359
 |
 |      Multiply a DataFrame of different shape with operator version.
 |
 |      >>> other = pd.DataFrame({'angles': [0, 3, 4]},
 |      ...                      index=['circle', 'triangle', 'rectangle'])
 |      >>> other
 |                 angles
 |      circle          0
 |      triangle        3
 |      rectangle       4
 |
 |      >>> df * other
 |                 angles  degrees
 |      circle          0      NaN
 |      triangle        9      NaN
 |      rectangle      16      NaN
 |
 |      >>> df.mul(other, fill_value=0)
 |                 angles  degrees
 |      circle          0      0.0
 |      triangle        9      0.0
 |      rectangle      16      0.0
 |
 |      Divide by a MultiIndex by level.
 |
 |      >>> df_multiindex = pd.DataFrame({'angles': [0, 3, 4, 4, 5, 6],
 |      ...                              'degrees': [360, 180, 360, 360, 540, 720]},
 |      ...                             index=[['A', 'A', 'A', 'B', 'B', 'B'],
 |      ...                                    ['circle', 'triangle', 'rectangle',
 |      ...                                     'square', 'pentagon', 'hexagon']])
 |      >>> df_multiindex
 |                   angles  degrees
 |      A circle          0      360
 |        triangle        3      180
 |        rectangle       4      360
 |      B square          4      360
 |        pentagon        5      540
 |        hexagon         6      720
 |
 |      >>> df.div(df_multiindex, level=1, fill_value=0)
 |                   angles  degrees
 |      A circle        NaN      1.0
 |        triangle      1.0      1.0
```

```
 |        rectangle     1.0      1.0
 |      B square        0.0      0.0
 |        pentagon      0.0      0.0
 |        hexagon       0.0      0.0
 |
 |  rmod(self, other, axis='columns', level=None, fill_value=None)
 |      Get Modulo of dataframe and other, element-wise (binary operator `rmod`).
 |
 |      Equivalent to ``other % dataframe``, but with support to substitute a
fill_value
 |      for missing data in one of the inputs. With reverse version, `mod`.
 |
 |      Among flexible wrappers (`add`, `sub`, `mul`, `div`, `mod`, `pow`) to
 |      arithmetic operators: `+`, `-`, `*`, `/`, `//`, `%`, `**`.
 |
 |      Parameters
 |      ----------
 |      other : scalar, sequence, Series, or DataFrame
 |          Any single or multiple element data structure, or list-like object.
 |      axis : {0 or 'index', 1 or 'columns'}
 |          Whether to compare by the index (0 or 'index') or columns
 |          (1 or 'columns'). For Series input, axis to match Series index on.
 |      level : int or label
 |          Broadcast across a level, matching Index values on the
 |          passed MultiIndex level.
 |      fill_value : float or None, default None
 |          Fill existing missing (NaN) values, and any new element needed for
 |          successful DataFrame alignment, with this value before computation.
 |          If data in both corresponding DataFrame locations is missing
 |          the result will be missing.
 |
 |      Returns
 |      -------
 |      DataFrame
 |          Result of the arithmetic operation.
 |
 |      See Also
 |      --------
 |      DataFrame.add : Add DataFrames.
 |      DataFrame.sub : Subtract DataFrames.
 |      DataFrame.mul : Multiply DataFrames.
 |      DataFrame.div : Divide DataFrames (float division).
 |      DataFrame.truediv : Divide DataFrames (float division).
 |      DataFrame.floordiv : Divide DataFrames (integer division).
 |      DataFrame.mod : Calculate modulo (remainder after division).
 |      DataFrame.pow : Calculate exponential power.
 |
 |      Notes
 |      -----
 |      Mismatched indices will be unioned together.
 |
 |      Examples
 |      --------
 |      >>> df = pd.DataFrame({'angles': [0, 3, 4],
 |      ...                    'degrees': [360, 180, 360]},
 |      ...                   index=['circle', 'triangle', 'rectangle'])
 |      >>> df
 |                 angles  degrees
 |      circle          0      360
 |      triangle        3      180
 |      rectangle       4      360
 |
 |      Add a scalar with operator version which return the same
 |      results.
 |
 |      >>> df + 1
 |                 angles  degrees
 |      circle          1      361
 |      triangle        4      181
 |      rectangle       5      361
 |
 |      >>> df.add(1)
 |                 angles  degrees
 |      circle          1      361
 |      triangle        4      181
 |      rectangle       5      361
 |
 |      Divide by constant with reverse version.
 |
 |      >>> df.div(10)
 |                 angles  degrees
 |      circle        0.0     36.0
 |      triangle      0.3     18.0
 |      rectangle     0.4     36.0
 |
 |      >>> df.rdiv(10)
 |                   angles   degrees
 |      circle          inf  0.027778
 |      triangle   3.333333  0.055556
 |      rectangle  2.500000  0.027778
 |
 |      Subtract a list and Series by axis with operator version.
 |
 |      >>> df - [1, 2]
 |                 angles  degrees
 |      circle         -1      358
 |      triangle        2      178
 |      rectangle       3      358
 |
 |      >>> df.sub([1, 2], axis='columns')
 |                 angles  degrees
 |      circle         -1      358
 |      triangle        2      178
 |      rectangle       3      358
 |
 |      >>> df.sub(pd.Series([1, 1, 1], index=['circle', 'triangle', 'rectangle']),
 |      ...        axis='index')
 |                 angles  degrees
 |      circle         -1      359
 |      triangle        2      179
 |      rectangle       3      359
 |
 |      Multiply a DataFrame of different shape with operator version.
 |
 |      >>> other = pd.DataFrame({'angles': [0, 3, 4]},
 |      ...                      index=['circle', 'triangle', 'rectangle'])
 |      >>> other
 |                 angles
 |      circle          0
 |      triangle        3
 |      rectangle       4
 |
 |      >>> df * other
 |                 angles  degrees
 |      circle          0      NaN
 |      triangle        9      NaN
 |      rectangle      16      NaN
 |
 |      >>> df.mul(other, fill_value=0)
 |                 angles  degrees
 |      circle          0      0.0
 |      triangle        9      0.0
 |      rectangle      16      0.0
 |
 |      Divide by a MultiIndex by level.
 |
 |      >>> df_multiindex = pd.DataFrame({'angles': [0, 3, 4, 4, 5, 6],
 |      ...                              'degrees': [360, 180, 360, 360, 540, 720]},
 |      ...                             index=[['A', 'A', 'A', 'B', 'B', 'B'],
 |      ...                                    ['circle', 'triangle', 'rectangle',
 |      ...                                     'square', 'pentagon', 'hexagon']])
 |      >>> df_multiindex
 |                   angles  degrees
 |      A circle          0      360
 |        triangle        3      180
 |        rectangle       4      360
 |      B square          4      360
 |        pentagon        5      540
 |        hexagon         6      720
 |
 |      >>> df.div(df_multiindex, level=1, fill_value=0)
 |                   angles  degrees
 |      A circle        NaN      1.0
 |        triangle      1.0      1.0
 |        rectangle     1.0      1.0
 |      B square        0.0      0.0
 |        pentagon      0.0      0.0
 |        hexagon       0.0      0.0
 |
 |  rmul(self, other, axis='columns', level=None, fill_value=None)
 |      Get Multiplication of dataframe and other, element-wise (binary operator
`rmul`).
 |
 |      Equivalent to ``other * dataframe``, but with support to substitute a
fill_value
 |      for missing data in one of the inputs. With reverse version, `mul`.
 |
 |      Among flexible wrappers (`add`, `sub`, `mul`, `div`, `mod`, `pow`) to
 |      arithmetic operators: `+`, `-`, `*`, `/`, `//`, `%`, `**`.
 |
 |      Parameters
 |      ----------
 |      other : scalar, sequence, Series, or DataFrame
 |          Any single or multiple element data structure, or list-like object.
 |      axis : {0 or 'index', 1 or 'columns'}
 |          Whether to compare by the index (0 or 'index') or columns
 |          (1 or 'columns'). For Series input, axis to match Series index on.
 |      level : int or label
 |          Broadcast across a level, matching Index values on the
 |          passed MultiIndex level.
 |      fill_value : float or None, default None
 |          Fill existing missing (NaN) values, and any new element needed for
 |          successful DataFrame alignment, with this value before computation.
 |          If data in both corresponding DataFrame locations is missing
 |          the result will be missing.
 |
 |      Returns
 |      -------
 |      DataFrame
 |          Result of the arithmetic operation.
 |
 |      See Also
 |      --------
 |      DataFrame.add : Add DataFrames.
 |      DataFrame.sub : Subtract DataFrames.
 |      DataFrame.mul : Multiply DataFrames.
 |      DataFrame.div : Divide DataFrames (float division).
 |      DataFrame.truediv : Divide DataFrames (float division).
 |      DataFrame.floordiv : Divide DataFrames (integer division).
 |      DataFrame.mod : Calculate modulo (remainder after division).
 |      DataFrame.pow : Calculate exponential power.
 |
 |      Notes
 |      -----
 |      Mismatched indices will be unioned together.
 |
 |      Examples
 |      --------
 |      >>> df = pd.DataFrame({'angles': [0, 3, 4],
 |      ...                    'degrees': [360, 180, 360]},
 |      ...                   index=['circle', 'triangle', 'rectangle'])
 |      >>> df
 |                 angles  degrees
 |      circle          0      360
 |      triangle        3      180
 |      rectangle       4      360
 |
 |      Add a scalar with operator version which return the same
 |      results.
 |
 |      >>> df + 1
 |                 angles  degrees
 |      circle          1      361
 |      triangle        4      181
 |      rectangle       5      361
 |
 |      >>> df.add(1)
 |                 angles  degrees
 |      circle          1      361
 |      triangle        4      181
 |      rectangle       5      361
 |
 |      Divide by constant with reverse version.
 |
 |      >>> df.div(10)
 |                 angles  degrees
 |      circle        0.0     36.0
 |      triangle      0.3     18.0
 |      rectangle     0.4     36.0
 |
 |      >>> df.rdiv(10)
 |                   angles   degrees
 |      circle          inf  0.027778
 |      triangle   3.333333  0.055556
 |      rectangle  2.500000  0.027778
 |
 |      Subtract a list and Series by axis with operator version.
 |
 |      >>> df - [1, 2]
 |                 angles  degrees
 |      circle         -1      358
 |      triangle        2      178
 |      rectangle       3      358
 |
 |      >>> df.sub([1, 2], axis='columns')
 |                 angles  degrees
 |      circle         -1      358
 |      triangle        2      178
 |      rectangle       3      358
 |
 |      >>> df.sub(pd.Series([1, 1, 1], index=['circle', 'triangle', 'rectangle']),
 |      ...        axis='index')
 |                 angles  degrees
 |      circle         -1      359
 |      triangle        2      179
 |      rectangle       3      359
 |
 |      Multiply a DataFrame of different shape with operator version.
 |
 |      >>> other = pd.DataFrame({'angles': [0, 3, 4]},
 |      ...                      index=['circle', 'triangle', 'rectangle'])
 |      >>> other
 |                 angles
 |      circle          0
 |      triangle        3
 |      rectangle       4
 |
 |      >>> df * other
 |                 angles  degrees
 |      circle          0      NaN
 |      triangle        9      NaN
 |      rectangle      16      NaN
 |
 |      >>> df.mul(other, fill_value=0)
 |                 angles  degrees
 |      circle          0      0.0
 |      triangle        9      0.0
 |      rectangle      16      0.0
 |
 |      Divide by a MultiIndex by level.
 |
 |      >>> df_multiindex = pd.DataFrame({'angles': [0, 3, 4, 4, 5, 6],
 |      ...                              'degrees': [360, 180, 360, 360, 540, 720]},
 |      ...                             index=[['A', 'A', 'A', 'B', 'B', 'B'],
 |      ...                                    ['circle', 'triangle', 'rectangle',
 |      ...                                     'square', 'pentagon', 'hexagon']])
 |      >>> df_multiindex
 |                   angles  degrees
 |      A circle          0      360
 |        triangle        3      180
 |        rectangle       4      360
 |      B square          4      360
 |        pentagon        5      540
 |        hexagon         6      720
 |
 |      >>> df.div(df_multiindex, level=1, fill_value=0)
 |                   angles  degrees
 |      A circle        NaN      1.0
 |        triangle      1.0      1.0
 |        rectangle     1.0      1.0
 |      B square        0.0      0.0
 |        pentagon      0.0      0.0
 |        hexagon       0.0      0.0
 |
 |  round(self, decimals: 'int | dict[IndexLabel, int] | Series' = 0, *args,
**kwargs) -> 'DataFrame'
 |      Round a DataFrame to a variable number of decimal places.
 |
 |      Parameters
 |      ----------
 |      decimals : int, dict, Series
 |          Number of decimal places to round each column to. If an int is
 |          given, round each column to the same number of places.
 |          Otherwise dict and Series round to variable numbers of places.
 |          Column names should be in the keys if `decimals` is a
 |          dict-like, or in the index if `decimals` is a Series. Any
 |          columns not included in `decimals` will be left as is. Elements
 |          of `decimals` which are not columns of the input will be
 |          ignored.
 |      *args
 |          Additional keywords have no effect but might be accepted for
 |          compatibility with numpy.
 |      **kwargs
 |          Additional keywords have no effect but might be accepted for
 |          compatibility with numpy.
 |
 |      Returns
 |      -------
 |      DataFrame
 |          A DataFrame with the affected columns rounded to the specified
 |          number of decimal places.
 |
 |      See Also
 |      --------
 |      numpy.around : Round a numpy array to the given number of decimals.
 |      Series.round : Round a Series to the given number of decimals.
 |
 |      Examples
 |      --------
 |      >>> df = pd.DataFrame([(.21, .32), (.01, .67), (.66, .03), (.21, .18)],
 |      ...                   columns=['dogs', 'cats'])
 |      >>> df
 |          dogs  cats
 |      0  0.21  0.32
 |      1  0.01  0.67
 |      2  0.66  0.03
 |      3  0.21  0.18
 |      
 |      By providing an integer each column is rounded to the same number
 |      of decimal places
 |      
 |      >>> df.round(1)
 |          dogs  cats
 |      0   0.2   0.3
 |      1   0.0   0.7
 |      2   0.7   0.0
 |      3   0.2   0.2
 |      
 |      With a dict, the number of places for specific columns can be
 |      specified with the column names as key and the number of decimal
 |      places as value
 |      
 |      >>> df.round({'dogs': 1, 'cats': 0})
 |          dogs  cats
 |      0   0.2   0.0
 |      1   0.0   1.0
 |      2   0.7   0.0
 |      3   0.2   0.0
 |      
 |      Using a Series, the number of places for specific columns can be
 |      specified with the column names as index and the number of
 |      decimal places as value
 |      
 |      >>> decimals = pd.Series([0, 1], index=['cats', 'dogs'])
 |      >>> df.round(decimals)
 |          dogs  cats
 |      0   0.2   0.0
 |      1   0.0   1.0
 |      2   0.7   0.0
 |      3   0.2   0.0
 |  
 |  rpow(self, other, axis='columns', level=None, fill_value=None)
 |      Get Exponential power of dataframe and other, element-wise (binary operator
`rpow`).
 |      
 |      Equivalent to ``other ** dataframe``, but with support to substitute a
fill_value
 |      for missing data in one of the inputs. With reverse version, `pow`.
 |      
 |      Among flexible wrappers (`add`, `sub`, `mul`, `div`, `mod`, `pow`) to
 |      arithmetic operators: `+`, `-`, `*`, `/`, `//`, `%`, `**`.
 |      
 |      Parameters
 |      ----------
 |      other : scalar, sequence, Series, or DataFrame
 |          Any single or multiple element data structure, or list-like object.
 |      axis : {0 or 'index', 1 or 'columns'}
 |          Whether to compare by the index (0 or 'index') or columns
 |          (1 or 'columns'). For Series input, axis to match Series index on.
 |      level : int or label
 |          Broadcast across a level, matching Index values on the
 |          passed MultiIndex level.
 |      fill_value : float or None, default None
 |          Fill existing missing (NaN) values, and any new element needed for
 |          successful DataFrame alignment, with this value before computation.
 |          If data in both corresponding DataFrame locations is missing
 |          the result will be missing.
 |      
 |      Returns
 |      -------
 |      DataFrame
 |          Result of the arithmetic operation.
 |      
 |      See Also
 |      --------
 |      DataFrame.add : Add DataFrames.
 |      DataFrame.sub : Subtract DataFrames.
 |      DataFrame.mul : Multiply DataFrames.
 |      DataFrame.div : Divide DataFrames (float division).
 |      DataFrame.truediv : Divide DataFrames (float division).
 |      DataFrame.floordiv : Divide DataFrames (integer division).
 |      DataFrame.mod : Calculate modulo (remainder after division).
 |      DataFrame.pow : Calculate exponential power.
 |      
 |      Notes
 |      -----
 |      Mismatched indices will be unioned together.
 |
 |      Examples
 |      --------
 |      >>> df = pd.DataFrame({'angles': [0, 3, 4],
 |      ...                    'degrees': [360, 180, 360]},
 |      ...                   index=['circle', 'triangle', 'rectangle'])
 |      >>> df
 |                 angles  degrees
 |      circle          0      360
 |      triangle        3      180
 |      rectangle       4      360
 |
 |      Add a scalar with operator version which return the same
 |      results.
 |
 |      >>> df + 1
 |                 angles  degrees
 |      circle          1      361
 |      triangle        4      181
 |      rectangle       5      361
 |
 |      >>> df.add(1)
 |                 angles  degrees
 |      circle          1      361
 |      triangle        4      181
 |      rectangle       5      361
 |
 |      Divide by constant with reverse version.
 |
 |      >>> df.div(10)
 |                 angles  degrees
 |      circle        0.0     36.0
 |      triangle      0.3     18.0
 |      rectangle     0.4     36.0
 |
 |      >>> df.rdiv(10)
 |                   angles   degrees
 |      circle          inf  0.027778
 |      triangle   3.333333  0.055556
 |      rectangle  2.500000  0.027778
 |
 |      Subtract a list and Series by axis with operator version.
 |
 |      >>> df - [1, 2]
 |                 angles  degrees
 |      circle         -1      358
 |      triangle        2      178
 |      rectangle       3      358
 |
 |      >>> df.sub([1, 2], axis='columns')
 |                 angles  degrees
 |      circle         -1      358
 |      triangle        2      178
 |      rectangle       3      358
 |
 |      >>> df.sub(pd.Series([1, 1, 1], index=['circle', 'triangle', 'rectangle']),
 |      ...        axis='index')
 |                 angles  degrees
 |      circle         -1      359
 |      triangle        2      179
 |      rectangle       3      359
 |
 |      Multiply a DataFrame of different shape with operator version.
 |
 |      >>> other = pd.DataFrame({'angles': [0, 3, 4]},
 |      ...                      index=['circle', 'triangle', 'rectangle'])
 |      >>> other
 |                 angles
 |      circle          0
 |      triangle        3
 |      rectangle       4
 |
 |      >>> df * other
 |                 angles  degrees
 |      circle          0      NaN
 |      triangle        9      NaN
 |      rectangle      16      NaN
 |
 |      >>> df.mul(other, fill_value=0)
 |                 angles  degrees
 |      circle          0      0.0
 |      triangle        9      0.0
 |      rectangle      16      0.0
 |
 |      Divide by a MultiIndex by level.
```

```
 |
 |      >>> df_multindex = pd.DataFrame({'angles': [0, 3, 4, 4, 5, 6],
 |      ...                              'degrees': [360, 180, 360, 360, 540, 720]},
 |      ...                             index=[['A', 'A', 'A', 'B', 'B', 'B'],
 |      ...                                    ['circle', 'triangle', 'rectangle',
 |      ...                                     'square', 'pentagon', 'hexagon']])
 |      >>> df_multindex
 |                   angles  degrees
 |      A circle          0      360
 |        triangle        3      180
 |        rectangle       4      360
 |      B square          4      360
 |        pentagon        5      540
 |        hexagon         6      720
 |
 |      >>> df.div(df_multindex, level=1, fill_value=0)
 |                   angles  degrees
 |      A circle        NaN      1.0
 |        triangle      1.0      1.0
 |        rectangle     1.0      1.0
 |      B square        0.0      0.0
 |        pentagon      0.0      0.0
 |        hexagon       0.0      0.0
 |
 |  rsub(self, other, axis='columns', level=None, fill_value=None)
 |      Get Subtraction of dataframe and other, element-wise (binary operator
`rsub`).
 |
 |      Equivalent to ``other - dataframe``, but with support to substitute a
fill_value
 |      for missing data in one of the inputs. With reverse version, `sub`.
 |
 |      Among flexible wrappers (`add`, `sub`, `mul`, `div`, `mod`, `pow`) to
 |      arithmetic operators: `+`, `-`, `*`, `/`, `//`, `%`, `**`.
 |
 |      Parameters
 |      ----------
 |      other : scalar, sequence, Series, or DataFrame
 |          Any single or multiple element data structure, or list-like object.
 |      axis : {0 or 'index', 1 or 'columns'}
 |          Whether to compare by the index (0 or 'index') or columns
 |          (1 or 'columns'). For Series input, axis to match Series index on.
 |      level : int or label
 |          Broadcast across a level, matching Index values on the
 |          passed MultiIndex level.
 |      fill_value : float or None, default None
 |          Fill existing missing (NaN) values, and any new element needed for
 |          successful DataFrame alignment, with this value before computation.
 |          If data in both corresponding DataFrame locations is missing
 |          the result will be missing.
 |
 |      Returns
 |      -------
 |      DataFrame
 |          Result of the arithmetic operation.
 |
 |      See Also
 |      --------
 |      DataFrame.add : Add DataFrames.
 |      DataFrame.sub : Subtract DataFrames.
 |      DataFrame.mul : Multiply DataFrames.
 |      DataFrame.div : Divide DataFrames (float division).
 |      DataFrame.truediv : Divide DataFrames (float division).
 |      DataFrame.floordiv : Divide DataFrames (integer division).
 |      DataFrame.mod : Calculate modulo (remainder after division).
 |      DataFrame.pow : Calculate exponential power.
 |
 |      Notes
 |      -----
 |      Mismatched indices will be unioned together.
 |
 |      Examples
 |      --------
 |      >>> df = pd.DataFrame({'angles': [0, 3, 4],
 |      ...                    'degrees': [360, 180, 360]},
 |      ...                   index=['circle', 'triangle', 'rectangle'])
 |      >>> df
 |                 angles  degrees
 |      circle          0      360
 |      triangle        3      180
 |      rectangle       4      360
 |
 |      Add a scalar with operator version which return the same
 |      results.
 |
 |      >>> df + 1
 |                 angles  degrees
 |      circle          1      361
 |      triangle        4      181
 |      rectangle       5      361
 |
 |      >>> df.add(1)
 |                 angles  degrees
 |      circle          1      361
 |      triangle        4      181
 |      rectangle       5      361
 |
 |      Divide by constant with reverse version.
 |
 |      >>> df.div(10)
 |                 angles  degrees
 |      circle        0.0     36.0
 |      triangle      0.3     18.0
 |      rectangle     0.4     36.0
 |
 |      >>> df.rdiv(10)
 |                   angles   degrees
 |      circle          inf  0.027778
 |      triangle   3.333333  0.055556
 |      rectangle  2.500000  0.027778
 |
 |      Subtract a list and Series by axis with operator version.
 |
 |      >>> df - [1, 2]
 |                 angles  degrees
 |      circle         -1      358
 |      triangle        2      178
 |      rectangle       3      358
 |
 |      >>> df.sub([1, 2], axis='columns')
 |                 angles  degrees
 |      circle         -1      358
 |      triangle        2      178
 |      rectangle       3      358
 |
 |      >>> df.sub(pd.Series([1, 1, 1], index=['circle', 'triangle', 'rectangle']),
 |      ...        axis='index')
 |                 angles  degrees
 |      circle         -1      359
 |      triangle        2      179
 |      rectangle       3      359
 |
 |      Multiply a DataFrame of different shape with operator version.
 |
 |      >>> other = pd.DataFrame({'angles': [0, 3, 4]},
 |      ...                      index=['circle', 'triangle', 'rectangle'])
 |      >>> other
 |                 angles
 |      circle          0
 |      triangle        3
 |      rectangle       4
 |
 |      >>> df * other
 |                 angles  degrees
 |      circle          0      NaN
 |      triangle        9      NaN
 |      rectangle      16      NaN
 |
 |      >>> df.mul(other, fill_value=0)
 |                 angles  degrees
 |      circle          0      0.0
 |      triangle        9      0.0
 |      rectangle      16      0.0
 |
 |      Divide by a MultiIndex by level.
 |
 |      >>> df_multiindex = pd.DataFrame({'angles': [0, 3, 4, 4, 5, 6],
 |      ...                              'degrees': [360, 180, 360, 360, 540, 720]},
 |      ...                             index=[['A', 'A', 'A', 'B', 'B', 'B'],
 |      ...                                    ['circle', 'triangle', 'rectangle',
 |      ...                                     'square', 'pentagon', 'hexagon']])
 |      >>> df_multiindex
 |                   angles  degrees
 |      A circle          0      360
 |        triangle        3      180
 |        rectangle       4      360
 |      B square          4      360
 |        pentagon        5      540
 |        hexagon         6      720
 |
 |      >>> df.div(df_multiindex, level=1, fill_value=0)
 |                   angles  degrees
 |      A circle        NaN      1.0
 |        triangle      1.0      1.0
```

```
 |        rectangle     1.0      1.0
 |      B square        0.0      0.0
 |        pentagon      0.0      0.0
 |        hexagon       0.0      0.0
 |
 |  rtruediv(self, other, axis='columns', level=None, fill_value=None)
 |      Get Floating division of dataframe and other, element-wise (binary operator
`rtruediv`).
 |
 |      Equivalent to ``other / dataframe``, but with support to substitute a
fill_value
 |      for missing data in one of the inputs. With reverse version, `truediv`.
 |
 |      Among flexible wrappers (`add`, `sub`, `mul`, `div`, `mod`, `pow`) to
 |      arithmetic operators: `+`, `-`, `*`, `/`, `//`, `%`, `**`.
 |
 |      Parameters
 |      ----------
 |      other : scalar, sequence, Series, or DataFrame
 |          Any single or multiple element data structure, or list-like object.
 |      axis : {0 or 'index', 1 or 'columns'}
 |          Whether to compare by the index (0 or 'index') or columns
 |          (1 or 'columns'). For Series input, axis to match Series index on.
 |      level : int or label
 |          Broadcast across a level, matching Index values on the
 |          passed MultiIndex level.
 |      fill_value : float or None, default None
 |          Fill existing missing (NaN) values, and any new element needed for
 |          successful DataFrame alignment, with this value before computation.
 |          If data in both corresponding DataFrame locations is missing
 |          the result will be missing.
 |
 |      Returns
 |      -------
 |      DataFrame
 |          Result of the arithmetic operation.
 |
 |      See Also
 |      --------
 |      DataFrame.add : Add DataFrames.
 |      DataFrame.sub : Subtract DataFrames.
 |      DataFrame.mul : Multiply DataFrames.
 |      DataFrame.div : Divide DataFrames (float division).
 |      DataFrame.truediv : Divide DataFrames (float division).
 |      DataFrame.floordiv : Divide DataFrames (integer division).
 |      DataFrame.mod : Calculate modulo (remainder after division).
 |      DataFrame.pow : Calculate exponential power.
 |
 |      Notes
 |      -----
 |      Mismatched indices will be unioned together.
 |
 |      Examples
 |      --------
 |      >>> df = pd.DataFrame({'angles': [0, 3, 4],
 |      ...                    'degrees': [360, 180, 360]},
 |      ...                   index=['circle', 'triangle', 'rectangle'])
 |      >>> df
 |                 angles  degrees
 |      circle          0      360
 |      triangle        3      180
 |      rectangle       4      360
 |
 |      Add a scalar with operator version which return the same
 |      results.
 |
 |      >>> df + 1
 |                 angles  degrees
 |      circle          1      361
 |      triangle        4      181
 |      rectangle       5      361
 |
 |      >>> df.add(1)
 |                 angles  degrees
 |      circle          1      361
 |      triangle        4      181
 |      rectangle       5      361
 |
 |      Divide by constant with reverse version.
 |
 |      >>> df.div(10)
 |                 angles  degrees
 |      circle        0.0     36.0
 |      triangle      0.3     18.0
 |      rectangle     0.4     36.0
 |
 |      >>> df.rdiv(10)
 |                   angles   degrees
 |      circle          inf  0.027778
 |      triangle   3.333333  0.055556
 |      rectangle  2.500000  0.027778
 |
 |      Subtract a list and Series by axis with operator version.
 |
 |      >>> df - [1, 2]
 |                 angles  degrees
 |      circle         -1      358
 |      triangle        2      178
 |      rectangle       3      358
 |
 |      >>> df.sub([1, 2], axis='columns')
 |                 angles  degrees
 |      circle         -1      358
 |      triangle        2      178
 |      rectangle       3      358
 |
 |      >>> df.sub(pd.Series([1, 1, 1], index=['circle', 'triangle', 'rectangle']),
 |      ...        axis='index')
 |                 angles  degrees
 |      circle         -1      359
 |      triangle        2      179
 |      rectangle       3      359
 |
 |      Multiply a DataFrame of different shape with operator version.
 |
 |      >>> other = pd.DataFrame({'angles': [0, 3, 4]},
 |      ...                      index=['circle', 'triangle', 'rectangle'])
 |      >>> other
 |                 angles
 |      circle          0
 |      triangle        3
 |      rectangle       4
 |
 |      >>> df * other
 |                 angles  degrees
 |      circle          0      NaN
 |      triangle        9      NaN
 |      rectangle      16      NaN
 |
 |      >>> df.mul(other, fill_value=0)
 |                 angles  degrees
 |      circle          0      0.0
 |      triangle        9      0.0
 |      rectangle      16      0.0
 |
 |      Divide by a MultiIndex by level.
 |
 |      >>> df_multindex = pd.DataFrame({'angles': [0, 3, 4, 4, 5, 6],
 |      ...                              'degrees': [360, 180, 360, 360, 540, 720]},
 |      ...                             index=[['A', 'A', 'A', 'B', 'B', 'B'],
 |      ...                                    ['circle', 'triangle', 'rectangle',
 |      ...                                     'square', 'pentagon', 'hexagon']])
 |      >>> df_multindex
 |                   angles  degrees
 |      A circle          0      360
 |        triangle        3      180
 |        rectangle       4      360
 |      B square          4      360
 |        pentagon        5      540
 |        hexagon         6      720
 |
 |      >>> df.div(df_multindex, level=1, fill_value=0)
 |                   angles  degrees
 |      A circle        NaN      1.0
 |        triangle      1.0      1.0
 |        rectangle     1.0      1.0
 |      B square        0.0      0.0
 |        pentagon      0.0      0.0
 |        hexagon       0.0      0.0
 |
 |  select_dtypes(self, include=None, exclude=None) -> 'DataFrame'
 |      Return a subset of the DataFrame's columns based on the column dtypes.
 |
 |      Parameters
 |      ----------
 |      include, exclude : scalar or list-like
 |          A selection of dtypes or strings to be included/excluded. At least
 |          one of these parameters must be supplied.
 |
 |      Returns
 |      -------
 |      DataFrame
 |          The subset of the frame including the dtypes in ``include`` and
 |          excluding the dtypes in ``exclude``.
```

```
 |
 |      Raises
 |      ------
 |      ValueError
 |          * If both of ``include`` and ``exclude`` are empty
 |          * If ``include`` and ``exclude`` have overlapping elements
 |          * If any kind of string dtype is passed in.
 |
 |      See Also
 |      --------
 |      DataFrame.dtypes: Return Series with the data type of each column.
 |
 |      Notes
 |      -----
 |      * To select all *numeric* types, use ``np.number`` or ``'number'``
 |      * To select strings you must use the ``object`` dtype, but note that
 |        this will return *all* object dtype columns
 |      * See the `numpy dtype hierarchy
 |        <https://numpy.org/doc/stable/reference/arrays.scalars.html>`__
 |      * To select datetimes, use ``np.datetime64``, ``'datetime'`` or
 |        ``'datetime64'``
 |      * To select timedeltas, use ``np.timedelta64``, ``'timedelta'`` or
 |        ``'timedelta64'``
 |      * To select Pandas categorical dtypes, use ``'category'``
 |      * To select Pandas datetimetz dtypes, use ``'datetimetz'`` (new in
 |        0.20.0) or ``'datetime64[ns, tz]'``
 |
 |      Examples
 |      --------
 |      >>> df = pd.DataFrame({'a': [1, 2] * 3,
 |      ...                    'b': [True, False] * 3,
 |      ...                    'c': [1.0, 2.0] * 3})
 |      >>> df
 |              a      b  c
 |      0       1   True  1.0
 |      1       2  False  2.0
 |      2       1   True  1.0
 |      3       2  False  2.0
 |      4       1   True  1.0
 |      5       2  False  2.0
 |
 |      >>> df.select_dtypes(include='bool')
 |         b
 |      0  True
 |      1  False
 |      2  True
 |      3  False
 |      4  True
 |      5  False
 |
 |      >>> df.select_dtypes(include=['float64'])
 |         c
 |      0  1.0
 |      1  2.0
 |      2  1.0
 |      3  2.0
 |      4  1.0
 |      5  2.0
 |
 |      >>> df.select_dtypes(exclude=['int64'])
 |             b    c
 |      0   True  1.0
 |      1  False  2.0
 |      2   True  1.0
 |      3  False  2.0
 |      4   True  1.0
 |      5  False  2.0
 |
 |  sem(self, axis=None, skipna=None, level=None, ddof=1, numeric_only=None,
**kwargs)
 |      Return unbiased standard error of the mean over requested axis.
 |
 |      Normalized by N-1 by default. This can be changed using the ddof argument
 |
 |      Parameters
 |      ----------
 |      axis : {index (0), columns (1)}
 |      skipna : bool, default True
 |          Exclude NA/null values. If an entire row/column is NA, the result
 |          will be NA.
 |      level : int or level name, default None
 |          If the axis is a MultiIndex (hierarchical), count along a
 |          particular level, collapsing into a Series.
 |      ddof : int, default 1
 |          Delta Degrees of Freedom. The divisor used in calculations is N - ddof,
 |          where N represents the number of elements.
 |      numeric_only : bool, default None
 |          Include only float, int, boolean columns. If None, will attempt to use
 |          everything, then use only numeric data. Not implemented for Series.
 |      
 |      Returns
 |      -------
 |      Series or DataFrame (if level specified)
 |      
 |      Notes
 |      -----
 |      To have the same behaviour as `numpy.std`, use `ddof=0` (instead of the
 |      default `ddof=1`)
 |  
 |  set_axis(self, labels, axis: 'Axis' = 0, inplace: 'bool' = False)
 |      Assign desired index to given axis.
 |      
 |      Indexes for column or row labels can be changed by assigning
 |      a list-like or Index.
 |      
 |      Parameters
 |      ----------
 |      labels : list-like, Index
 |          The values for the new index.
 |      
 |      axis : {0 or 'index', 1 or 'columns'}, default 0
 |          The axis to update. The value 0 identifies the rows, and 1 identifies the
columns.
 |      
 |      inplace : bool, default False
 |          Whether to return a new DataFrame instance.
 |      
 |      Returns
 |      -------
 |      renamed : DataFrame or None
 |          An object of type DataFrame or None if ``inplace=True``.
 |      
 |      See Also
 |      --------
 |      DataFrame.rename_axis : Alter the name of the index or columns.
 |      
 |              Examples
 |              --------
 |              >>> df = pd.DataFrame({"A": [1, 2, 3], "B": [4, 5, 6]})
 |      
 |              Change the row labels.
 |      
 |              >>> df.set_axis(['a', 'b', 'c'], axis='index')
 |                 A  B
 |              a  1  4
 |              b  2  5
 |              c  3  6
 |      
 |              Change the column labels.
 |      
 |              >>> df.set_axis(['I', 'II'], axis='columns')
 |                 I  II
 |              0  1   4
 |              1  2   5
 |              2  3   6
 |      
 |              Now, update the labels inplace.
 |      
 |              >>> df.set_axis(['i', 'ii'], axis='columns', inplace=True)
 |              >>> df
 |                 i  ii
 |              0  1   4
 |              1  2   5
 |              2  3   6
 |  
 |  set_index(self, keys, drop: 'bool' = True, append: 'bool' = False, inplace:
'bool' = False, verify_integrity: 'bool' = False)
 |      Set the DataFrame index using existing columns.
 |      
 |      Set the DataFrame index (row labels) using one or more existing
 |      columns or arrays (of the correct length). The index can replace the
 |      existing index or expand on it.
 |      
 |      Parameters
 |      ----------
 |      keys : label or array-like or list of labels/arrays
 |          This parameter can be either a single column key, a single array of
 |          the same length as the calling DataFrame, or a list containing an
 |          arbitrary combination of column keys and arrays. Here, "array"
 |          encompasses :class:`Series`, :class:`Index`, ``np.ndarray``, and
 |          instances of :class:`~collections.abc.Iterator`.
 |      drop : bool, default True
 |          Delete columns to be used as the new index.
 |      append : bool, default False
 |          Whether to append columns to existing index.
 |      inplace : bool, default False
 |          If True, modifies the DataFrame in place (do not create a new object).
 |      verify_integrity : bool, default False
 |          Check the new index for duplicates. Otherwise defer the check until
 |          necessary. Setting to False will improve the performance of this
 |          method.
 |      
 |      Returns
 |      -------
 |      DataFrame or None
 |          Changed row labels or None if ``inplace=True``.
 |      
 |      See Also
 |      --------
 |      DataFrame.reset_index : Opposite of set_index.
 |      DataFrame.reindex : Change to new indices or expand indices.
 |      DataFrame.reindex_like : Change to same indices as other DataFrame.
 |      
 |      Examples
 |      --------
 |      >>> df = pd.DataFrame({'month': [1, 4, 7, 10],
 |      ...                    'year': [2012, 2014, 2013, 2014],
 |      ...                    'sale': [55, 40, 84, 31]})
 |      >>> df
 |         month  year  sale
 |      0      1  2012    55
 |      1      4  2014    40
 |      2      7  2013    84
 |      3     10  2014    31
 |      
 |      Set the index to become the 'month' column:
 |      
 |      >>> df.set_index('month')
 |             year  sale
 |      month
 |      1      2012    55
 |      4      2014    40
 |      7      2013    84
 |      10     2014    31
 |      
 |      Create a MultiIndex using columns 'year' and 'month':
 |      
 |      >>> df.set_index(['year', 'month'])
 |                  sale
 |      year  month
 |      2012  1     55
 |      2014  4     40
 |      2013  7     84
 |      2014  10    31
 |      
 |      Create a MultiIndex using an Index and a column:
 |      
 |      >>> df.set_index([pd.Index([1, 2, 3, 4]), 'year'])
 |               month  sale
 |         year
 |      1  2012  1      55
 |      2  2014  4      40
 |      3  2013  7      84
 |      4  2014  10     31
 |      
 |      Create a MultiIndex using two Series:
 |      
 |      >>> s = pd.Series([1, 2, 3, 4])
 |      >>> df.set_index([s, s**2])
 |            month  year  sale
 |      1 1       1  2012    55
 |      2 4       4  2014    40
 |      3 9       7  2013    84
 |      4 16     10  2014    31
 |  
 |  shift(self, periods=1, freq: 'Frequency | None' = None, axis: 'Axis' = 0,
fill_value=<no_default>) -> 'DataFrame'
 |      Shift index by desired number of periods with an optional time `freq`.
 |      
 |      When `freq` is not passed, shift the index without realigning the data.
 |      If `freq` is passed (in this case, the index must be date or datetime,
 |      or it will raise a `NotImplementedError`), the index will be
 |      increased using the periods and the `freq`. `freq` can be inferred
 |      when specified as "infer" as long as either freq or inferred_freq
 |      attribute is set in the index.
 |      
 |      Parameters
 |      ----------
 |      periods : int
 |          Number of periods to shift. Can be positive or negative.
 |      freq : DateOffset, tseries.offsets, timedelta, or str, optional
 |          Offset to use from the tseries module or time rule (e.g. 'EOM').
 |          If `freq` is specified then the index values are shifted but the
 |          data is not realigned. That is, use `freq` if you would like to
 |          extend the index when shifting and preserve the original data.
 |          If `freq` is specified as "infer" then it will be inferred from
 |          the freq or inferred_freq attributes of the index. If neither of
 |          those attributes exist, a ValueError is thrown.
 |      axis : {0 or 'index', 1 or 'columns', None}, default None
 |          Shift direction.
 |      fill_value : object, optional
 |          The scalar value to use for newly introduced missing values.
 |          the default depends on the dtype of `self`.
 |          For numeric data, ``np.nan`` is used.
 |          For datetime, timedelta, or period data, etc. :attr:`NaT` is used.
 |          For extension dtypes, ``self.dtype.na_value`` is used.
 |
 |          .. versionchanged:: 1.1.0
 |
 |      Returns
 |      -------
 |      DataFrame
 |          Copy of input object, shifted.
 |
 |      See Also
 |      --------
 |      Index.shift : Shift values of Index.
 |      DatetimeIndex.shift : Shift values of DatetimeIndex.
 |      PeriodIndex.shift : Shift values of PeriodIndex.
 |      tshift : Shift the time index, using the index's frequency if
 |          available.
 |
 |      Examples
 |      --------
 |      >>> df = pd.DataFrame({"Col1": [10, 20, 15, 30, 45],
 |      ...                    "Col2": [13, 23, 18, 33, 48],
 |      ...                    "Col3": [17, 27, 22, 37, 52]},
 |      ...                   index=pd.date_range("2020-01-01", "2020-01-05"))
 |      >>> df
 |                  Col1  Col2  Col3
 |      2020-01-01    10    13    17
 |      2020-01-02    20    23    27
 |      2020-01-03    15    18    22
 |      2020-01-04    30    33    37
 |      2020-01-05    45    48    52
 |
 |      >>> df.shift(periods=3)
 |                  Col1  Col2  Col3
 |      2020-01-01   NaN   NaN   NaN
 |      2020-01-02   NaN   NaN   NaN
 |      2020-01-03   NaN   NaN   NaN
 |      2020-01-04  10.0  13.0  17.0
 |      2020-01-05  20.0  23.0  27.0
 |
 |      >>> df.shift(periods=1, axis="columns")
 |                  Col1  Col2  Col3
 |      2020-01-01   NaN    10    13
 |      2020-01-02   NaN    20    23
 |      2020-01-03   NaN    15    18
 |      2020-01-04   NaN    30    33
 |      2020-01-05   NaN    45    48
 |
 |      >>> df.shift(periods=3, fill_value=0)
 |                  Col1  Col2  Col3
 |      2020-01-01     0     0     0
 |      2020-01-02     0     0     0
 |      2020-01-03     0     0     0
 |      2020-01-04    10    13    17
 |      2020-01-05    20    23    27
 |
 |      >>> df.shift(periods=3, freq="D")
 |                  Col1  Col2  Col3
 |      2020-01-04    10    13    17
 |      2020-01-05    20    23    27
 |      2020-01-06    15    18    22
 |      2020-01-07    30    33    37
 |      2020-01-08    45    48    52
 |
 |      >>> df.shift(periods=3, freq="infer")
 |                  Col1  Col2  Col3
 |      2020-01-04    10    13    17
 |      2020-01-05    20    23    27
 |      2020-01-06    15    18    22
 |      2020-01-07    30    33    37
 |      2020-01-08    45    48    52
 |
 |  skew(self, axis=None, skipna=None, level=None, numeric_only=None, **kwargs)
 |      Return unbiased skew over requested axis.
 |      
 |      Normalized by N-1.
 |      
 |      Parameters
 |      ----------
 |      axis : {index (0), columns (1)}
 |          Axis for the function to be applied on.
 |      skipna : bool, default True
 |          Exclude NA/null values when computing the result.
 |      level : int or level name, default None
 |          If the axis is a MultiIndex (hierarchical), count along a
 |          particular level, collapsing into a Series.
 |      numeric_only : bool, default None
 |          Include only float, int, boolean columns. If None, will attempt to use
 |          everything, then use only numeric data. Not implemented for Series.
 |      **kwargs
 |          Additional keyword arguments to be passed to the function.
 |      
 |      Returns
 |      -------
 |      Series or DataFrame (if level specified)
 |  
 |  sort_index(self, axis: 'Axis' = 0, level: 'Level | None' = None, ascending: 'bool
| int | Sequence[bool | int]' = True, inplace: 'bool' = False, kind: 'str' =
'quicksort', na_position: 'str' = 'last', sort_remaining: 'bool' = True,
ignore_index: 'bool' = False, key: 'IndexKeyFunc' = None)
 |      Sort object by labels (along an axis).
 |      
 |      Returns a new DataFrame sorted by label if `inplace` argument is
 |      ``False``, otherwise updates the original DataFrame and returns None.
 |      
 |      Parameters
 |      ----------
 |      axis : {0 or 'index', 1 or 'columns'}, default 0
 |          The axis along which to sort.  The value 0 identifies the rows,
 |          and 1 identifies the columns.
 |      level : int or level name or list of ints or list of level names
 |          If not None, sort on values in specified index level(s).
 |      ascending : bool or list-like of bools, default True
 |          Sort ascending vs. descending. When the index is a MultiIndex the
 |          sort direction can be controlled for each level individually.
 |      inplace : bool, default False
 |          If True, perform operation in-place.
 |      kind : {'quicksort', 'mergesort', 'heapsort', 'stable'}, default 'quicksort'
 |          Choice of sorting algorithm. See also :func:`numpy.sort` for more
 |          information. `mergesort` and `stable` are the only stable algorithms. For
 |          DataFrames, this option is only applied when sorting on a single
 |          column or label.
 |      na_position : {'first', 'last'}, default 'last'
 |          Puts NaNs at the beginning if `first`; `last` puts NaNs at the end.
 |          Not implemented for MultiIndex.
 |      sort_remaining : bool, default True
 |          If True and sorting by level and index is multilevel, sort by other
 |          levels too (in order) after sorting by specified level.
 |      ignore_index : bool, default False
 |          If True, the resulting axis will be labeled 0, 1, …, n - 1.
 |      
 |          .. versionadded:: 1.0.0
 |      
 |      key : callable, optional
 |          If not None, apply the key function to the index values
 |          before sorting. This is similar to the `key` argument in the
 |          builtin :meth:`sorted` function, with the notable difference that
 |          this `key` function should be *vectorized*. It should expect an
 |          ``Index`` and return an ``Index`` of the same shape. For MultiIndex
 |          inputs, the key is applied *per level*.
 |      
 |          .. versionadded:: 1.1.0
 |      
 |      Returns
 |      -------
 |      DataFrame or None
 |          The original DataFrame sorted by the labels or None if ``inplace=True``.
 |      
 |      See Also
 |      --------
 |      Series.sort_index : Sort Series by the index.
 |      DataFrame.sort_values : Sort DataFrame by the value.
 |      Series.sort_values : Sort Series by the value.
 |      
 |      Examples
 |      --------
 |      >>> df = pd.DataFrame([1, 2, 3, 4, 5], index=[100, 29, 234, 1, 150],
 |      ...                   columns=['A'])
 |      >>> df.sort_index()
 |           A
 |      1    4
 |      29   2
 |      100  1
 |      150  5
 |      234  3
 |
 |      By default, it sorts in ascending order, to sort in descending order,
 |      use ``ascending=False``
 |
 |      >>> df.sort_index(ascending=False)
 |           A
 |      234  3
 |      150  5
 |      100  1
 |      29   2
 |      1    4
 |
 |      A key function can be specified which is applied to the index before
 |      sorting. For a ``MultiIndex`` this is applied to each level separately.
 |
 |      >>> df = pd.DataFrame({"a": [1, 2, 3, 4]}, index=['A', 'b', 'C', 'd'])
 |      >>> df.sort_index(key=lambda x: x.str.lower())
 |         a
 |      A  1
 |      b  2
 |      C  3
 |      d  4
 |
 |  sort_values(self, by, axis: 'Axis' = 0, ascending=True, inplace: 'bool' = False,
kind: 'str' = 'quicksort', na_position: 'str' = 'last', ignore_index: 'bool' = False,
key: 'ValueKeyFunc' = None)
 |      Sort by the values along either axis.
 |
 |      Parameters
 |      ----------
 |              by : str or list of str
 |                  Name or list of names to sort by.
 |
 |                  - if `axis` is 0 or `'index'` then `by` may contain index
 |                    levels and/or column labels.
 |                  - if `axis` is 1 or `'columns'` then `by` may contain column
 |                    levels and/or index labels.
 |      axis : {0 or 'index', 1 or 'columns'}, default 0
 |           Axis to be sorted.
 |      ascending : bool or list of bool, default True
 |           Sort ascending vs. descending. Specify list for multiple sort
 |           orders.  If this is a list of bools, must match the length of
 |           the by.
 |      inplace : bool, default False
 |           If True, perform operation in-place.
 |      kind : {'quicksort', 'mergesort', 'heapsort', 'stable'}, default 'quicksort'
 |           Choice of sorting algorithm. See also :func:`numpy.sort` for more
 |           information. `mergesort` and `stable` are the only stable algorithms.
For
 |           DataFrames, this option is only applied when sorting on a single
 |           column or label.
 |      na_position : {'first', 'last'}, default 'last'
 |           Puts NaNs at the beginning if `first`; `last` puts NaNs at the
 |           end.
 |      ignore_index : bool, default False
 |           If True, the resulting axis will be labeled 0, 1, …, n - 1.
 |
 |           .. versionadded:: 1.0.0
 |
 |      key : callable, optional
 |          Apply the key function to the values
 |          before sorting. This is similar to the `key` argument in the
 |          builtin :meth:`sorted` function, with the notable difference that
 |          this `key` function should be *vectorized*. It should expect a
 |          ``Series`` and return a Series with the same shape as the input.
 |          It will be applied to each column in `by` independently.
 |
 |          .. versionadded:: 1.1.0
 |
 |      Returns
 |      -------
 |      DataFrame or None
 |          DataFrame with sorted values or None if ``inplace=True``.
 |
 |      See Also
 |      --------
 |      DataFrame.sort_index : Sort a DataFrame by the index.
 |      Series.sort_values : Similar method for a Series.
 |
 |      Examples
 |      --------
 |      >>> df = pd.DataFrame({
 |      ...     'col1': ['A', 'A', 'B', np.nan, 'D', 'C'],
 |      ...     'col2': [2, 1, 9, 8, 7, 4],
 |      ...     'col3': [0, 1, 9, 4, 2, 3],
 |      ...     'col4': ['a', 'B', 'c', 'D', 'e', 'F']
 |      ... })
 |      >>> df
 |        col1  col2  col3 col4
 |      0    A     2     0    a
 |      1    A     1     1    B
 |      2    B     9     9    c
 |      3  NaN     8     4    D
 |      4    D     7     2    e
 |      5    C     4     3    F
 |
 |      Sort by col1
 |
 |      >>> df.sort_values(by=['col1'])
 |        col1  col2  col3 col4
 |      0    A     2     0    a
 |      1    A     1     1    B
 |      2    B     9     9    c
 |      5    C     4     3    F
 |      4    D     7     2    e
 |      3  NaN     8     4    D
 |
 |      Sort by multiple columns
 |
 |      >>> df.sort_values(by=['col1', 'col2'])
 |        col1  col2  col3 col4
 |      1    A     1     1    B
 |      0    A     2     0    a
 |      2    B     9     9    c
 |      5    C     4     3    F
 |      4    D     7     2    e
 |      3  NaN     8     4    D
 |
 |      Sort Descending
 |
 |      >>> df.sort_values(by='col1', ascending=False)
 |        col1  col2  col3 col4
 |      4    D     7     2    e
 |      5    C     4     3    F
 |      2    B     9     9    c
 |      0    A     2     0    a
 |      1    A     1     1    B
 |      3  NaN     8     4    D
 |
 |      Putting NAs first
 |
 |      >>> df.sort_values(by='col1', ascending=False, na_position='first')
 |        col1  col2  col3 col4
 |      3  NaN     8     4    D
 |      4    D     7     2    e
 |      5    C     4     3    F
 |      2    B     9     9    c
 |      0    A     2     0    a
 |      1    A     1     1    B
 |
 |      Sorting with a key function
 |
 |      >>> df.sort_values(by='col4', key=lambda col: col.str.lower())
 |         col1  col2  col3 col4
 |      0    A     2     0    a
 |      1    A     1     1    B
 |      2    B     9     9    c
 |      3  NaN     8     4    D
 |      4    D     7     2    e
 |      5    C     4     3    F
 |
 |      Natural sort with the key argument,
 |      using the `natsort <https://github.com/SethMMorton/natsort>` package.
 |
 |      >>> df = pd.DataFrame({
 |      ...    "time": ['0hr', '128hr', '72hr', '48hr', '96hr'],
 |      ...    "value": [10, 20, 30, 40, 50]
 |      ... })
 |      >>> df
 |          time  value
 |      0    0hr     10
 |      1  128hr     20
 |      2   72hr     30
 |      3   48hr     40
 |      4   96hr     50
 |      >>> from natsort import index_natsorted
 |      >>> df.sort_values(
 |      ...    by="time",
 |      ...    key=lambda x: np.argsort(index_natsorted(df["time"]))
 |      ... )
 |          time  value
 |      0    0hr     10
 |      3   48hr     40
 |      2   72hr     30
 |      4   96hr     50
 |      1  128hr     20
 |  
 |  stack(self, level: 'Level' = -1, dropna: 'bool' = True)
 |      Stack the prescribed level(s) from columns to index.
 |      
 |      Return a reshaped DataFrame or Series having a multi-level
 |      index with one or more new inner-most levels compared to the current
 |      DataFrame. The new inner-most levels are created by pivoting the
 |      columns of the current dataframe:
 |      
 |        - if the columns have a single level, the output is a Series;
 |        - if the columns have multiple levels, the new index
 |          level(s) is (are) taken from the prescribed level(s) and
 |          the output is a DataFrame.
 |      
 |      Parameters
 |      ----------
 |      level : int, str, list, default -1
 |          Level(s) to stack from the column axis onto the index
 |          axis, defined as one index or label, or a list of indices
 |          or labels.
 |      dropna : bool, default True
 |          Whether to drop rows in the resulting Frame/Series with
 |          missing values. Stacking a column level onto the index
 |          axis can create combinations of index and column values
 |          that are missing from the original dataframe. See Examples
 |          section.
 |      
 |      Returns
 |      -------
 |      DataFrame or Series
 |          Stacked dataframe or series.
 |      
 |      See Also
 |      --------
 |      DataFrame.unstack : Unstack prescribed level(s) from index axis
 |           onto column axis.
 |      DataFrame.pivot : Reshape dataframe from long format to wide
 |           format.
 |      DataFrame.pivot_table : Create a spreadsheet-style pivot table
 |           as a DataFrame.
 |      
 |      Notes
 |      -----
 |      The function is named by analogy with a collection of books
 |      being reorganized from being side by side on a horizontal
 |      position (the columns of the dataframe) to being stacked
 |      vertically on top of each other (in the index of the
 |      dataframe).
 |      
 |      Examples
 |      --------
 |      **Single level columns**
 |      
 |      >>> df_single_level_cols = pd.DataFrame([[0, 1], [2, 3]],
 |      ...                                     index=['cat', 'dog'],
 |      ...                                     columns=['weight', 'height'])
 |      
 |      Stacking a dataframe with a single level column axis returns a Series:
 |      
 |      >>> df_single_level_cols
 |           weight height
 |      cat       0      1
 |      dog       2      3
 |      >>> df_single_level_cols.stack()
 |      cat  weight    0
 |           height    1
 |      dog  weight    2
 |           height    3
 |      dtype: int64
 |      
 |      **Multi level columns: simple case**
 |      
 |      >>> multicol1 = pd.MultiIndex.from_tuples([('weight', 'kg'),
 |      ...                                        ('weight', 'pounds')])
 |      >>> df_multi_level_cols1 = pd.DataFrame([[1, 2], [2, 4]],
 |      ...                                     index=['cat', 'dog'],
 |      ...                                     columns=multicol1)
 |      
 |      Stacking a dataframe with a multi-level column axis:
 |      
 |      >>> df_multi_level_cols1
```

```
 |           weight
 |               kg    pounds
 |      cat       1        2
 |      dog       2        4
 |      >>> df_multi_level_cols1.stack()
 |                  weight
 |      cat kg           1
 |          pounds       2
 |      dog kg           2
 |          pounds       4
 |      
 |      **Missing values**
 |      
 |      >>> multicol2 = pd.MultiIndex.from_tuples([('weight', 'kg'),
 |      ...                                        ('height', 'm')])
 |      >>> df_multi_level_cols2 = pd.DataFrame([[1.0, 2.0], [3.0, 4.0]],
 |      ...                                     index=['cat', 'dog'],
 |      ...                                     columns=multicol2)
 |      
 |      It is common to have missing values when stacking a dataframe
 |      with multi-level columns, as the stacked dataframe typically
 |      has more values than the original dataframe. Missing values
 |      are filled with NaNs:
 |      
 |      >>> df_multi_level_cols2
 |          weight height
 |              kg      m
 |      cat    1.0    2.0
 |      dog    3.0    4.0
 |      >>> df_multi_level_cols2.stack()
 |              height  weight
 |      cat kg     NaN     1.0
 |          m      2.0     NaN
 |      dog kg     NaN     3.0
 |          m      4.0     NaN
 |      
 |      **Prescribing the level(s) to be stacked**
 |      
 |      The first parameter controls which level or levels are stacked:
 |      
 |      >>> df_multi_level_cols2.stack(0)
 |                   kg    m
 |      cat height  NaN  2.0
 |          weight  1.0  NaN
 |      dog height  NaN  4.0
 |          weight  3.0  NaN
 |      >>> df_multi_level_cols2.stack([0, 1])
 |      cat  height  m     2.0
 |           weight  kg    1.0
 |      dog  height  m     4.0
 |           weight  kg    3.0
 |      dtype: float64
 |      
 |      **Dropping missing values**
 |      
 |      >>> df_multi_level_cols3 = pd.DataFrame([[None, 1.0], [2.0, 3.0]],
 |      ...                                     index=['cat', 'dog'],
 |      ...                                     columns=multicol2)
 |      
 |      Note that rows where all values are missing are dropped by
 |      default but this behaviour can be controlled via the dropna
 |      keyword parameter:
 |      
 |      >>> df_multi_level_cols3
 |          weight height
 |              kg      m
 |      cat    NaN    1.0
 |      dog    2.0    3.0
 |      >>> df_multi_level_cols3.stack(dropna=False)
 |              height  weight
 |      cat kg     NaN     NaN
 |          m      1.0     NaN
 |      dog kg     NaN     2.0
 |          m      3.0     NaN
 |      >>> df_multi_level_cols3.stack(dropna=True)
 |              height  weight
 |      cat m      1.0     NaN
 |      dog kg     NaN     2.0
 |          m      3.0     NaN
 |  
 |  std(self, axis=None, skipna=None, level=None, ddof=1, numeric_only=None, 
**kwargs)
 |      Return sample standard deviation over requested axis.
 |      
 |      Normalized by N-1 by default. This can be changed using the ddof argument
 |      
 |      Parameters
 |      ----------
 |      axis : {index (0), columns (1)}
 |      skipna : bool, default True
 |          Exclude NA/null values. If an entire row/column is NA, the result
 |          will be NA.
 |      level : int or level name, default None
 |          If the axis is a MultiIndex (hierarchical), count along a
 |          particular level, collapsing into a Series.
 |      ddof : int, default 1
 |          Delta Degrees of Freedom. The divisor used in calculations is N - ddof,
 |          where N represents the number of elements.
 |      numeric_only : bool, default None
 |          Include only float, int, boolean columns. If None, will attempt to use
 |          everything, then use only numeric data. Not implemented for Series.
 |
 |      Returns
 |      -------
 |      Series or DataFrame (if level specified)
 |
 |      Notes
 |      -----
 |      To have the same behaviour as `numpy.std`, use `ddof=0` (instead of the
 |      default `ddof=1`)
 |
 |  sub(self, other, axis='columns', level=None, fill_value=None)
 |      Get Subtraction of dataframe and other, element-wise (binary operator `sub`).
 |
 |      Equivalent to ``dataframe - other``, but with support to substitute a
fill_value
 |      for missing data in one of the inputs. With reverse version, `rsub`.
 |
 |      Among flexible wrappers (`add`, `sub`, `mul`, `div`, `mod`, `pow`) to
 |      arithmetic operators: `+`, `-`, `*`, `/`, `//`, `%`, `**`.
 |
 |      Parameters
 |      ----------
 |      other : scalar, sequence, Series, or DataFrame
 |          Any single or multiple element data structure, or list-like object.
 |      axis : {0 or 'index', 1 or 'columns'}
 |          Whether to compare by the index (0 or 'index') or columns
 |          (1 or 'columns'). For Series input, axis to match Series index on.
 |      level : int or label
 |          Broadcast across a level, matching Index values on the
 |          passed MultiIndex level.
 |      fill_value : float or None, default None
 |          Fill existing missing (NaN) values, and any new element needed for
 |          successful DataFrame alignment, with this value before computation.
 |          If data in both corresponding DataFrame locations is missing
 |          the result will be missing.
 |
 |      Returns
 |      -------
 |      DataFrame
 |          Result of the arithmetic operation.
 |
 |      See Also
 |      --------
 |      DataFrame.add : Add DataFrames.
 |      DataFrame.sub : Subtract DataFrames.
 |      DataFrame.mul : Multiply DataFrames.
 |      DataFrame.div : Divide DataFrames (float division).
 |      DataFrame.truediv : Divide DataFrames (float division).
 |      DataFrame.floordiv : Divide DataFrames (integer division).
 |      DataFrame.mod : Calculate modulo (remainder after division).
 |      DataFrame.pow : Calculate exponential power.
 |
 |      Notes
 |      -----
 |      Mismatched indices will be unioned together.
 |
 |      Examples
 |      --------
 |      >>> df = pd.DataFrame({'angles': [0, 3, 4],
 |      ...                    'degrees': [360, 180, 360]},
 |      ...                   index=['circle', 'triangle', 'rectangle'])
 |      >>> df
 |                 angles  degrees
 |      circle          0      360
 |      triangle        3      180
 |      rectangle       4      360
 |
 |      Add a scalar with operator version which return the same
 |      results.
 |
 |      >>> df + 1
 |                 angles  degrees
 |      circle          1      361
 |      triangle        4      181
 |      rectangle       5      361
 |
 |      >>> df.add(1)
 |                 angles  degrees
 |      circle          1      361
 |      triangle        4      181
 |      rectangle       5      361
 |
 |      Divide by constant with reverse version.
 |
 |      >>> df.div(10)
 |                 angles  degrees
 |      circle        0.0     36.0
 |      triangle      0.3     18.0
 |      rectangle     0.4     36.0
 |
 |      >>> df.rdiv(10)
 |                   angles   degrees
 |      circle          inf  0.027778
 |      triangle   3.333333  0.055556
 |      rectangle  2.500000  0.027778
 |
 |      Subtract a list and Series by axis with operator version.
 |
 |      >>> df - [1, 2]
 |                 angles  degrees
 |      circle         -1      358
 |      triangle        2      178
 |      rectangle       3      358
 |
 |      >>> df.sub([1, 2], axis='columns')
 |                 angles  degrees
 |      circle         -1      358
 |      triangle        2      178
 |      rectangle       3      358
 |
 |      >>> df.sub(pd.Series([1, 1, 1], index=['circle', 'triangle', 'rectangle']),
 |      ...        axis='index')
 |                 angles  degrees
 |      circle         -1      359
 |      triangle        2      179
 |      rectangle       3      359
 |
 |      Multiply a DataFrame of different shape with operator version.
 |
 |      >>> other = pd.DataFrame({'angles': [0, 3, 4]},
 |      ...                      index=['circle', 'triangle', 'rectangle'])
 |      >>> other
 |                 angles
 |      circle          0
 |      triangle        3
 |      rectangle       4
 |
 |      >>> df * other
 |                 angles  degrees
 |      circle          0      NaN
 |      triangle        9      NaN
 |      rectangle      16      NaN
 |
 |      >>> df.mul(other, fill_value=0)
 |                 angles  degrees
 |      circle          0      0.0
 |      triangle        9      0.0
 |      rectangle      16      0.0
 |
 |      Divide by a MultiIndex by level.
 |
 |      >>> df_multindex = pd.DataFrame({'angles': [0, 3, 4, 4, 5, 6],
 |      ...                              'degrees': [360, 180, 360, 360, 540, 720]},
 |      ...                             index=[['A', 'A', 'A', 'B', 'B', 'B'],
 |      ...                                    ['circle', 'triangle', 'rectangle',
 |      ...                                     'square', 'pentagon', 'hexagon']])
 |      >>> df_multindex
 |                   angles  degrees
 |      A circle          0      360
 |        triangle        3      180
 |        rectangle       4      360
 |      B square          4      360
 |        pentagon        5      540
 |        hexagon         6      720
 |
 |      >>> df.div(df_multindex, level=1, fill_value=0)
 |                   angles  degrees
 |      A circle        NaN      1.0
 |        triangle      1.0      1.0
 |        rectangle     1.0      1.0
```

```
 |      B square        0.0      0.0
 |        pentagon      0.0      0.0
 |        hexagon       0.0      0.0
 |  
 |  subtract = sub(self, other, axis='columns', level=None, fill_value=None)
 |  
 |  sum(self, axis=None, skipna=None, level=None, numeric_only=None, min_count=0,
**kwargs)
 |      Return the sum of the values over the requested axis.
 |      
 |      This is equivalent to the method ``numpy.sum``.
 |      
 |      Parameters
 |      ----------
 |      axis : {index (0), columns (1)}
 |          Axis for the function to be applied on.
 |      skipna : bool, default True
 |          Exclude NA/null values when computing the result.
 |      level : int or level name, default None
 |          If the axis is a MultiIndex (hierarchical), count along a
 |          particular level, collapsing into a Series.
 |      numeric_only : bool, default None
 |          Include only float, int, boolean columns. If None, will attempt to use
 |          everything, then use only numeric data. Not implemented for Series.
 |      min_count : int, default 0
 |          The required number of valid values to perform the operation. If fewer
than
 |          ``min_count`` non-NA values are present the result will be NA.
 |      **kwargs
 |          Additional keyword arguments to be passed to the function.
 |      
 |      Returns
 |      -------
 |      Series or DataFrame (if level specified)
 |      
 |      See Also
 |      --------
 |      Series.sum : Return the sum.
 |      Series.min : Return the minimum.
 |      Series.max : Return the maximum.
 |      Series.idxmin : Return the index of the minimum.
 |      Series.idxmax : Return the index of the maximum.
 |      DataFrame.sum : Return the sum over the requested axis.
 |      DataFrame.min : Return the minimum over the requested axis.
 |      DataFrame.max : Return the maximum over the requested axis.
 |      DataFrame.idxmin : Return the index of the minimum over the requested axis.
 |      DataFrame.idxmax : Return the index of the maximum over the requested axis.
 |      
 |      Examples
 |      --------
 |      >>> idx = pd.MultiIndex.from_arrays([
 |      ...     ['warm', 'warm', 'cold', 'cold'],
 |      ...     ['dog', 'falcon', 'fish', 'spider']],
 |      ...     names=['blooded', 'animal'])
 |      >>> s = pd.Series([4, 2, 0, 8], name='legs', index=idx)
 |      >>> s
 |      blooded  animal
 |      warm     dog       4
 |               falcon    2
 |      cold     fish      0
 |               spider    8
 |      Name: legs, dtype: int64
 |      
 |      >>> s.sum()
 |      14
 |      
 |      By default, the sum of an empty or all-NA Series is ``0``.
 |      
 |      >>> pd.Series([], dtype="float64").sum()  # min_count=0 is the default
 |      0.0
 |      
 |      This can be controlled with the ``min_count`` parameter. For example, if
 |      you'd like the sum of an empty series to be NaN, pass ``min_count=1``.
 |      
 |      >>> pd.Series([], dtype="float64").sum(min_count=1)
 |      nan
 |      
 |      Thanks to the ``skipna`` parameter, ``min_count`` handles all-NA and
 |      empty series identically.
 |      
 |      >>> pd.Series([np.nan]).sum()
 |      0.0
 |      
 |      >>> pd.Series([np.nan]).sum(min_count=1)
 |      nan
 |  
 |  swaplevel(self, i: 'Axis' = -2, j: 'Axis' = -1, axis: 'Axis' = 0) -> 'DataFrame'
 |      Swap levels i and j in a :class:`MultiIndex`.
 |      
 |      Default is to swap the two innermost levels of the index.
 |      
 |      Parameters
 |      ----------
 |      i, j : int or str
 |          Levels of the indices to be swapped. Can pass level name as string.
 |      axis : {0 or 'index', 1 or 'columns'}, default 0
 |                  The axis to swap levels on. 0 or 'index' for row-wise, 1 or
 |                  'columns' for column-wise.
 |      
 |      Returns
 |      -------
 |      DataFrame
 |          DataFrame with levels swapped in MultiIndex.
 |      
 |      Examples
 |              --------
 |              >>> df = pd.DataFrame(
 |              ...     {"Grade": ["A", "B", "A", "C"]},
 |              ...     index=[
 |              ...         ["Final exam", "Final exam", "Coursework", "Coursework"],
 |              ...         ["History", "Geography", "History", "Geography"],
 |              ...         ["January", "February", "March", "April"],
 |              ...     ],
 |              ... )
 |              >>> df
 |                                                  Grade
 |              Final exam  History     January      A
 |                          Geography   February     B
 |              Coursework  History     March        A
 |                          Geography   April        C
 |      
 |              In the following example, we will swap the levels of the indices.
 |              Here, we will swap the levels column-wise, but levels can be swapped
row-wise
 |              in a similar manner. Note that column-wise is the default behaviour.
 |              By not supplying any arguments for i and j, we swap the last and
second to
 |              last indices.
 |      
 |              >>> df.swaplevel()
 |                                                  Grade
 |              Final exam  January     History         A
 |                          February    Geography       B
 |              Coursework  March       History         A
 |                          April       Geography       C
 |      
 |              By supplying one argument, we can choose which index to swap the last
 |              index with. We can for example swap the first index with the last one
as
 |              follows.
 |      
 |              >>> df.swaplevel(0)
 |                                                  Grade
 |              January     History     Final exam      A
 |              February    Geography   Final exam      B
 |              March       History     Coursework      A
 |              April       Geography   Coursework      C
 |      
 |              We can also define explicitly which indices we want to swap by
supplying values
 |              for both i and j. Here, we for example swap the first and second
indices.
 |      
 |              >>> df.swaplevel(0, 1)
 |                                                  Grade
 |              History     Final exam  January         A
 |              Geography   Final exam  February        B
 |              History     Coursework  March           A
 |              Geography   Coursework  April           C
 |  
 |  to_dict(self, orient: 'str' = 'dict', into=<class 'dict'>)
 |      Convert the DataFrame to a dictionary.
 |      
 |      The type of the key-value pairs can be customized with the parameters
 |      (see below).
 |      
 |      Parameters
 |      ----------
 |      orient : str {'dict', 'list', 'series', 'split', 'records', 'index'}
 |          Determines the type of the values of the dictionary.
 |      
 |          - 'dict' (default) : dict like {column -> {index -> value}}
 |          - 'list' : dict like {column -> [values]}
 |          - 'series' : dict like {column -> Series(values)}
 |          - 'split' : dict like
 |            {'index' -> [index], 'columns' -> [columns], 'data' -> [values]}
 |          - 'records' : list like
 |            [{column -> value}, ... , {column -> value}]
 |          - 'index' : dict like {index -> {column -> value}}
 |
 |          Abbreviations are allowed. `s` indicates `series` and `sp`
 |          indicates `split`.
 |
 |      into : class, default dict
 |          The collections.abc.Mapping subclass used for all Mappings
 |          in the return value.  Can be the actual class or an empty
 |          instance of the mapping type you want.  If you want a
 |          collections.defaultdict, you must pass it initialized.
 |
 |      Returns
 |      -------
 |      dict, list or collections.abc.Mapping
 |          Return a collections.abc.Mapping object representing the DataFrame.
 |          The resulting transformation depends on the `orient` parameter.
 |
 |      See Also
 |      --------
 |      DataFrame.from_dict: Create a DataFrame from a dictionary.
 |      DataFrame.to_json: Convert a DataFrame to JSON format.
 |
 |      Examples
 |      --------
 |      >>> df = pd.DataFrame({'col1': [1, 2],
 |      ...                    'col2': [0.5, 0.75]},
 |      ...                   index=['row1', 'row2'])
 |      >>> df
 |            col1  col2
 |      row1     1  0.50
 |      row2     2  0.75
 |      >>> df.to_dict()
 |      {'col1': {'row1': 1, 'row2': 2}, 'col2': {'row1': 0.5, 'row2': 0.75}}
 |
 |      You can specify the return orientation.
 |
 |      >>> df.to_dict('series')
 |      {'col1': row1    1
 |               row2    2
 |      Name: col1, dtype: int64,
 |      'col2': row1    0.50
 |              row2    0.75
 |      Name: col2, dtype: float64}
 |
 |      >>> df.to_dict('split')
 |      {'index': ['row1', 'row2'], 'columns': ['col1', 'col2'],
 |       'data': [[1, 0.5], [2, 0.75]]}
 |
 |      >>> df.to_dict('records')
 |      [{'col1': 1, 'col2': 0.5}, {'col1': 2, 'col2': 0.75}]
 |
 |      >>> df.to_dict('index')
 |      {'row1': {'col1': 1, 'col2': 0.5}, 'row2': {'col1': 2, 'col2': 0.75}}
 |
 |      You can also specify the mapping type.
 |
 |      >>> from collections import OrderedDict, defaultdict
 |      >>> df.to_dict(into=OrderedDict)
 |      OrderedDict([('col1', OrderedDict([('row1', 1), ('row2', 2)])),
 |                   ('col2', OrderedDict([('row1', 0.5), ('row2', 0.75)]))])
 |
 |      If you want a `defaultdict`, you need to initialize it:
 |
 |      >>> dd = defaultdict(list)
 |      >>> df.to_dict('records', into=dd)
 |      [defaultdict(<class 'list'>, {'col1': 1, 'col2': 0.5}),
 |       defaultdict(<class 'list'>, {'col1': 2, 'col2': 0.75})]
 |
 |  to_feather(self, path: 'FilePathOrBuffer[AnyStr]', **kwargs) -> 'None'
 |      Write a DataFrame to the binary Feather format.
 |
 |      Parameters
 |      ----------
 |      path : str or file-like object
 |          If a string, it will be used as Root Directory path.
 |      **kwargs :
 |          Additional keywords passed to :func:`pyarrow.feather.write_feather`.
 |          Starting with pyarrow 0.17, this includes the `compression`,
 |          `compression_level`, `chunksize` and `version` keywords.
 |
 |          .. versionadded:: 1.1.0
 |
 |  to_gbq(self, destination_table: 'str', project_id: 'str | None' = None,
chunksize: 'int | None' = None, reauth: 'bool' = False, if_exists: 'str' = 'fail',
auth_local_webserver: 'bool' = False, table_schema: 'list[dict[str, str]] | None' =
None, location: 'str | None' = None, progress_bar: 'bool' = True, credentials=None) -
> 'None'
 |      Write a DataFrame to a Google BigQuery table.
 |      
 |      This function requires the `pandas-gbq package
 |      <https://pandas-gbq.readthedocs.io>`__.
 |      
 |      See the `How to authenticate with Google BigQuery
 |      <https://pandas-gbq.readthedocs.io/en/latest/howto/authentication.html>`__
 |      guide for authentication instructions.
 |      
 |      Parameters
 |      ----------
 |      destination_table : str
 |          Name of table to be written, in the form ``dataset.tablename``.
 |      project_id : str, optional
 |          Google BigQuery Account project ID. Optional when available from
 |          the environment.
 |      chunksize : int, optional
 |          Number of rows to be inserted in each chunk from the dataframe.
 |          Set to ``None`` to load the whole dataframe at once.
 |      reauth : bool, default False
 |          Force Google BigQuery to re-authenticate the user. This is useful
 |          if multiple accounts are used.
 |      if_exists : str, default 'fail'
 |          Behavior when the destination table exists. Value can be one of:
 |      
 |          ``'fail'``
 |              If table exists raise pandas_gbq.gbq.TableCreationError.
 |          ``'replace'``
 |              If table exists, drop it, recreate it, and insert data.
 |          ``'append'``
 |              If table exists, insert data. Create if does not exist.
 |      auth_local_webserver : bool, default False
 |          Use the `local webserver flow`_ instead of the `console flow`_
 |          when getting user credentials.
 |      
 |          .. _local webserver flow:
 |              https://google-auth-
oauthlib.readthedocs.io/en/latest/reference/google_auth_oauthlib.flow.html#google_aut
h_oauthlib.flow.InstalledAppFlow.run_local_server
 |          .. _console flow:
 |              https://google-auth-
oauthlib.readthedocs.io/en/latest/reference/google_auth_oauthlib.flow.html#google_aut
h_oauthlib.flow.InstalledAppFlow.run_console
 |      
 |          *New in version 0.2.0 of pandas-gbq*.
 |      table_schema : list of dicts, optional
 |          List of BigQuery table fields to which according DataFrame
 |          columns conform to, e.g. ``[{'name': 'col1', 'type':
 |          'STRING'},...]``. If schema is not provided, it will be
 |          generated according to dtypes of DataFrame columns. See
 |          BigQuery API documentation on available names of a field.
 |      
 |          *New in version 0.3.1 of pandas-gbq*.
 |      location : str, optional
 |          Location where the load job should run. See the `BigQuery locations
 |          documentation
 |          <https://cloud.google.com/bigquery/docs/dataset-locations>`__ for a
 |          list of available locations. The location must match that of the
 |          target dataset.
 |      
 |          *New in version 0.5.0 of pandas-gbq*.
 |      progress_bar : bool, default True
 |          Use the library `tqdm` to show the progress bar for the upload,
 |          chunk by chunk.
 |      
 |          *New in version 0.5.0 of pandas-gbq*.
 |      credentials : google.auth.credentials.Credentials, optional
 |          Credentials for accessing Google APIs. Use this parameter to
 |          override default credentials, such as to use Compute Engine
 |          :class:`google.auth.compute_engine.Credentials` or Service
 |          Account :class:`google.oauth2.service_account.Credentials`
 |          directly.
 |      
 |          *New in version 0.8.0 of pandas-gbq*.
 |      
 |      See Also
 |      --------
 |      pandas_gbq.to_gbq : This function in the pandas-gbq library.
 |      read_gbq : Read a DataFrame from Google BigQuery.
 |  
 |  to_html(self, buf: 'FilePathOrBuffer[str] | None' = None, columns: 'Sequence[str]
| None' = None, col_space: 'ColspaceArgType | None' = None, header: 'bool |
Sequence[str]' = True, index: 'bool' = True, na_rep: 'str' = 'NaN', formatters:
```

```
'FormattersType | None' = None, float_format: 'FloatFormatType | None' = None,
sparsify: 'bool | None' = None, index_names: 'bool' = True, justify: 'str | None' =
None, max_rows: 'int | None' = None, max_cols: 'int | None' = None, show_dimensions:
'bool | str' = False, decimal: 'str' = '.', bold_rows: 'bool' = True, classes: 'str |
list | tuple | None' = None, escape: 'bool' = True, notebook: 'bool' = False, border:
'int | None' = None, table_id: 'str | None' = None, render_links: 'bool' = False,
encoding: 'str | None' = None)
 |      Render a DataFrame as an HTML table.
 |
 |      Parameters
 |      ----------
 |      buf : str, Path or StringIO-like, optional, default None
 |          Buffer to write to. If None, the output is returned as a string.
 |      columns : sequence, optional, default None
 |          The subset of columns to write. Writes all columns by default.
 |      col_space : str or int, list or dict of int or str, optional
 |          The minimum width of each column in CSS length units.  An int is assumed
to be px units.
 |
 |          .. versionadded:: 0.25.0
 |              Ability to use str.
 |      header : bool, optional
 |          Whether to print column labels, default True.
 |      index : bool, optional, default True
 |          Whether to print index (row) labels.
 |      na_rep : str, optional, default 'NaN'
 |          String representation of ``NaN`` to use.
 |      formatters : list, tuple or dict of one-param. functions, optional
 |          Formatter functions to apply to columns' elements by position or
 |          name.
 |          The result of each function must be a unicode string.
 |          List/tuple must be of length equal to the number of columns.
 |      float_format : one-parameter function, optional, default None
 |          Formatter function to apply to columns' elements if they are
 |          floats. This function must return a unicode string and will be
 |          applied only to the non-``NaN`` elements, with ``NaN`` being
 |          handled by ``na_rep``.
 |
 |          .. versionchanged:: 1.2.0
 |
 |      sparsify : bool, optional, default True
 |          Set to False for a DataFrame with a hierarchical index to print
 |          every multiindex key at each row.
 |      index_names : bool, optional, default True
 |          Prints the names of the indexes.
 |      justify : str, default None
 |          How to justify the column labels. If None uses the option from
 |          the print configuration (controlled by set_option), 'right' out
 |          of the box. Valid values are
 |
 |          * left
 |          * right
 |          * center
 |          * justify
 |          * justify-all
 |          * start
 |          * end
 |          * inherit
 |          * match-parent
 |          * initial
 |          * unset.
 |      max_rows : int, optional
 |          Maximum number of rows to display in the console.
 |      min_rows : int, optional
 |          The number of rows to display in the console in a truncated repr
 |          (when number of rows is above `max_rows`).
 |      max_cols : int, optional
 |          Maximum number of columns to display in the console.
 |      show_dimensions : bool, default False
 |          Display DataFrame dimensions (number of rows by number of columns).
 |      decimal : str, default '.'
 |          Character recognized as decimal separator, e.g. ',' in Europe.
 |
 |      bold_rows : bool, default True
 |          Make the row labels bold in the output.
 |      classes : str or list or tuple, default None
 |          CSS class(es) to apply to the resulting html table.
 |      escape : bool, default True
 |          Convert the characters <, >, and & to HTML-safe sequences.
 |      notebook : {True, False}, default False
 |          Whether the generated HTML is for IPython Notebook.
 |      border : int
 |          A ``border=border`` attribute is included in the opening
 |          `<table>` tag. Default ``pd.options.display.html.border``.
 |      encoding : str, default "utf-8"
 |          Set character encoding.
 |
 |          .. versionadded:: 1.0
 |
 |      table_id : str, optional
 |          A css id is included in the opening `<table>` tag if specified.
 |      render_links : bool, default False
 |          Convert URLs to HTML links.
 |
 |      Returns
 |      -------
 |      str or None
 |          If buf is None, returns the result as a string. Otherwise returns
 |          None.
 |
 |      See Also
 |      --------
 |      to_string : Convert DataFrame to a string.
 |
 |  to_markdown(self, buf: 'IO[str] | str | None' = None, mode: 'str' = 'wt', index:
'bool' = True, storage_options: 'StorageOptions' = None, **kwargs) -> 'str | None'
 |      Print DataFrame in Markdown-friendly format.
 |
 |      .. versionadded:: 1.0.0
 |
 |      Parameters
 |      ----------
 |      buf : str, Path or StringIO-like, optional, default None
 |          Buffer to write to. If None, the output is returned as a string.
 |      mode : str, optional
 |          Mode in which file is opened, "wt" by default.
 |      index : bool, optional, default True
 |          Add index (row) labels.
 |
 |          .. versionadded:: 1.1.0
 |      storage_options : dict, optional
 |          Extra options that make sense for a particular storage connection, e.g.
 |          host, port, username, password, etc. For HTTP(S) URLs the key-value pairs
 |          are forwarded to ``urllib`` as header options. For other URLs (e.g.
 |          starting with "s3://", and "gcs://") the key-value pairs are forwarded to
 |          ``fsspec``. Please see ``fsspec`` and ``urllib`` for more details.
 |
 |          .. versionadded:: 1.2.0
 |
 |      **kwargs
 |          These parameters will be passed to `tabulate
<https://pypi.org/project/tabulate>`_.
 |
 |      Returns
 |      -------
 |      str
 |          DataFrame in Markdown-friendly format.
 |
 |      Notes
 |      -----
 |      Requires the `tabulate <https://pypi.org/project/tabulate>`_ package.
 |
 |      Examples
 |      --------
 |      >>> s = pd.Series(["elk", "pig", "dog", "quetzal"], name="animal")
 |      >>> print(s.to_markdown())
 |      |    | animal   |
 |      |---:|:---------|
 |      |  0 | elk      |
 |      |  1 | pig      |
 |      |  2 | dog      |
 |      |  3 | quetzal  |
 |
 |      Output markdown with a tabulate option.
 |
 |      >>> print(s.to_markdown(tablefmt="grid"))
 |      +----+----------+
 |      |    | animal   |
 |      +====+==========+
 |      |  0 | elk      |
 |      +----+----------+
 |      |  1 | pig      |
 |      +----+----------+
 |      |  2 | dog      |
 |      +----+----------+
 |      |  3 | quetzal  |
 |      +----+----------+
 |
 |  to_numpy(self, dtype: 'NpDtype | None' = None, copy: 'bool' = False, na_value=
<no_default>) -> 'np.ndarray'
 |      Convert the DataFrame to a NumPy array.
 |
 |      By default, the dtype of the returned array will be the common NumPy
 |      dtype of all types in the DataFrame. For example, if the dtypes are
 |      ``float16`` and ``float32``, the results dtype will be ``float32``.
 |      This may require copying data and coercing values, which may be
 |      expensive.
 |      
 |      Parameters
 |      ----------
 |      dtype : str or numpy.dtype, optional
 |          The dtype to pass to :meth:`numpy.asarray`.
 |      copy : bool, default False
 |          Whether to ensure that the returned value is not a view on
 |          another array. Note that ``copy=False`` does not *ensure* that
 |          ``to_numpy()`` is no-copy. Rather, ``copy=True`` ensure that
 |          a copy is made, even if not strictly necessary.
 |      na_value : Any, optional
 |          The value to use for missing values. The default value depends
 |          on `dtype` and the dtypes of the DataFrame columns.
 |      
 |          .. versionadded:: 1.1.0
 |      
 |      Returns
 |      -------
 |      numpy.ndarray
 |      
 |      See Also
 |      --------
 |      Series.to_numpy : Similar method for Series.
 |      
 |      Examples
 |      --------
 |      >>> pd.DataFrame({"A": [1, 2], "B": [3, 4]}).to_numpy()
 |      array([[1, 3],
 |             [2, 4]])
 |      
 |      With heterogeneous data, the lowest common type will have to
 |      be used.
 |      
 |      >>> df = pd.DataFrame({"A": [1, 2], "B": [3.0, 4.5]})
 |      >>> df.to_numpy()
 |      array([[1. , 3. ],
 |             [2. , 4.5]])
 |      
 |      For a mix of numeric and non-numeric types, the output array will
 |      have object dtype.
 |      
 |      >>> df['C'] = pd.date_range('2000', periods=2)
 |      >>> df.to_numpy()
 |      array([[1, 3.0, Timestamp('2000-01-01 00:00:00')],
 |             [2, 4.5, Timestamp('2000-01-02 00:00:00')]], dtype=object)
 |  
 |  to_parquet(self, path: 'FilePathOrBuffer | None' = None, engine: 'str' = 'auto',
compression: 'str | None' = 'snappy', index: 'bool | None' = None, partition_cols:
'list[str] | None' = None, storage_options: 'StorageOptions' = None, **kwargs) ->
'bytes | None'
 |      Write a DataFrame to the binary parquet format.
 |      
 |      This function writes the dataframe as a `parquet file
 |      <https://parquet.apache.org/>`_. You can choose different parquet
 |      backends, and have the option of compression. See
 |      :ref:`the user guide <io.parquet>` for more details.
 |      
 |      Parameters
 |      ----------
 |      path : str or file-like object, default None
 |          If a string, it will be used as Root Directory path
 |          when writing a partitioned dataset. By file-like object,
 |          we refer to objects with a write() method, such as a file handle
 |          (e.g. via builtin open function) or io.BytesIO. The engine
 |          fastparquet does not accept file-like objects. If path is None,
 |          a bytes object is returned.
 |      
 |          .. versionchanged:: 1.2.0
 |      
 |          Previously this was "fname"
 |      
 |      engine : {'auto', 'pyarrow', 'fastparquet'}, default 'auto'
 |          Parquet library to use. If 'auto', then the option
 |          ``io.parquet.engine`` is used. The default ``io.parquet.engine``
 |          behavior is to try 'pyarrow', falling back to 'fastparquet' if
 |          'pyarrow' is unavailable.
 |      compression : {'snappy', 'gzip', 'brotli', None}, default 'snappy'
 |          Name of the compression to use. Use ``None`` for no compression.
 |      index : bool, default None
 |          If ``True``, include the dataframe's index(es) in the file output.
 |          If ``False``, they will not be written to the file.
 |          If ``None``, similar to ``True`` the dataframe's index(es)
 |          will be saved. However, instead of being saved as values,
 |          the RangeIndex will be stored as a range in the metadata so it
 |          doesn't require much space and is faster. Other indexes will
 |          be included as columns in the file output.
 |      partition_cols : list, optional, default None
 |          Column names by which to partition the dataset.
 |          Columns are partitioned in the order they are given.
 |          Must be None if path is not a string.
 |      storage_options : dict, optional
 |          Extra options that make sense for a particular storage connection, e.g.
 |          host, port, username, password, etc. For HTTP(S) URLs the key-value pairs
 |          are forwarded to ``urllib`` as header options. For other URLs (e.g.
 |          starting with "s3://", and "gcs://") the key-value pairs are forwarded to
 |          ``fsspec``. Please see ``fsspec`` and ``urllib`` for more details.
 |
 |          .. versionadded:: 1.2.0
 |
 |      **kwargs
 |          Additional arguments passed to the parquet library. See
 |          :ref:`pandas io <io.parquet>` for more details.
 |
 |      Returns
 |      -------
 |      bytes if no path argument is provided else None
 |
 |      See Also
 |      --------
 |      read_parquet : Read a parquet file.
 |      DataFrame.to_csv : Write a csv file.
 |      DataFrame.to_sql : Write to a sql table.
 |      DataFrame.to_hdf : Write to hdf.
 |
 |      Notes
 |      -----
 |      This function requires either the `fastparquet
 |      <https://pypi.org/project/fastparquet>`_ or `pyarrow
 |      <https://arrow.apache.org/docs/python/>`_ library.
 |
 |      Examples
 |      --------
 |      >>> df = pd.DataFrame(data={'col1': [1, 2], 'col2': [3, 4]})
 |      >>> df.to_parquet('df.parquet.gzip',
 |      ...               compression='gzip')  # doctest: +SKIP
 |      >>> pd.read_parquet('df.parquet.gzip')  # doctest: +SKIP
 |         col1  col2
 |      0     1     3
 |      1     2     4
 |
 |      If you want to get a buffer to the parquet content you can use a io.BytesIO
 |      object, as long as you don't use partition_cols, which creates multiple
files.
 |
 |      >>> import io
 |      >>> f = io.BytesIO()
 |      >>> df.to_parquet(f)
 |      >>> f.seek(0)
 |      0
 |      >>> content = f.read()
 |
 |  to_period(self, freq: 'Frequency | None' = None, axis: 'Axis' = 0, copy: 'bool' =
True) -> 'DataFrame'
 |      Convert DataFrame from DatetimeIndex to PeriodIndex.
 |
 |      Convert DataFrame from DatetimeIndex to PeriodIndex with desired
 |      frequency (inferred from index if not passed).
 |
 |      Parameters
 |      ----------
 |      freq : str, default
 |          Frequency of the PeriodIndex.
 |      axis : {0 or 'index', 1 or 'columns'}, default 0
 |          The axis to convert (the index by default).
 |      copy : bool, default True
 |          If False then underlying input data is not copied.
 |
 |      Returns
 |      -------
 |      DataFrame with PeriodIndex
 |
 |  to_records(self, index=True, column_dtypes=None, index_dtypes=None) ->
'np.recarray'
 |      Convert DataFrame to a NumPy record array.
 |
 |      Index will be included as the first field of the record array if
 |      requested.
 |
 |      Parameters
 |      ----------
 |      index : bool, default True
 |          Include index in resulting record array, stored in 'index'
 |          field or using the index label, if set.
 |      column_dtypes : str, type, dict, default None
 |          If a string or type, the data type to store all columns. If
 |          a dictionary, a mapping of column names and indices (zero-indexed)
 |          to specific data types.
 |      index_dtypes : str, type, dict, default None
 |          If a string or type, the data type to store all index levels. If
 |          a dictionary, a mapping of index level names and indices
 |          (zero-indexed) to specific data types.
 |
 |          This mapping is applied only if `index=True`.
 |
 |      Returns
 |      -------
 |      numpy.recarray
 |          NumPy ndarray with the DataFrame labels as fields and each row
 |          of the DataFrame as entries.
 |
 |      See Also
 |      --------
 |      DataFrame.from_records: Convert structured or record ndarray
 |          to DataFrame.
 |      numpy.recarray: An ndarray that allows field access using
 |          attributes, analogous to typed columns in a
 |          spreadsheet.
 |
 |      Examples
 |      --------
 |      >>> df = pd.DataFrame({'A': [1, 2], 'B': [0.5, 0.75]},
 |      ...                   index=['a', 'b'])
 |      >>> df
 |         A     B
 |      a  1  0.50
 |      b  2  0.75
 |      >>> df.to_records()
 |      rec.array([('a', 1, 0.5 ), ('b', 2, 0.75)],
 |                dtype=[('index', 'O'), ('A', '<i8'), ('B', '<f8')])
 |
 |      If the DataFrame index has no label then the recarray field name
 |      is set to 'index'. If the index has a label then this is used as the
 |      field name:
 |
 |      >>> df.index = df.index.rename("I")
 |      >>> df.to_records()
 |      rec.array([('a', 1, 0.5 ), ('b', 2, 0.75)],
 |                dtype=[('I', 'O'), ('A', '<i8'), ('B', '<f8')])
 |
 |      The index can be excluded from the record array:
 |
 |      >>> df.to_records(index=False)
 |      rec.array([(1, 0.5 ), (2, 0.75)],
 |                dtype=[('A', '<i8'), ('B', '<f8')])
 |
 |      Data types can be specified for the columns:
 |
 |      >>> df.to_records(column_dtypes={"A": "int32"})
 |      rec.array([('a', 1, 0.5 ), ('b', 2, 0.75)],
 |                dtype=[('I', 'O'), ('A', '<i4'), ('B', '<f8')])
 |
 |      As well as for the index:
 |
 |      >>> df.to_records(index_dtypes="<S2")
 |      rec.array([(b'a', 1, 0.5 ), (b'b', 2, 0.75)],
 |                dtype=[('I', 'S2'), ('A', '<i8'), ('B', '<f8')])
 |
 |      >>> index_dtypes = f"<S{df.index.str.len().max()}"
 |      >>> df.to_records(index_dtypes=index_dtypes)
 |      rec.array([(b'a', 1, 0.5 ), (b'b', 2, 0.75)],
 |                dtype=[('I', 'S1'), ('A', '<i8'), ('B', '<f8')])
 |
 |  to_stata(self, path: 'FilePathOrBuffer', convert_dates: 'dict[Hashable, str] |
None' = None, write_index: 'bool' = True, byteorder: 'str | None' = None, time_stamp:
'datetime.datetime | None' = None, data_label: 'str | None' = None, variable_labels:
'dict[Hashable, str] | None' = None, version: 'int | None' = 114, convert_strl:
'Sequence[Hashable] | None' = None, compression: 'CompressionOptions' = 'infer',
storage_options: 'StorageOptions' = None) -> 'None'
 |      Export DataFrame object to Stata dta format.
 |
 |      Writes the DataFrame to a Stata dataset file.
 |      "dta" files contain a Stata dataset.
 |
 |      Parameters
 |      ----------
 |      path : str, buffer or path object
 |          String, path object (pathlib.Path or py._path.local.LocalPath) or
 |          object implementing a binary write() function. If using a buffer
 |          then the buffer will not be automatically closed after the file
 |          data has been written.
 |
 |          .. versionchanged:: 1.0.0
 |
 |          Previously this was "fname"
 |
 |      convert_dates : dict
 |          Dictionary mapping columns containing datetime types to stata
 |          internal format to use when writing the dates. Options are 'tc',
 |          'td', 'tm', 'tw', 'th', 'tq', 'ty'. Column can be either an integer
 |          or a name. Datetime columns that do not have a conversion type
 |          specified will be converted to 'tc'. Raises NotImplementedError if
 |          a datetime column has timezone information.
 |      write_index : bool
 |          Write the index to Stata dataset.
 |      byteorder : str
 |          Can be ">", "<", "little", or "big". default is `sys.byteorder`.
 |      time_stamp : datetime
 |          A datetime to use as file creation date.  Default is the current
 |          time.
 |      data_label : str, optional
 |          A label for the data set.  Must be 80 characters or smaller.
 |      variable_labels : dict
 |          Dictionary containing columns as keys and variable labels as
 |          values. Each label must be 80 characters or smaller.
 |      version : {114, 117, 118, 119, None}, default 114
 |          Version to use in the output dta file. Set to None to let pandas
 |          decide between 118 or 119 formats depending on the number of
 |          columns in the frame. Version 114 can be read by Stata 10 and
 |          later. Version 117 can be read by Stata 13 or later. Version 118
 |          is supported in Stata 14 and later. Version 119 is supported in
 |          Stata 15 and later. Version 114 limits string variables to 244
 |          characters or fewer while versions 117 and later allow strings
 |          with lengths up to 2,000,000 characters. Versions 118 and 119
 |          support Unicode characters, and version 119 supports more than
 |          32,767 variables.
 |
 |          Version 119 should usually only be used when the number of
 |          variables exceeds the capacity of dta format 118. Exporting
 |          smaller datasets in format 119 may have unintended consequences,
 |          and, as of November 2020, Stata SE cannot read version 119 files.
 |
 |          .. versionchanged:: 1.0.0
 |
 |              Added support for formats 118 and 119.
 |
 |      convert_strl : list, optional
 |          List of column names to convert to string columns to Stata StrL
 |          format. Only available if version is 117.  Storing strings in the
 |          StrL format can produce smaller dta files if strings have more than
 |          8 characters and values are repeated.
 |      compression : str or dict, default 'infer'
 |          For on-the-fly compression of the output dta. If string, specifies
 |          compression mode. If dict, value at key 'method' specifies
 |          compression mode. Compression mode must be one of {'infer', 'gzip',
 |          'bz2', 'zip', 'xz', None}. If compression mode is 'infer' and
 |          `fname` is path-like, then detect compression from the following
 |          extensions: '.gz', '.bz2', '.zip', or '.xz' (otherwise no
 |          compression). If dict and compression mode is one of {'zip',
 |          'gzip', 'bz2'}, or inferred as one of the above, other entries
 |          passed as additional compression options.
 |
 |          .. versionadded:: 1.1.0
 |
 |      storage_options : dict, optional
 |          Extra options that make sense for a particular storage connection, e.g.
 |          host, port, username, password, etc. For HTTP(S) URLs the key-value pairs
 |          are forwarded to ``urllib`` as header options. For other URLs (e.g.
 |          starting with "s3://", and "gcs://") the key-value pairs are forwarded to
 |          ``fsspec``. Please see ``fsspec`` and ``urllib`` for more details.
 |
 |          .. versionadded:: 1.2.0
 |
 |      Raises
 |      ------
 |      NotImplementedError
 |          * If datetimes contain timezone information
 |          * Column dtype is not representable in Stata
 |      ValueError
 |          * Columns listed in convert_dates are neither datetime64[ns]
 |            or datetime.datetime
 |          * Column listed in convert_dates is not in DataFrame
 |          * Categorical label contains more than 32,000 characters
 |
 |      See Also
 |      --------
```

```
 |      read_stata : Import Stata data files.
 |      io.stata.StataWriter : Low-level writer for Stata data files.
 |      io.stata.StataWriter117 : Low-level writer for version 117 files.
 |      
 |      Examples
 |      --------
 |      >>> df = pd.DataFrame({'animal': ['falcon', 'parrot', 'falcon',
 |      ...                               'parrot'],
 |      ...                    'speed': [350, 18, 361, 15]})
 |      >>> df.to_stata('animals.dta')  # doctest: +SKIP
 |  
 |  to_string(self, buf: 'FilePathOrBuffer[str] | None' = None, columns: 
'Sequence[str] | None' = None, col_space: 'int | None' = None, header: 'bool | 
Sequence[str]' = True, index: 'bool' = True, na_rep: 'str' = 'NaN', formatters: 
'fmt.FormattersType | None' = None, float_format: 'fmt.FloatFormatType | None' = 
None, sparsify: 'bool | None' = None, index_names: 'bool' = True, justify: 'str | 
None' = None, max_rows: 'int | None' = None, min_rows: 'int | None' = None, max_cols: 
'int | None' = None, show_dimensions: 'bool' = False, decimal: 'str' = '.', 
line_width: 'int | None' = None, max_colwidth: 'int | None' = None, encoding: 'str | 
None' = None) -> 'str | None'
 |      Render a DataFrame to a console-friendly tabular output.
 |      
 |      Parameters
 |      ----------
 |      buf : str, Path or StringIO-like, optional, default None
 |          Buffer to write to. If None, the output is returned as a string.
 |      columns : sequence, optional, default None
 |          The subset of columns to write. Writes all columns by default.
 |      col_space : int, list or dict of int, optional
 |          The minimum width of each column.
 |      header : bool or sequence, optional
 |          Write out the column names. If a list of strings is given, it is assumed 
to be aliases for the column names.
 |      index : bool, optional, default True
 |          Whether to print index (row) labels.
 |      na_rep : str, optional, default 'NaN'
 |          String representation of ``NaN`` to use.
 |      formatters : list, tuple or dict of one-param. functions, optional
 |          Formatter functions to apply to columns' elements by position or
 |          name.
 |          The result of each function must be a unicode string.
 |          List/tuple must be of length equal to the number of columns.
 |      float_format : one-parameter function, optional, default None
 |          Formatter function to apply to columns' elements if they are
 |          floats. This function must return a unicode string and will be
 |          applied only to the non-``NaN`` elements, with ``NaN`` being
 |          handled by ``na_rep``.
 |      
 |          .. versionchanged:: 1.2.0
 |      
 |      sparsify : bool, optional, default True
 |          Set to False for a DataFrame with a hierarchical index to print
 |          every multiindex key at each row.
 |      index_names : bool, optional, default True
 |          Prints the names of the indexes.
 |      justify : str, default None
 |          How to justify the column labels. If None uses the option from
 |          the print configuration (controlled by set_option), 'right' out
 |          of the box. Valid values are
 |      
 |          * left
 |          * right
 |          * center
 |          * justify
 |          * justify-all
 |          * start
 |          * end
 |          * inherit
 |          * match-parent
 |          * initial
 |          * unset.
 |      max_rows : int, optional
 |          Maximum number of rows to display in the console.
 |      min_rows : int, optional
 |          The number of rows to display in the console in a truncated repr
 |          (when number of rows is above `max_rows`).
 |      max_cols : int, optional
 |          Maximum number of columns to display in the console.
 |      show_dimensions : bool, default False
 |          Display DataFrame dimensions (number of rows by number of columns).
 |      decimal : str, default '.'
 |          Character recognized as decimal separator, e.g. ',' in Europe.
 |      
 |      line_width : int, optional
 |          Width to wrap a line in characters.
 |      max_colwidth : int, optional
 |          Max width to truncate each column in characters. By default, no limit.
 |
 |          .. versionadded:: 1.0.0
 |      encoding : str, default "utf-8"
 |          Set character encoding.
 |
 |          .. versionadded:: 1.0
 |
 |      Returns
 |      -------
 |      str or None
 |          If buf is None, returns the result as a string. Otherwise returns
 |          None.
 |
 |      See Also
 |      --------
 |      to_html : Convert DataFrame to HTML.
 |
 |      Examples
 |      --------
 |      >>> d = {'col1': [1, 2, 3], 'col2': [4, 5, 6]}
 |      >>> df = pd.DataFrame(d)
 |      >>> print(df.to_string())
 |         col1  col2
 |      0     1     4
 |      1     2     5
 |      2     3     6
 |
 |  to_timestamp(self, freq: 'Frequency | None' = None, how: 'str' = 'start', axis:
'Axis' = 0, copy: 'bool' = True) -> 'DataFrame'
 |      Cast to DatetimeIndex of timestamps, at *beginning* of period.
 |
 |      Parameters
 |      ----------
 |      freq : str, default frequency of PeriodIndex
 |          Desired frequency.
 |      how : {'s', 'e', 'start', 'end'}
 |          Convention for converting period to timestamp; start of period
 |          vs. end.
 |      axis : {0 or 'index', 1 or 'columns'}, default 0
 |          The axis to convert (the index by default).
 |      copy : bool, default True
 |          If False then underlying input data is not copied.
 |
 |      Returns
 |      -------
 |      DataFrame with DatetimeIndex
 |
 |  to_xml(self, path_or_buffer: 'FilePathOrBuffer | None' = None, index: 'bool' =
True, root_name: 'str | None' = 'data', row_name: 'str | None' = 'row', na_rep: 'str
| None' = None, attr_cols: 'str | list[str] | None' = None, elem_cols: 'str |
list[str] | None' = None, namespaces: 'dict[str | None, str] | None' = None, prefix:
'str | None' = None, encoding: 'str' = 'utf-8', xml_declaration: 'bool | None' =
True, pretty_print: 'bool | None' = True, parser: 'str | None' = 'lxml', stylesheet:
'FilePathOrBuffer | None' = None, compression: 'CompressionOptions' = 'infer',
storage_options: 'StorageOptions' = None) -> 'str | None'
 |      Render a DataFrame to an XML document.
 |
 |      .. versionadded:: 1.3.0
 |
 |      Parameters
 |      ----------
 |      path_or_buffer : str, path object or file-like object, optional
 |          File to write output to. If None, the output is returned as a
 |          string.
 |      index : bool, default True
 |          Whether to include index in XML document.
 |      root_name : str, default 'data'
 |          The name of root element in XML document.
 |      row_name : str, default 'row'
 |          The name of row element in XML document.
 |      na_rep : str, optional
 |          Missing data representation.
 |      attr_cols : list-like, optional
 |          List of columns to write as attributes in row element.
 |          Hierarchical columns will be flattened with underscore
 |          delimiting the different levels.
 |      elem_cols : list-like, optional
 |          List of columns to write as children in row element. By default,
 |          all columns output as children of row element. Hierarchical
 |          columns will be flattened with underscore delimiting the
 |          different levels.
 |      namespaces : dict, optional
 |          All namespaces to be defined in root element. Keys of dict
 |          should be prefix names and values of dict corresponding URIs.
 |          Default namespaces should be given empty string key. For
 |          example, ::
 |
 |              namespaces = {"": "https://example.com"}
 |
 |      prefix : str, optional
 |          Namespace prefix to be used for every element and/or attribute
 |          in document. This should be one of the keys in ``namespaces``
 |          dict.
 |      encoding : str, default 'utf-8'
 |          Encoding of the resulting document.
 |      xml_declaration : bool, default True
 |          Whether to include the XML declaration at start of document.
 |      pretty_print : bool, default True
 |          Whether output should be pretty printed with indentation and
 |          line breaks.
 |      parser : {'lxml','etree'}, default 'lxml'
 |          Parser module to use for building of tree. Only 'lxml' and
 |          'etree' are supported. With 'lxml', the ability to use XSLT
 |          stylesheet is supported.
 |      stylesheet : str, path object or file-like object, optional
 |          A URL, file-like object, or a raw string containing an XSLT
 |          script used to transform the raw XML output. Script should use
 |          layout of elements and attributes from original output. This
 |          argument requires ``lxml`` to be installed. Only XSLT 1.0
 |          scripts and not later versions is currently supported.
 |      compression : {'infer', 'gzip', 'bz2', 'zip', 'xz', None}, default 'infer'
 |          For on-the-fly decompression of on-disk data. If 'infer', then use
 |          gzip, bz2, zip or xz if path_or_buffer is a string ending in
 |          '.gz', '.bz2', '.zip', or 'xz', respectively, and no decompression
 |          otherwise. If using 'zip', the ZIP file must contain only one data
 |          file to be read in. Set to None for no decompression.
 |      storage_options : dict, optional
 |          Extra options that make sense for a particular storage connection, e.g.
 |          host, port, username, password, etc. For HTTP(S) URLs the key-value pairs
 |          are forwarded to ``urllib`` as header options. For other URLs (e.g.
 |          starting with "s3://", and "gcs://") the key-value pairs are forwarded to
 |          ``fsspec``. Please see ``fsspec`` and ``urllib`` for more details.
 |
 |      Returns
 |      -------
 |      None or str
 |          If ``io`` is None, returns the resulting XML format as a
 |          string. Otherwise returns None.
 |
 |      See Also
 |      --------
 |      to_json : Convert the pandas object to a JSON string.
 |      to_html : Convert DataFrame to a html.
 |
 |      Examples
 |      --------
 |      >>> df = pd.DataFrame({'shape': ['square', 'circle', 'triangle'],
 |      ...                    'degrees': [360, 360, 180],
 |      ...                    'sides': [4, np.nan, 3]})
 |
 |      >>> df.to_xml()  # doctest: +SKIP
 |      <?xml version='1.0' encoding='utf-8'?>
 |      <data>
 |        <row>
 |          <index>0</index>
 |          <shape>square</shape>
 |          <degrees>360</degrees>
 |          <sides>4.0</sides>
 |        </row>
 |        <row>
 |          <index>1</index>
 |          <shape>circle</shape>
 |          <degrees>360</degrees>
 |          <sides/>
 |        </row>
 |        <row>
 |          <index>2</index>
 |          <shape>triangle</shape>
 |          <degrees>180</degrees>
 |          <sides>3.0</sides>
 |        </row>
 |      </data>
 |
 |      >>> df.to_xml(attr_cols=[
 |      ...           'index', 'shape', 'degrees', 'sides'
 |      ...           ])  # doctest: +SKIP
 |      <?xml version='1.0' encoding='utf-8'?>
 |      <data>
 |        <row index="0" shape="square" degrees="360" sides="4.0"/>
 |        <row index="1" shape="circle" degrees="360"/>
 |        <row index="2" shape="triangle" degrees="180" sides="3.0"/>
 |      </data>
 |
 |      >>> df.to_xml(namespaces={"doc": "https://example.com"},
 |      ...           prefix="doc")  # doctest: +SKIP
 |      <?xml version='1.0' encoding='utf-8'?>
 |      <doc:data xmlns:doc="https://example.com">
 |        <doc:row>
 |          <doc:index>0</doc:index>
 |          <doc:shape>square</doc:shape>
 |          <doc:degrees>360</doc:degrees>
 |          <doc:sides>4.0</doc:sides>
 |        </doc:row>
 |        <doc:row>
 |          <doc:index>1</doc:index>
 |          <doc:shape>circle</doc:shape>
 |          <doc:degrees>360</doc:degrees>
 |          <doc:sides/>
 |        </doc:row>
 |        <doc:row>
 |          <doc:index>2</doc:index>
 |          <doc:shape>triangle</doc:shape>
 |          <doc:degrees>180</doc:degrees>
 |          <doc:sides>3.0</doc:sides>
 |        </doc:row>
 |      </doc:data>
 |
 |  transform(self, func: 'AggFuncType', axis: 'Axis' = 0, *args, **kwargs) ->
'DataFrame'
 |      Call ``func`` on self producing a DataFrame with transformed values.
 |
 |      Produced DataFrame will have same axis length as self.
 |
 |      Parameters
 |      ----------
 |      func : function, str, list-like or dict-like
 |          Function to use for transforming the data. If a function, must either
 |          work when passed a DataFrame or when passed to DataFrame.apply. If func
 |          is both list-like and dict-like, dict-like behavior takes precedence.
 |
 |          Accepted combinations are:
 |
 |          - function
 |          - string function name
 |          - list-like of functions and/or function names, e.g. ``[np.exp, 'sqrt']``
 |          - dict-like of axis labels -> functions, function names or list-like of
such.
 |      axis : {0 or 'index', 1 or 'columns'}, default 0
 |              If 0 or 'index': apply function to each column.
 |              If 1 or 'columns': apply function to each row.
 |      *args
 |          Positional arguments to pass to `func`.
 |      **kwargs
 |          Keyword arguments to pass to `func`.
 |
 |      Returns
 |      -------
 |      DataFrame
 |          A DataFrame that must have the same length as self.
 |
 |      Raises
 |      ------
 |      ValueError : If the returned DataFrame has a different length than self.
 |
 |      See Also
 |      --------
 |      DataFrame.agg : Only perform aggregating type operations.
 |      DataFrame.apply : Invoke function on a DataFrame.
 |
 |      Notes
 |      -----
 |      Functions that mutate the passed object can produce unexpected
 |      behavior or errors and are not supported. See :ref:`gotchas.udf-mutation`
 |      for more details.
 |
 |      Examples
 |      --------
 |      >>> df = pd.DataFrame({'A': range(3), 'B': range(1, 4)})
 |      >>> df
 |         A  B
 |      0  0  1
 |      1  1  2
 |      2  2  3
 |      >>> df.transform(lambda x: x + 1)
 |         A  B
 |      0  1  2
 |      1  2  3
 |      2  3  4
 |
 |      Even though the resulting DataFrame must have the same length as the
 |      input DataFrame, it is possible to provide several input functions:
```

```
|
|      >>> s = pd.Series(range(3))
|      >>> s
|      0    0
|      1    1
|      2    2
|      dtype: int64
|      >>> s.transform([np.sqrt, np.exp])
|             sqrt        exp
|      0  0.000000   1.000000
|      1  1.000000   2.718282
|      2  1.414214   7.389056
|
|      You can call transform on a GroupBy object:
|
|      >>> df = pd.DataFrame({
|      ...     "Date": [
|      ...         "2015-05-08", "2015-05-07", "2015-05-06", "2015-05-05",
|      ...         "2015-05-08", "2015-05-07", "2015-05-06", "2015-05-05"],
|      ...     "Data": [5, 8, 6, 1, 50, 100, 60, 120],
|      ... })
|      >>> df
|               Date  Data
|      0  2015-05-08     5
|      1  2015-05-07     8
|      2  2015-05-06     6
|      3  2015-05-05     1
|      4  2015-05-08    50
|      5  2015-05-07   100
|      6  2015-05-06    60
|      7  2015-05-05   120
|      >>> df.groupby('Date')['Data'].transform('sum')
|      0     55
|      1    108
|      2     66
|      3    121
|      4     55
|      5    108
|      6     66
|      7    121
|      Name: Data, dtype: int64
|
|      >>> df = pd.DataFrame({
|      ...     "c": [1, 1, 1, 2, 2, 2, 2],
|      ...     "type": ["m", "n", "o", "m", "m", "n", "n"]
|      ... })
|      >>> df
|         c type
|      0  1    m
|      1  1    n
|      2  1    o
|      3  2    m
|      4  2    m
|      5  2    n
|      6  2    n
|      >>> df['size'] = df.groupby('c')['type'].transform(len)
|      >>> df
|         c type size
|      0  1    m    3
|      1  1    n    3
|      2  1    o    3
|      3  2    m    4
|      4  2    m    4
|      5  2    n    4
|      6  2    n    4
|
|  transpose(self, *args, copy: 'bool' = False) -> 'DataFrame'
|      Transpose index and columns.
|
|      Reflect the DataFrame over its main diagonal by writing rows as columns
|      and vice-versa. The property :attr:`.T` is an accessor to the method
|      :meth:`transpose`.
|
|      Parameters
|      ----------
|      *args : tuple, optional
|          Accepted for compatibility with NumPy.
|      copy : bool, default False
|          Whether to copy the data after transposing, even for DataFrames
|          with a single dtype.
|
|          Note that a copy is always required for mixed dtype DataFrames,
|          or for DataFrames with any extension types.
|
|      Returns
|      -------
|      DataFrame
```

```
 |          The transposed DataFrame.
 |      
 |      See Also
 |      --------
 |      numpy.transpose : Permute the dimensions of a given array.
 |      
 |      Notes
 |      -----
 |      Transposing a DataFrame with mixed dtypes will result in a homogeneous
 |      DataFrame with the `object` dtype. In such a case, a copy of the data
 |      is always made.
 |      
 |      Examples
 |      --------
 |      **Square DataFrame with homogeneous dtype**
 |      
 |      >>> d1 = {'col1': [1, 2], 'col2': [3, 4]}
 |      >>> df1 = pd.DataFrame(data=d1)
 |      >>> df1
 |         col1  col2
 |      0     1     3
 |      1     2     4
 |      
 |      >>> df1_transposed = df1.T # or df1.transpose()
 |      >>> df1_transposed
 |            0  1
 |      col1  1  2
 |      col2  3  4
 |      
 |      When the dtype is homogeneous in the original DataFrame, we get a
 |      transposed DataFrame with the same dtype:
 |      
 |      >>> df1.dtypes
 |      col1    int64
 |      col2    int64
 |      dtype: object
 |      >>> df1_transposed.dtypes
 |      0    int64
 |      1    int64
 |      dtype: object
 |      
 |      **Non-square DataFrame with mixed dtypes**
 |      
 |      >>> d2 = {'name': ['Alice', 'Bob'],
 |      ...       'score': [9.5, 8],
 |      ...       'employed': [False, True],
 |      ...       'kids': [0, 0]}
 |      >>> df2 = pd.DataFrame(data=d2)
 |      >>> df2
 |          name  score  employed  kids
 |      0  Alice    9.5     False     0
 |      1    Bob    8.0      True     0
 |      
 |      >>> df2_transposed = df2.T # or df2.transpose()
 |      >>> df2_transposed
 |                    0     1
 |      name      Alice   Bob
 |      score       9.5   8.0
 |      employed  False  True
 |      kids          0     0
 |      
 |      When the DataFrame has mixed dtypes, we get a transposed DataFrame with
 |      the `object` dtype:
 |      
 |      >>> df2.dtypes
 |      name         object
 |      score       float64
 |      employed       bool
 |      kids          int64
 |      dtype: object
 |      >>> df2_transposed.dtypes
 |      0    object
 |      1    object
 |      dtype: object
 |  
 |  truediv(self, other, axis='columns', level=None, fill_value=None)
 |      Get Floating division of dataframe and other, element-wise (binary operator
`truediv`).
 |      
 |      Equivalent to ``dataframe / other``, but with support to substitute a
fill_value
 |      for missing data in one of the inputs. With reverse version, `rtruediv`.
 |      
 |      Among flexible wrappers (`add`, `sub`, `mul`, `div`, `mod`, `pow`) to
 |      arithmetic operators: `+`, `-`, `*`, `/`, `//`, `%`, `**`.
 |      
 |      Parameters
 |      ----------
 |      other : scalar, sequence, Series, or DataFrame
 |          Any single or multiple element data structure, or list-like object.
 |      axis : {0 or 'index', 1 or 'columns'}
 |          Whether to compare by the index (0 or 'index') or columns
 |          (1 or 'columns'). For Series input, axis to match Series index on.
 |      level : int or label
 |          Broadcast across a level, matching Index values on the
 |          passed MultiIndex level.
 |      fill_value : float or None, default None
 |          Fill existing missing (NaN) values, and any new element needed for
 |          successful DataFrame alignment, with this value before computation.
 |          If data in both corresponding DataFrame locations is missing
 |          the result will be missing.
 |
 |      Returns
 |      -------
 |      DataFrame
 |          Result of the arithmetic operation.
 |
 |      See Also
 |      --------
 |      DataFrame.add : Add DataFrames.
 |      DataFrame.sub : Subtract DataFrames.
 |      DataFrame.mul : Multiply DataFrames.
 |      DataFrame.div : Divide DataFrames (float division).
 |      DataFrame.truediv : Divide DataFrames (float division).
 |      DataFrame.floordiv : Divide DataFrames (integer division).
 |      DataFrame.mod : Calculate modulo (remainder after division).
 |      DataFrame.pow : Calculate exponential power.
 |
 |      Notes
 |      -----
 |      Mismatched indices will be unioned together.
 |
 |      Examples
 |      --------
 |      >>> df = pd.DataFrame({'angles': [0, 3, 4],
 |      ...                    'degrees': [360, 180, 360]},
 |      ...                   index=['circle', 'triangle', 'rectangle'])
 |      >>> df
 |                 angles  degrees
 |      circle          0      360
 |      triangle        3      180
 |      rectangle       4      360
 |
 |      Add a scalar with operator version which return the same
 |      results.
 |
 |      >>> df + 1
 |                 angles  degrees
 |      circle          1      361
 |      triangle        4      181
 |      rectangle       5      361
 |
 |      >>> df.add(1)
 |                 angles  degrees
 |      circle          1      361
 |      triangle        4      181
 |      rectangle       5      361
 |
 |      Divide by constant with reverse version.
 |
 |      >>> df.div(10)
 |                 angles  degrees
 |      circle        0.0     36.0
 |      triangle      0.3     18.0
 |      rectangle     0.4     36.0
 |
 |      >>> df.rdiv(10)
 |                   angles   degrees
 |      circle          inf  0.027778
 |      triangle   3.333333  0.055556
 |      rectangle  2.500000  0.027778
 |
 |      Subtract a list and Series by axis with operator version.
 |
 |      >>> df - [1, 2]
 |                 angles  degrees
 |      circle         -1      358
 |      triangle        2      178
 |      rectangle       3      358
 |
 |      >>> df.sub([1, 2], axis='columns')
 |                 angles  degrees
 |      circle         -1      358
 |      triangle        2      178
```

```
 |      rectangle       3      358
 |
 |      >>> df.sub(pd.Series([1, 1, 1], index=['circle', 'triangle', 'rectangle']),
 |      ...        axis='index')
 |                 angles  degrees
 |      circle         -1      359
 |      triangle        2      179
 |      rectangle       3      359
 |
 |      Multiply a DataFrame of different shape with operator version.
 |
 |      >>> other = pd.DataFrame({'angles': [0, 3, 4]},
 |      ...                      index=['circle', 'triangle', 'rectangle'])
 |      >>> other
 |                 angles
 |      circle          0
 |      triangle        3
 |      rectangle       4
 |
 |      >>> df * other
 |                 angles  degrees
 |      circle          0      NaN
 |      triangle        9      NaN
 |      rectangle      16      NaN
 |
 |      >>> df.mul(other, fill_value=0)
 |                 angles  degrees
 |      circle          0      0.0
 |      triangle        9      0.0
 |      rectangle      16      0.0
 |
 |      Divide by a MultiIndex by level.
 |
 |      >>> df_multiindex = pd.DataFrame({'angles': [0, 3, 4, 4, 5, 6],
 |      ...                              'degrees': [360, 180, 360, 360, 540, 720]},
 |      ...                             index=[['A', 'A', 'A', 'B', 'B', 'B'],
 |      ...                                    ['circle', 'triangle', 'rectangle',
 |      ...                                     'square', 'pentagon', 'hexagon']])
 |      >>> df_multiindex
 |                   angles  degrees
 |      A circle          0      360
 |        triangle        3      180
 |        rectangle       4      360
 |      B square          4      360
 |        pentagon        5      540
 |        hexagon         6      720
 |
 |      >>> df.div(df_multiindex, level=1, fill_value=0)
 |                   angles  degrees
 |      A circle        NaN      1.0
 |        triangle      1.0      1.0
 |        rectangle     1.0      1.0
 |      B square        0.0      0.0
 |        pentagon      0.0      0.0
 |        hexagon       0.0      0.0
 |
 |  unstack(self, level: 'Level' = -1, fill_value=None)
 |      Pivot a level of the (necessarily hierarchical) index labels.
 |
 |      Returns a DataFrame having a new level of column labels whose inner-most
level
 |      consists of the pivoted index labels.
 |
 |      If the index is not a MultiIndex, the output will be a Series
 |      (the analogue of stack when the columns are not a MultiIndex).
 |
 |      Parameters
 |      ----------
 |      level : int, str, or list of these, default -1 (last level)
 |          Level(s) of index to unstack, can pass level name.
 |      fill_value : int, str or dict
 |          Replace NaN with this value if the unstack produces missing values.
 |
 |      Returns
 |      -------
 |      Series or DataFrame
 |
 |      See Also
 |      --------
 |      DataFrame.pivot : Pivot a table based on column values.
 |      DataFrame.stack : Pivot a level of the column labels (inverse operation
 |          from `unstack`).
 |
 |      Examples
 |      --------
 |      >>> index = pd.MultiIndex.from_tuples([('one', 'a'), ('one', 'b'),
 |      ...                                    ('two', 'a'), ('two', 'b')])
 |      >>> s = pd.Series(np.arange(1.0, 5.0), index=index)
 |      >>> s
 |      one  a   1.0
 |           b   2.0
 |      two  a   3.0
 |           b   4.0
 |      dtype: float64
 |
 |      >>> s.unstack(level=-1)
 |           a   b
 |      one  1.0  2.0
 |      two  3.0  4.0
 |
 |      >>> s.unstack(level=0)
 |         one  two
 |      a  1.0   3.0
 |      b  2.0   4.0
 |
 |      >>> df = s.unstack(level=0)
 |      >>> df.unstack()
 |      one  a  1.0
 |           b  2.0
 |      two  a  3.0
 |           b  4.0
 |      dtype: float64
 |
 |  update(self, other, join: 'str' = 'left', overwrite: 'bool' = True,
filter_func=None, errors: 'str' = 'ignore') -> 'None'
 |      Modify in place using non-NA values from another DataFrame.
 |
 |      Aligns on indices. There is no return value.
 |
 |      Parameters
 |      ----------
 |      other : DataFrame, or object coercible into a DataFrame
 |          Should have at least one matching index/column label
 |          with the original DataFrame. If a Series is passed,
 |          its name attribute must be set, and that will be
 |          used as the column name to align with the original DataFrame.
 |      join : {'left'}, default 'left'
 |          Only left join is implemented, keeping the index and columns of the
 |          original object.
 |      overwrite : bool, default True
 |          How to handle non-NA values for overlapping keys:
 |
 |          * True: overwrite original DataFrame's values
 |            with values from `other`.
 |          * False: only update values that are NA in
 |            the original DataFrame.
 |
 |      filter_func : callable(1d-array) -> bool 1d-array, optional
 |          Can choose to replace values other than NA. Return True for values
 |          that should be updated.
 |      errors : {'raise', 'ignore'}, default 'ignore'
 |          If 'raise', will raise a ValueError if the DataFrame and `other`
 |          both contain non-NA data in the same place.
 |
 |      Returns
 |      -------
 |      None : method directly changes calling object
 |
 |      Raises
 |      ------
 |      ValueError
 |          * When `errors='raise'` and there's overlapping non-NA data.
 |          * When `errors` is not either `'ignore'` or `'raise'`
 |      NotImplementedError
 |          * If `join != 'left'`
 |
 |      See Also
 |      --------
 |      dict.update : Similar method for dictionaries.
 |      DataFrame.merge : For column(s)-on-column(s) operations.
 |
 |      Examples
 |      --------
 |      >>> df = pd.DataFrame({'A': [1, 2, 3],
 |      ...                    'B': [400, 500, 600]})
 |      >>> new_df = pd.DataFrame({'B': [4, 5, 6],
 |      ...                        'C': [7, 8, 9]})
 |      >>> df.update(new_df)
 |      >>> df
 |         A  B
 |      0  1  4
 |      1  2  5
 |      2  3  6
 |
 |      The DataFrame's length does not increase as a result of the update,
 |      only values at matching index/column labels are updated.
 |
 |      >>> df = pd.DataFrame({'A': ['a', 'b', 'c'],
 |      ...                    'B': ['x', 'y', 'z']})
 |      >>> new_df = pd.DataFrame({'B': ['d', 'e', 'f', 'g', 'h', 'i']})
 |      >>> df.update(new_df)
 |      >>> df
 |         A  B
 |      0  a  d
 |      1  b  e
 |      2  c  f
 |
 |      For Series, its name attribute must be set.
 |
 |      >>> df = pd.DataFrame({'A': ['a', 'b', 'c'],
 |      ...                    'B': ['x', 'y', 'z']})
 |      >>> new_column = pd.Series(['d', 'e'], name='B', index=[0, 2])
 |      >>> df.update(new_column)
 |      >>> df
 |         A  B
 |      0  a  d
 |      1  b  y
 |      2  c  e
 |      >>> df = pd.DataFrame({'A': ['a', 'b', 'c'],
 |      ...                    'B': ['x', 'y', 'z']})
 |      >>> new_df = pd.DataFrame({'B': ['d', 'e']}, index=[1, 2])
 |      >>> df.update(new_df)
 |      >>> df
 |         A  B
 |      0  a  x
 |      1  b  d
 |      2  c  e
 |
 |      If `other` contains NaNs the corresponding values are not updated
 |      in the original dataframe.
 |
 |      >>> df = pd.DataFrame({'A': [1, 2, 3],
 |      ...                    'B': [400, 500, 600]})
 |      >>> new_df = pd.DataFrame({'B': [4, np.nan, 6]})
 |      >>> df.update(new_df)
 |      >>> df
 |         A      B
 |      0  1    4.0
 |      1  2  500.0
 |      2  3    6.0
 |
 |  value_counts(self, subset: 'Sequence[Hashable] | None' = None, normalize: 'bool'
= False, sort: 'bool' = True, ascending: 'bool' = False, dropna: 'bool' = True)
 |      Return a Series containing counts of unique rows in the DataFrame.
 |
 |      .. versionadded:: 1.1.0
 |
 |      Parameters
 |      ----------
 |      subset : list-like, optional
 |          Columns to use when counting unique combinations.
 |      normalize : bool, default False
 |          Return proportions rather than frequencies.
 |      sort : bool, default True
 |          Sort by frequencies.
 |      ascending : bool, default False
 |          Sort in ascending order.
 |      dropna : bool, default True
 |          Don’t include counts of rows that contain NA values.
 |
 |          .. versionadded:: 1.3.0
 |
 |      Returns
 |      -------
 |      Series
 |
 |      See Also
 |      --------
 |      Series.value_counts: Equivalent method on Series.
 |
 |      Notes
 |      -----
 |      The returned Series will have a MultiIndex with one level per input
 |      column. By default, rows that contain any NA values are omitted from
 |      the result. By default, the resulting Series will be in descending
 |      order so that the first element is the most frequently-occurring row.
 |
 |      Examples
 |      --------
 |      >>> df = pd.DataFrame({'num_legs': [2, 4, 4, 6],
 |      ...                    'num_wings': [2, 0, 0, 0]},
 |      ...                   index=['falcon', 'dog', 'cat', 'ant'])
 |      >>> df
 |              num_legs  num_wings
 |      falcon         2          2
 |      dog            4          0
 |      cat            4          0
 |      ant            6          0
 |      
 |      >>> df.value_counts()
 |      num_legs  num_wings
 |      4         0            2
 |      2         2            1
 |      6         0            1
 |      dtype: int64
 |      
 |      >>> df.value_counts(sort=False)
 |      num_legs  num_wings
 |      2         2            1
 |      4         0            2
 |      6         0            1
 |      dtype: int64
 |      
 |      >>> df.value_counts(ascending=True)
 |      num_legs  num_wings
 |      2         2            1
 |      6         0            1
 |      4         0            2
 |      dtype: int64
 |      
 |      >>> df.value_counts(normalize=True)
 |      num_legs  num_wings
 |      4         0            0.50
 |      2         2            0.25
 |      6         0            0.25
 |      dtype: float64
 |      
 |      With `dropna` set to `False` we can also count rows with NA values.
 |      
 |      >>> df = pd.DataFrame({'first_name': ['John', 'Anne', 'John', 'Beth'],
 |      ...                    'middle_name': ['Smith', pd.NA, pd.NA, 'Louise']})
 |      >>> df
 |        first_name middle_name
 |      0       John       Smith
 |      1       Anne        <NA>
 |      2       John        <NA>
 |      3       Beth      Louise
 |      
 |      >>> df.value_counts()
 |      first_name  middle_name
 |      Beth        Louise         1
 |      John        Smith          1
 |      dtype: int64
 |      
 |      >>> df.value_counts(dropna=False)
 |      first_name  middle_name
 |      Anne        NaN            1
 |      Beth        Louise         1
 |      John        Smith          1
 |                  NaN            1
 |      dtype: int64
 |  
 |  var(self, axis=None, skipna=None, level=None, ddof=1, numeric_only=None,
**kwargs)
 |      Return unbiased variance over requested axis.
 |      
 |      Normalized by N-1 by default. This can be changed using the ddof argument
 |      
 |      Parameters
 |      ----------
 |      axis : {index (0), columns (1)}
 |      skipna : bool, default True
 |          Exclude NA/null values. If an entire row/column is NA, the result
 |          will be NA.
 |      level : int or level name, default None
 |          If the axis is a MultiIndex (hierarchical), count along a
 |          particular level, collapsing into a Series.
 |      ddof : int, default 1
 |          Delta Degrees of Freedom. The divisor used in calculations is N - ddof,
 |          where N represents the number of elements.
 |      numeric_only : bool, default None
 |          Include only float, int, boolean columns. If None, will attempt to use
 |          everything, then use only numeric data. Not implemented for Series.
 |      
 |      Returns
 |      -------
 |      Series or DataFrame (if level specified)
 |      
 |      Notes
 |      -----
 |      To have the same behaviour as `numpy.std`, use `ddof=0` (instead of the
 |      default `ddof=1`)
 |
 |  where(self, cond, other=nan, inplace=False, axis=None, level=None,
errors='raise', try_cast=<no_default>)
 |      Replace values where the condition is False.
 |
 |      Parameters
 |      ----------
 |      cond : bool Series/DataFrame, array-like, or callable
 |          Where `cond` is True, keep the original value. Where
 |          False, replace with corresponding value from `other`.
 |          If `cond` is callable, it is computed on the Series/DataFrame and
 |          should return boolean Series/DataFrame or array. The callable must
 |          not change input Series/DataFrame (though pandas doesn't check it).
 |      other : scalar, Series/DataFrame, or callable
 |          Entries where `cond` is False are replaced with
 |          corresponding value from `other`.
 |          If other is callable, it is computed on the Series/DataFrame and
 |          should return scalar or Series/DataFrame. The callable must not
 |          change input Series/DataFrame (though pandas doesn't check it).
 |      inplace : bool, default False
 |          Whether to perform the operation in place on the data.
 |      axis : int, default None
 |          Alignment axis if needed.
 |      level : int, default None
 |          Alignment level if needed.
 |      errors : str, {'raise', 'ignore'}, default 'raise'
 |          Note that currently this parameter won't affect
 |          the results and will always coerce to a suitable dtype.
 |
 |          - 'raise' : allow exceptions to be raised.
 |          - 'ignore' : suppress exceptions. On error return original object.
 |
 |      try_cast : bool, default None
 |          Try to cast the result back to the input type (if possible).
 |
 |          .. deprecated:: 1.3.0
 |              Manually cast back if necessary.
 |
 |      Returns
 |      -------
 |      Same type as caller or None if ``inplace=True``.
 |
 |      See Also
 |      --------
 |      :func:`DataFrame.mask` : Return an object of same shape as
 |          self.
 |
 |      Notes
 |      -----
 |      The where method is an application of the if-then idiom. For each
 |      element in the calling DataFrame, if ``cond`` is ``True`` the
 |      element is used; otherwise the corresponding element from the DataFrame
 |      ``other`` is used.
 |
 |      The signature for :func:`DataFrame.where` differs from
 |      :func:`numpy.where`. Roughly ``df1.where(m, df2)`` is equivalent to
 |      ``np.where(m, df1, df2)``.
 |
 |      For further details and examples see the ``where`` documentation in
 |      :ref:`indexing <indexing.where_mask>`.
 |
 |      Examples
 |      --------
 |      >>> s = pd.Series(range(5))
 |      >>> s.where(s > 0)
 |      0    NaN
 |      1    1.0
 |      2    2.0
 |      3    3.0
 |      4    4.0
 |      dtype: float64
 |      >>> s.mask(s > 0)
 |      0    0.0
 |      1    NaN
 |      2    NaN
 |      3    NaN
 |      4    NaN
 |      dtype: float64
 |
 |      >>> s.where(s > 1, 10)
 |      0    10
 |      1    10
 |      2    2
 |      3    3
 |      4    4
 |      dtype: int64
 |      >>> s.mask(s > 1, 10)
 |      0     0
 |      1     1
 |      2    10
 |      3    10
 |      4    10
 |      dtype: int64
 |
 |      >>> df = pd.DataFrame(np.arange(10).reshape(-1, 2), columns=['A', 'B'])
 |      >>> df
 |         A  B
 |      0  0  1
 |      1  2  3
 |      2  4  5
 |      3  6  7
 |      4  8  9
 |      >>> m = df % 3 == 0
 |      >>> df.where(m, -df)
 |         A  B
 |      0  0 -1
 |      1 -2  3
 |      2 -4 -5
 |      3  6 -7
 |      4 -8  9
 |      >>> df.where(m, -df) == np.where(m, df, -df)
 |            A     B
 |      0  True  True
 |      1  True  True
 |      2  True  True
 |      3  True  True
 |      4  True  True
 |      >>> df.where(m, -df) == df.mask(~m, -df)
 |            A     B
 |      0  True  True
 |      1  True  True
 |      2  True  True
 |      3  True  True
 |      4  True  True
 |
 |  ----------------------------------------------------------------------
 |  Class methods defined here:
 |
 |  from_dict(data, orient: 'str' = 'columns', dtype: 'Dtype | None' = None,
columns=None) -> 'DataFrame' from builtins.type
 |      Construct DataFrame from dict of array-like or dicts.
 |
 |      Creates DataFrame object from dictionary by columns or by index
 |      allowing dtype specification.
 |
 |      Parameters
 |      ----------
 |      data : dict
 |          Of the form {field : array-like} or {field : dict}.
 |      orient : {'columns', 'index'}, default 'columns'
 |          The "orientation" of the data. If the keys of the passed dict
 |          should be the columns of the resulting DataFrame, pass 'columns'
 |          (default). Otherwise if the keys should be rows, pass 'index'.
 |      dtype : dtype, default None
 |          Data type to force, otherwise infer.
 |      columns : list, default None
 |          Column labels to use when ``orient='index'``. Raises a ValueError
 |          if used with ``orient='columns'``.
 |
 |      Returns
 |      -------
 |      DataFrame
 |
 |      See Also
 |      --------
 |      DataFrame.from_records : DataFrame from structured ndarray, sequence
 |          of tuples or dicts, or DataFrame.
 |      DataFrame : DataFrame object creation using constructor.
 |
 |      Examples
 |      --------
 |      By default the keys of the dict become the DataFrame columns:
 |
 |      >>> data = {'col_1': [3, 2, 1, 0], 'col_2': ['a', 'b', 'c', 'd']}
 |      >>> pd.DataFrame.from_dict(data)
 |         col_1 col_2
 |      0      3     a
 |      1      2     b
 |      2      1     c
 |      3      0     d
 |
 |      Specify ``orient='index'`` to create the DataFrame using dictionary
 |      keys as rows:
 |
 |      >>> data = {'row_1': [3, 2, 1, 0], 'row_2': ['a', 'b', 'c', 'd']}
 |      >>> pd.DataFrame.from_dict(data, orient='index')
 |             0  1  2  3
 |      row_1  3  2  1  0
 |      row_2  a  b  c  d
 |
 |      When using the 'index' orientation, the column names can be
 |      specified manually:
 |
 |      >>> pd.DataFrame.from_dict(data, orient='index',
 |      ...                        columns=['A', 'B', 'C', 'D'])
 |             A  B  C  D
 |      row_1  3  2  1  0
 |      row_2  a  b  c  d
 |
 |  from_records(data, index=None, exclude=None, columns=None, coerce_float: 'bool' =
False, nrows: 'int | None' = None) -> 'DataFrame' from builtins.type
 |      Convert structured or record ndarray to DataFrame.
 |
 |      Creates a DataFrame object from a structured ndarray, sequence of
 |      tuples or dicts, or DataFrame.
 |
 |      Parameters
 |      ----------
 |      data : structured ndarray, sequence of tuples or dicts, or DataFrame
 |          Structured input data.
 |      index : str, list of fields, array-like
 |          Field of array to use as the index, alternately a specific set of
 |          input labels to use.
 |      exclude : sequence, default None
 |          Columns or fields to exclude.
 |      columns : sequence, default None
 |          Column names to use. If the passed data do not have names
 |          associated with them, this argument provides names for the
 |          columns. Otherwise this argument indicates the order of the columns
 |          in the result (any names not found in the data will become all-NA
 |          columns).
 |      coerce_float : bool, default False
 |          Attempt to convert values of non-string, non-numeric objects (like
 |          decimal.Decimal) to floating point, useful for SQL result sets.
 |      nrows : int, default None
 |          Number of rows to read if data is an iterator.
 |
 |      Returns
 |      -------
 |      DataFrame
 |
 |      See Also
 |      --------
 |      DataFrame.from_dict : DataFrame from dict of array-like or dicts.
 |      DataFrame : DataFrame object creation using constructor.
 |
 |      Examples
 |      --------
 |      Data can be provided as a structured ndarray:
 |
 |      >>> data = np.array([(3, 'a'), (2, 'b'), (1, 'c'), (0, 'd')],
 |      ...                 dtype=[('col_1', 'i4'), ('col_2', 'U1')])
 |      >>> pd.DataFrame.from_records(data)
 |         col_1 col_2
 |      0      3     a
 |      1      2     b
 |      2      1     c
 |      3      0     d
 |
 |      Data can be provided as a list of dicts:
 |
 |      >>> data = [{'col_1': 3, 'col_2': 'a'},
 |      ...         {'col_1': 2, 'col_2': 'b'},
 |      ...         {'col_1': 1, 'col_2': 'c'},
 |      ...         {'col_1': 0, 'col_2': 'd'}]
 |      >>> pd.DataFrame.from_records(data)
 |         col_1 col_2
 |      0      3     a
 |      1      2     b
 |      2      1     c
 |      3      0     d
 |
 |      Data can be provided as a list of tuples with corresponding columns:
 |
 |      >>> data = [(3, 'a'), (2, 'b'), (1, 'c'), (0, 'd')]
 |      >>> pd.DataFrame.from_records(data, columns=['col_1', 'col_2'])
 |         col_1 col_2
 |      0      3     a
 |      1      2     b
 |      2      1     c
 |      3      0     d
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors defined here:
 |
 |  T
 |
 |  axes
 |      Return a list representing the axes of the DataFrame.
 |
 |      It has the row axis labels and column axis labels as the only members.
 |      They are returned in that order.
 |
 |      Examples
 |      --------
 |      >>> df = pd.DataFrame({'col1': [1, 2], 'col2': [3, 4]})
 |      >>> df.axes
 |      [RangeIndex(start=0, stop=2, step=1), Index(['col1', 'col2'],
 |      dtype='object')]
 |
 |  columns
 |      The column labels of the DataFrame.
 |
 |  index
 |      The index (row labels) of the DataFrame.
 |
 |  shape
 |      Return a tuple representing the dimensionality of the DataFrame.
 |
 |      See Also
 |      --------
 |      ndarray.shape : Tuple of array dimensions.
 |
 |      Examples
 |      --------
 |      >>> df = pd.DataFrame({'col1': [1, 2], 'col2': [3, 4]})
 |      >>> df.shape
 |      (2, 2)
 |
 |      >>> df = pd.DataFrame({'col1': [1, 2], 'col2': [3, 4],
 |      ...                    'col3': [5, 6]})
 |      >>> df.shape
 |      (2, 3)
 |
 |  style
 |      Returns a Styler object.
 |
 |      Contains methods for building a styled HTML representation of the DataFrame.
 |
 |      See Also
 |      --------
 |      io.formats.style.Styler : Helps style a DataFrame or Series according to the
 |          data with HTML and CSS.
 |
 |  values
 |      Return a Numpy representation of the DataFrame.
 |
 |      .. warning::
 |
 |         We recommend using :meth:`DataFrame.to_numpy` instead.
 |
 |      Only the values in the DataFrame will be returned, the axes labels
 |      will be removed.
 |
 |      Returns
 |      -------
 |      numpy.ndarray
 |          The values of the DataFrame.
 |
 |      See Also
 |      --------
 |      DataFrame.to_numpy : Recommended alternative to this method.
 |      DataFrame.index : Retrieve the index labels.
 |      DataFrame.columns : Retrieving the column names.
 |
 |      Notes
 |      -----
 |      The dtype will be a lower-common-denominator dtype (implicit
 |      upcasting); that is to say if the dtypes (even of numeric types)
 |      are mixed, the one that accommodates all will be chosen. Use this
 |      with care if you are not dealing with the blocks.
 |
 |      e.g. If the dtypes are float16 and float32, dtype will be upcast to
 |      float32.  If dtypes are int32 and uint8, dtype will be upcast to
```

```
 |      int32. By :func:`numpy.find_common_type` convention, mixing int64
 |      and uint64 will result in a float64 dtype.
 |      
 |      Examples
 |      --------
 |      A DataFrame where all columns are the same type (e.g., int64) results
 |      in an array of the same type.
 |      
 |      >>> df = pd.DataFrame({'age':    [ 3,  29],
 |      ...                    'height': [94, 170],
 |      ...                    'weight': [31, 115]})
 |      >>> df
 |         age  height  weight
 |      0    3      94      31
 |      1   29     170     115
 |      >>> df.dtypes
 |      age       int64
 |      height    int64
 |      weight    int64
 |      dtype: object
 |      >>> df.values
 |      array([[  3,  94,  31],
 |             [ 29, 170, 115]])
 |      
 |      A DataFrame with mixed type columns(e.g., str/object, int64, float32)
 |      results in an ndarray of the broadest type that accommodates these
 |      mixed types (e.g., object).
 |      
 |      >>> df2 = pd.DataFrame([('parrot',   24.0, 'second'),
 |      ...                     ('lion',     80.5, 1),
 |      ...                     ('monkey', np.nan, None)],
 |      ...                   columns=('name', 'max_speed', 'rank'))
 |      >>> df2.dtypes
 |      name          object
 |      max_speed    float64
 |      rank          object
 |      dtype: object
 |      >>> df2.values
 |      array([['parrot', 24.0, 'second'],
 |             ['lion', 80.5, 1],
 |             ['monkey', nan, None]], dtype=object)
 |  
 |  ----------------------------------------------------------------------
 |  Data and other attributes defined here:
 |  
 |  __annotations__ = {'_AXIS_TO_AXIS_NUMBER': 'dict[Axis, int]', '_access...
 |  
 |  plot = <class 'pandas.plotting._core.PlotAccessor'>
 |      Make plots of Series or DataFrame.
 |      
 |      Uses the backend specified by the
 |      option ``plotting.backend``. By default, matplotlib is used.
 |      
 |      Parameters
 |      ----------
 |      data : Series or DataFrame
 |          The object for which the method is called.
 |      x : label or position, default None
 |          Only used if data is a DataFrame.
 |      y : label, position or list of label, positions, default None
 |          Allows plotting of one column versus another. Only used if data is a
 |          DataFrame.
 |      kind : str
 |          The kind of plot to produce:
 |      
 |          - 'line' : line plot (default)
 |          - 'bar' : vertical bar plot
 |          - 'barh' : horizontal bar plot
 |          - 'hist' : histogram
 |          - 'box' : boxplot
 |          - 'kde' : Kernel Density Estimation plot
 |          - 'density' : same as 'kde'
 |          - 'area' : area plot
 |          - 'pie' : pie plot
 |          - 'scatter' : scatter plot (DataFrame only)
 |          - 'hexbin' : hexbin plot (DataFrame only)
 |      ax : matplotlib axes object, default None
 |          An axes of the current figure.
 |      subplots : bool, default False
 |          Make separate subplots for each column.
 |      sharex : bool, default True if ax is None else False
 |          In case ``subplots=True``, share x axis and set some x axis labels
 |          to invisible; defaults to True if ax is None otherwise False if
 |          an ax is passed in; Be aware, that passing in both an ax and
 |          ``sharex=True`` will alter all x axis labels for all axis in a figure.
 |      sharey : bool, default False
 |          In case ``subplots=True``, share y axis and set some y axis labels to
```

```
invisible.
 |      layout : tuple, optional
 |          (rows, columns) for the layout of subplots.
 |      figsize : a tuple (width, height) in inches
 |          Size of a figure object.
 |      use_index : bool, default True
 |          Use index as ticks for x axis.
 |      title : str or list
 |          Title to use for the plot. If a string is passed, print the string
 |          at the top of the figure. If a list is passed and `subplots` is
 |          True, print each item in the list above the corresponding subplot.
 |      grid : bool, default None (matlab style default)
 |          Axis grid lines.
 |      legend : bool or {'reverse'}
 |          Place legend on axis subplots.
 |      style : list or dict
 |          The matplotlib line style per column.
 |      logx : bool or 'sym', default False
 |          Use log scaling or symlog scaling on x axis.
 |          .. versionchanged:: 0.25.0
 |
 |      logy : bool or 'sym' default False
 |          Use log scaling or symlog scaling on y axis.
 |          .. versionchanged:: 0.25.0
 |
 |      loglog : bool or 'sym', default False
 |          Use log scaling or symlog scaling on both x and y axes.
 |          .. versionchanged:: 0.25.0
 |
 |      xticks : sequence
 |          Values to use for the xticks.
 |      yticks : sequence
 |          Values to use for the yticks.
 |      xlim : 2-tuple/list
 |          Set the x limits of the current axes.
 |      ylim : 2-tuple/list
 |          Set the y limits of the current axes.
 |      xlabel : label, optional
 |          Name to use for the xlabel on x-axis. Default uses index name as xlabel,
or the
 |          x-column name for planar plots.
 |
 |          .. versionadded:: 1.1.0
 |
 |          .. versionchanged:: 1.2.0
 |
 |             Now applicable to planar plots (`scatter`, `hexbin`).
 |
 |      ylabel : label, optional
 |          Name to use for the ylabel on y-axis. Default will show no ylabel, or the
 |          y-column name for planar plots.
 |
 |          .. versionadded:: 1.1.0
 |
 |          .. versionchanged:: 1.2.0
 |
 |             Now applicable to planar plots (`scatter`, `hexbin`).
 |
 |      rot : int, default None
 |          Rotation for ticks (xticks for vertical, yticks for horizontal
 |          plots).
 |      fontsize : int, default None
 |          Font size for xticks and yticks.
 |      colormap : str or matplotlib colormap object, default None
 |          Colormap to select colors from. If string, load colormap with that
 |          name from matplotlib.
 |      colorbar : bool, optional
 |          If True, plot colorbar (only relevant for 'scatter' and 'hexbin'
 |          plots).
 |      position : float
 |          Specify relative alignments for bar plot layout.
 |          From 0 (left/bottom-end) to 1 (right/top-end). Default is 0.5
 |          (center).
 |      table : bool, Series or DataFrame, default False
 |          If True, draw a table using the data in the DataFrame and the data
 |          will be transposed to meet matplotlib's default layout.
 |          If a Series or DataFrame is passed, use passed data to draw a
 |          table.
 |      yerr : DataFrame, Series, array-like, dict and str
 |          See :ref:`Plotting with Error Bars <visualization.errorbars>` for
 |          detail.
 |      xerr : DataFrame, Series, array-like, dict and str
 |          Equivalent to yerr.
 |      stacked : bool, default False in line and bar plots, and True in area plot
 |          If True, create stacked plot.
 |      sort_columns : bool, default False
 |          Sort column names to determine plot ordering.
 |      secondary_y : bool or sequence, default False
 |          Whether to plot on the secondary y-axis if a list/tuple, which
 |          columns to plot on secondary y-axis.
 |      mark_right : bool, default True
 |          When using a secondary_y axis, automatically mark the column
 |          labels with "(right)" in the legend.
 |      include_bool : bool, default is False
 |          If True, boolean values can be plotted.
 |      backend : str, default None
 |          Backend to use instead of the backend specified in the option
 |          ``plotting.backend``. For instance, 'matplotlib'. Alternatively, to
 |          specify the ``plotting.backend`` for the whole session, set
 |          ``pd.options.plotting.backend``.
 |
 |          .. versionadded:: 1.0.0
 |
 |      **kwargs
 |          Options to pass to matplotlib plotting method.
 |
 |      Returns
 |      -------
 |      :class:`matplotlib.axes.Axes` or numpy.ndarray of them
 |          If the backend is not the default matplotlib one, the return value
 |          will be the object returned by the backend.
 |
 |      Notes
 |      -----
 |      - See matplotlib documentation online for more on this subject
 |      - If `kind` = 'bar' or 'barh', you can specify relative alignments
 |        for bar plot layout by `position` keyword.
 |        From 0 (left/bottom-end) to 1 (right/top-end). Default is 0.5
 |        (center)
 |
 |  sparse = <class 'pandas.core.arrays.sparse.accessor.SparseFrameAccesso...
 |      DataFrame accessor for sparse data.
 |
 |      .. versionadded:: 0.25.0
 |
 |  ----------------------------------------------------------------------
 |  Methods inherited from pandas.core.generic.NDFrame:
 |
 |  __abs__(self: 'FrameOrSeries') -> 'FrameOrSeries'
 |
 |  __array__(self, dtype: 'NpDtype | None' = None) -> 'np.ndarray'
 |
 |  __array_ufunc__(self, ufunc: 'np.ufunc', method: 'str', *inputs: 'Any', **kwargs:
'Any')
 |
 |  __array_wrap__(self, result: 'np.ndarray', context: 'tuple[Callable, tuple[Any,
...], int] | None' = None)
 |      Gets called after a ufunc and other functions.
 |
 |      Parameters
 |      ----------
 |      result: np.ndarray
 |          The result of the ufunc or other function called on the NumPy array
 |          returned by __array__
 |      context: tuple of (func, tuple, int)
 |          This parameter is returned by ufuncs as a 3-element tuple: (name of the
 |          ufunc, arguments of the ufunc, domain of the ufunc), but is not set by
 |          other numpy functions.q
 |
 |      Notes
 |      -----
 |      Series implements __array_ufunc_ so this not called for ufunc on Series.
 |
 |  __bool__ = __nonzero__(self)
 |
 |  __contains__(self, key) -> 'bool_t'
 |      True if the key is in the info axis
 |
 |  __copy__(self: 'FrameOrSeries', deep: 'bool_t' = True) -> 'FrameOrSeries'
 |
 |  __deepcopy__(self: 'FrameOrSeries', memo=None) -> 'FrameOrSeries'
 |      Parameters
 |      ----------
 |      memo, default None
 |          Standard signature. Unused
 |
 |  __delitem__(self, key) -> 'None'
 |      Delete item
 |
 |  __finalize__(self: 'FrameOrSeries', other, method: 'str | None' = None, **kwargs)
-> 'FrameOrSeries'
 |      Propagate metadata from other to self.
 |
 |      Parameters
 |      ----------
 |      other : the object from which to get the attributes that we are going
 |          to propagate
 |      method : str, optional
 |          A passed method name providing context on where ``__finalize__``
 |          was called.
 |
 |          .. warning::
 |
 |             The value passed as `method` are not currently considered
 |             stable across pandas releases.
 |
 |  __getattr__(self, name: 'str')
 |      After regular attribute access, try looking up the name
 |      This allows simpler access to columns for interactive use.
 |
 |  __getstate__(self) -> 'dict[str, Any]'
 |
 |  __iadd__(self, other)
 |
 |  __iand__(self, other)
 |
 |  __ifloordiv__(self, other)
 |
 |  __imod__(self, other)
 |
 |  __imul__(self, other)
 |
 |  __invert__(self)
 |
 |  __ior__(self, other)
 |
 |  __ipow__(self, other)
 |
 |  __isub__(self, other)
 |
 |  __iter__(self)
 |      Iterate over info axis.
 |
 |      Returns
 |      -------
 |      iterator
 |          Info axis as iterator.
 |
 |  __itruediv__(self, other)
 |
 |  __ixor__(self, other)
 |
 |  __neg__(self)
 |
 |  __nonzero__(self)
 |
 |  __pos__(self)
 |
 |  __round__(self: 'FrameOrSeries', decimals: 'int' = 0) -> 'FrameOrSeries'
 |
 |  __setattr__(self, name: 'str', value) -> 'None'
 |      After regular attribute access, try setting the name
 |      This allows simpler access to columns for interactive use.
 |
 |  __setstate__(self, state)
 |
 |  abs(self: 'FrameOrSeries') -> 'FrameOrSeries'
 |      Return a Series/DataFrame with absolute numeric value of each element.
 |
 |      This function only applies to elements that are all numeric.
 |
 |      Returns
 |      -------
 |      abs
 |          Series/DataFrame containing the absolute value of each element.
 |
 |      See Also
 |      --------
 |      numpy.absolute : Calculate the absolute value element-wise.
 |
 |      Notes
 |      -----
 |      For ``complex`` inputs, ``1.2 + 1j``, the absolute value is
 |      :math:`\sqrt{ a^2 + b^2 }`.
 |
 |      Examples
 |      --------
 |      Absolute numeric values in a Series.
 |
 |      >>> s = pd.Series([-1.10, 2, -3.33, 4])
 |      >>> s.abs()
 |      0    1.10
 |      1    2.00
 |      2    3.33
 |      3    4.00
 |      dtype: float64
 |
 |      Absolute numeric values in a Series with complex numbers.
 |
 |      >>> s = pd.Series([1.2 + 1j])
 |      >>> s.abs()
 |      0    1.56205
 |      dtype: float64
 |
 |      Absolute numeric values in a Series with a Timedelta element.
 |
 |      >>> s = pd.Series([pd.Timedelta('1 days')])
 |      >>> s.abs()
 |      0   1 days
 |      dtype: timedelta64[ns]
 |
 |      Select rows with data closest to certain value using argsort (from
 |      `StackOverflow <https://stackoverflow.com/a/17758115>`__).
 |
 |      >>> df = pd.DataFrame({
 |      ...     'a': [4, 5, 6, 7],
 |      ...     'b': [10, 20, 30, 40],
 |      ...     'c': [100, 50, -30, -50]
 |      ... })
 |      >>> df
 |           a    b    c
 |      0    4   10  100
 |      1    5   20   50
 |      2    6   30  -30
 |      3    7   40  -50
 |      >>> df.loc[(df.c - 43).abs().argsort()]
 |           a    b    c
 |      1    5   20   50
 |      0    4   10  100
 |      2    6   30  -30
 |      3    7   40  -50
 |
 |  add_prefix(self: 'FrameOrSeries', prefix: 'str') -> 'FrameOrSeries'
 |      Prefix labels with string `prefix`.
 |
 |      For Series, the row labels are prefixed.
 |      For DataFrame, the column labels are prefixed.
 |
 |      Parameters
 |      ----------
 |      prefix : str
 |          The string to add before each label.
 |
 |      Returns
 |      -------
 |      Series or DataFrame
 |          New Series or DataFrame with updated labels.
 |
 |      See Also
 |      --------
 |      Series.add_suffix: Suffix row labels with string `suffix`.
 |      DataFrame.add_suffix: Suffix column labels with string `suffix`.
 |
 |      Examples
 |      --------
 |      >>> s = pd.Series([1, 2, 3, 4])
 |      >>> s
 |      0    1
 |      1    2
 |      2    3
 |      3    4
 |      dtype: int64
 |
 |      >>> s.add_prefix('item_')
 |      item_0    1
 |      item_1    2
 |      item_2    3
 |      item_3    4
 |      dtype: int64
 |
 |      >>> df = pd.DataFrame({'A': [1, 2, 3, 4], 'B': [3, 4, 5, 6]})
 |      >>> df
 |         A  B
 |      0  1  3
 |      1  2  4
 |      2  3  5
 |      3  4  6
 |
 |      >>> df.add_prefix('col_')
 |           col_A  col_B
 |      0       1       3
 |      1       2       4
 |      2       3       5
 |      3       4       6
 |
 |  add_suffix(self: 'FrameOrSeries', suffix: 'str') -> 'FrameOrSeries'
 |      Suffix labels with string `suffix`.
 |
 |      For Series, the row labels are suffixed.
 |      For DataFrame, the column labels are suffixed.
 |
 |      Parameters
 |      ----------
 |      suffix : str
 |          The string to add after each label.
 |
 |      Returns
 |      -------
 |      Series or DataFrame
 |          New Series or DataFrame with updated labels.
 |
 |      See Also
 |      --------
 |      Series.add_prefix: Prefix row labels with string `prefix`.
 |      DataFrame.add_prefix: Prefix column labels with string `prefix`.
 |
 |      Examples
 |      --------
 |      >>> s = pd.Series([1, 2, 3, 4])
 |      >>> s
 |      0    1
 |      1    2
 |      2    3
 |      3    4
 |      dtype: int64
 |
 |      >>> s.add_suffix('_item')
 |      0_item    1
 |      1_item    2
 |      2_item    3
 |      3_item    4
 |      dtype: int64
 |
 |      >>> df = pd.DataFrame({'A': [1, 2, 3, 4], 'B': [3, 4, 5, 6]})
 |      >>> df
 |         A  B
 |      0  1  3
 |      1  2  4
 |      2  3  5
 |      3  4  6
 |
 |      >>> df.add_suffix('_col')
 |           A_col  B_col
 |      0       1       3
 |      1       2       4
 |      2       3       5
 |      3       4       6
 |
 |  asof(self, where, subset=None)
 |      Return the last row(s) without any NaNs before `where`.
 |
 |      The last row (for each element in `where`, if list) without any
 |      NaN is taken.
 |      In case of a :class:`~pandas.DataFrame`, the last row without NaN
 |      considering only the subset of columns (if not `None`)
 |
 |      If there is no good value, NaN is returned for a Series or
 |      a Series of NaN values for a DataFrame
 |
 |      Parameters
 |      ----------
 |      where : date or array-like of dates
 |          Date(s) before which the last row(s) are returned.
 |      subset : str or array-like of str, default `None`
 |          For DataFrame, if not `None`, only use these columns to
 |          check for NaNs.
 |
 |      Returns
 |      -------
 |      scalar, Series, or DataFrame
 |
 |          The return can be:
 |
 |          * scalar : when `self` is a Series and `where` is a scalar
 |          * Series: when `self` is a Series and `where` is an array-like,
 |            or when `self` is a DataFrame and `where` is a scalar
 |          * DataFrame : when `self` is a DataFrame and `where` is an
 |            array-like
 |      
 |          Return scalar, Series, or DataFrame.
 |      
 |      See Also
 |      --------
 |      merge_asof : Perform an asof merge. Similar to left join.
 |      
 |      Notes
 |      -----
 |      Dates are assumed to be sorted. Raises if this is not the case.
 |      
 |      Examples
 |      --------
 |      A Series and a scalar `where`.
 |      
 |      >>> s = pd.Series([1, 2, np.nan, 4], index=[10, 20, 30, 40])
 |      >>> s
 |      10    1.0
 |      20    2.0
 |      30    NaN
 |      40    4.0
 |      dtype: float64
 |      
 |      >>> s.asof(20)
 |      2.0
 |      
 |      For a sequence `where`, a Series is returned. The first value is
 |      NaN, because the first element of `where` is before the first
 |      index value.
 |      
 |      >>> s.asof([5, 20])
 |      5     NaN
 |      20    2.0
 |      dtype: float64
 |      
 |      Missing values are not considered. The following is ``2.0``, not
 |      NaN, even though NaN is at the index location for ``30``.
 |      
 |      >>> s.asof(30)
 |      2.0
 |      
 |      Take all columns into consideration
 |      
 |      >>> df = pd.DataFrame({'a': [10, 20, 30, 40, 50],
 |      ...                    'b': [None, None, None, None, 500]},
 |      ...                   index=pd.DatetimeIndex(['2018-02-27 09:01:00',
 |      ...                                           '2018-02-27 09:02:00',
 |      ...                                           '2018-02-27 09:03:00',
 |      ...                                           '2018-02-27 09:04:00',
 |      ...                                           '2018-02-27 09:05:00']))
 |      >>> df.asof(pd.DatetimeIndex(['2018-02-27 09:03:30',
 |      ...                           '2018-02-27 09:04:30']))
 |                            a   b
 |      2018-02-27 09:03:30 NaN NaN
 |      2018-02-27 09:04:30 NaN NaN
 |      
 |      Take a single column into consideration
 |      
 |      >>> df.asof(pd.DatetimeIndex(['2018-02-27 09:03:30',
 |      ...                           '2018-02-27 09:04:30']),
 |      ...         subset=['a'])
 |                               a   b
 |      2018-02-27 09:03:30   30.0 NaN
 |      2018-02-27 09:04:30   40.0 NaN
 |  
 |  astype(self: 'FrameOrSeries', dtype, copy: 'bool_t' = True, errors: 'str' = 
'raise') -> 'FrameOrSeries'
 |      Cast a pandas object to a specified dtype ``dtype``.
 |      
 |      Parameters
 |      ----------
 |      dtype : data type, or dict of column name -> data type
 |          Use a numpy.dtype or Python type to cast entire pandas object to
 |          the same type. Alternatively, use {col: dtype, ...}, where col is a
 |          column label and dtype is a numpy.dtype or Python type to cast one
 |          or more of the DataFrame's columns to column-specific types.
 |      copy : bool, default True
 |          Return a copy when ``copy=True`` (be very careful setting
 |          ``copy=False`` as changes to values then may propagate to other
 |          pandas objects).
 |      errors : {'raise', 'ignore'}, default 'raise'
 |          Control raising of exceptions on invalid data for provided dtype.
 |      
 |          - ``raise`` : allow exceptions to be raised
 |          - ``ignore`` : suppress exceptions. On error return original object.
 |
 |      Returns
 |      -------
 |      casted : same type as caller
 |
 |      See Also
 |      --------
 |      to_datetime : Convert argument to datetime.
 |      to_timedelta : Convert argument to timedelta.
 |      to_numeric : Convert argument to a numeric type.
 |      numpy.ndarray.astype : Cast a numpy array to a specified type.
 |
 |      Notes
 |      -----
 |      .. deprecated:: 1.3.0
 |
 |          Using ``astype`` to convert from timezone-naive dtype to
 |          timezone-aware dtype is deprecated and will raise in a
 |          future version.  Use :meth:`Series.dt.tz_localize` instead.
 |
 |      Examples
 |      --------
 |      Create a DataFrame:
 |
 |      >>> d = {'col1': [1, 2], 'col2': [3, 4]}
 |      >>> df = pd.DataFrame(data=d)
 |      >>> df.dtypes
 |      col1    int64
 |      col2    int64
 |      dtype: object
 |
 |      Cast all columns to int32:
 |
 |      >>> df.astype('int32').dtypes
 |      col1    int32
 |      col2    int32
 |      dtype: object
 |
 |      Cast col1 to int32 using a dictionary:
 |
 |      >>> df.astype({'col1': 'int32'}).dtypes
 |      col1    int32
 |      col2    int64
 |      dtype: object
 |
 |      Create a series:
 |
 |      >>> ser = pd.Series([1, 2], dtype='int32')
 |      >>> ser
 |      0    1
 |      1    2
 |      dtype: int32
 |      >>> ser.astype('int64')
 |      0    1
 |      1    2
 |      dtype: int64
 |
 |      Convert to categorical type:
 |
 |      >>> ser.astype('category')
 |      0    1
 |      1    2
 |      dtype: category
 |      Categories (2, int64): [1, 2]
 |
 |      Convert to ordered categorical type with custom ordering:
 |
 |      >>> from pandas.api.types import CategoricalDtype
 |      >>> cat_dtype = CategoricalDtype(
 |      ...     categories=[2, 1], ordered=True)
 |      >>> ser.astype(cat_dtype)
 |      0    1
 |      1    2
 |      dtype: category
 |      Categories (2, int64): [2 < 1]
 |
 |      Note that using ``copy=False`` and changing data on a new
 |      pandas object may propagate changes:
 |
 |      >>> s1 = pd.Series([1, 2])
 |      >>> s2 = s1.astype('int64', copy=False)
 |      >>> s2[0] = 10
 |      >>> s1  # note that s1[0] has changed too
 |      0    10
 |      1     2
 |      dtype: int64
```

```
 |
 |      Create a series of dates:
 |
 |      >>> ser_date = pd.Series(pd.date_range('20200101', periods=3))
 |      >>> ser_date
 |      0   2020-01-01
 |      1   2020-01-02
 |      2   2020-01-03
 |      dtype: datetime64[ns]
 |
 |  at_time(self: 'FrameOrSeries', time, asof: 'bool_t' = False, axis=None) ->
'FrameOrSeries'
 |      Select values at particular time of day (e.g., 9:30AM).
 |
 |      Parameters
 |      ----------
 |      time : datetime.time or str
 |      axis : {0 or 'index', 1 or 'columns'}, default 0
 |
 |      Returns
 |      -------
 |      Series or DataFrame
 |
 |      Raises
 |      ------
 |      TypeError
 |          If the index is not  a :class:`DatetimeIndex`
 |
 |      See Also
 |      --------
 |      between_time : Select values between particular times of the day.
 |      first : Select initial periods of time series based on a date offset.
 |      last : Select final periods of time series based on a date offset.
 |      DatetimeIndex.indexer_at_time : Get just the index locations for
 |          values at particular time of the day.
 |
 |      Examples
 |      --------
 |      >>> i = pd.date_range('2018-04-09', periods=4, freq='12H')
 |      >>> ts = pd.DataFrame({'A': [1, 2, 3, 4]}, index=i)
 |      >>> ts
 |                           A
 |      2018-04-09 00:00:00  1
 |      2018-04-09 12:00:00  2
 |      2018-04-10 00:00:00  3
 |      2018-04-10 12:00:00  4
 |
 |      >>> ts.at_time('12:00')
 |                           A
 |      2018-04-09 12:00:00  2
 |      2018-04-10 12:00:00  4
 |
 |  backfill = bfill(self: 'FrameOrSeries', axis: 'None | Axis' = None, inplace:
'bool_t' = False, limit: 'None | int' = None, downcast=None) -> 'FrameOrSeries |
None'
 |      Synonym for :meth:`DataFrame.fillna` with ``method='bfill'``.
 |
 |      Returns
 |      -------
 |      Series/DataFrame or None
 |          Object with missing values filled or None if ``inplace=True``.
 |
 |  between_time(self: 'FrameOrSeries', start_time, end_time, include_start: 'bool_t'
= True, include_end: 'bool_t' = True, axis=None) -> 'FrameOrSeries'
 |      Select values between particular times of the day (e.g., 9:00-9:30 AM).
 |
 |      By setting ``start_time`` to be later than ``end_time``,
 |      you can get the times that are *not* between the two times.
 |
 |      Parameters
 |      ----------
 |      start_time : datetime.time or str
 |          Initial time as a time filter limit.
 |      end_time : datetime.time or str
 |          End time as a time filter limit.
 |      include_start : bool, default True
 |          Whether the start time needs to be included in the result.
 |      include_end : bool, default True
 |          Whether the end time needs to be included in the result.
 |      axis : {0 or 'index', 1 or 'columns'}, default 0
 |          Determine range time on index or columns value.
 |
 |      Returns
 |      -------
 |      Series or DataFrame
 |          Data from the original object filtered to the specified dates range.
 |
 |      Raises
 |      ------
 |      TypeError
 |          If the index is not  a :class:`DatetimeIndex`
 |      
 |      See Also
 |      --------
 |      at_time : Select values at a particular time of the day.
 |      first : Select initial periods of time series based on a date offset.
 |      last : Select final periods of time series based on a date offset.
 |      DatetimeIndex.indexer_between_time : Get just the index locations for
 |          values between particular times of the day.
 |      
 |      Examples
 |      --------
 |      >>> i = pd.date_range('2018-04-09', periods=4, freq='1D20min')
 |      >>> ts = pd.DataFrame({'A': [1, 2, 3, 4]}, index=i)
 |      >>> ts
 |                           A
 |      2018-04-09 00:00:00  1
 |      2018-04-10 00:20:00  2
 |      2018-04-11 00:40:00  3
 |      2018-04-12 01:00:00  4
 |      
 |      >>> ts.between_time('0:15', '0:45')
 |                           A
 |      2018-04-10 00:20:00  2
 |      2018-04-11 00:40:00  3
 |      
 |      You get the times that are *not* between two times by setting
 |      ``start_time`` later than ``end_time``:
 |      
 |      >>> ts.between_time('0:45', '0:15')
 |                           A
 |      2018-04-09 00:00:00  1
 |      2018-04-12 01:00:00  4
 |  
 |  bool(self)
 |      Return the bool of a single element Series or DataFrame.
 |      
 |      This must be a boolean scalar value, either True or False. It will raise a
 |      ValueError if the Series or DataFrame does not have exactly 1 element, or
that
 |      element is not boolean (integer values 0 and 1 will also raise an exception).
 |      
 |      Returns
 |      -------
 |      bool
 |          The value in the Series or DataFrame.
 |      
 |      See Also
 |      --------
 |      Series.astype : Change the data type of a Series, including to boolean.
 |      DataFrame.astype : Change the data type of a DataFrame, including to boolean.
 |      numpy.bool_ : NumPy boolean data type, used by pandas for boolean values.
 |      
 |      Examples
 |      --------
 |      The method will only work for single element objects with a boolean value:
 |      
 |      >>> pd.Series([True]).bool()
 |      True
 |      >>> pd.Series([False]).bool()
 |      False
 |      
 |      >>> pd.DataFrame({'col': [True]}).bool()
 |      True
 |      >>> pd.DataFrame({'col': [False]}).bool()
 |      False
 |  
 |  convert_dtypes(self: 'FrameOrSeries', infer_objects: 'bool_t' = True,
convert_string: 'bool_t' = True, convert_integer: 'bool_t' = True, convert_boolean:
'bool_t' = True, convert_floating: 'bool_t' = True) -> 'FrameOrSeries'
 |      Convert columns to best possible dtypes using dtypes supporting ``pd.NA``.
 |      
 |      .. versionadded:: 1.0.0
 |      
 |      Parameters
 |      ----------
 |      infer_objects : bool, default True
 |          Whether object dtypes should be converted to the best possible types.
 |      convert_string : bool, default True
 |          Whether object dtypes should be converted to ``StringDtype()``.
 |      convert_integer : bool, default True
 |          Whether, if possible, conversion can be done to integer extension types.
 |      convert_boolean : bool, defaults True
 |          Whether object dtypes should be converted to ``BooleanDtypes()``.
 |      convert_floating : bool, defaults True
 |          Whether, if possible, conversion can be done to floating extension types.
 |          If `convert_integer` is also True, preference will be give to integer
 |          dtypes if the floats can be faithfully casted to integers.
 |
 |          .. versionadded:: 1.2.0
 |
 |      Returns
 |      -------
 |      Series or DataFrame
 |          Copy of input object with new dtype.
 |
 |      See Also
 |      --------
 |      infer_objects : Infer dtypes of objects.
 |      to_datetime : Convert argument to datetime.
 |      to_timedelta : Convert argument to timedelta.
 |      to_numeric : Convert argument to a numeric type.
 |
 |      Notes
 |      -----
 |      By default, ``convert_dtypes`` will attempt to convert a Series (or each
 |      Series in a DataFrame) to dtypes that support ``pd.NA``. By using the options
 |      ``convert_string``, ``convert_integer``, ``convert_boolean`` and
 |      ``convert_boolean``, it is possible to turn off individual conversions
 |      to ``StringDtype``, the integer extension types, ``BooleanDtype``
 |      or floating extension types, respectively.
 |
 |      For object-dtyped columns, if ``infer_objects`` is ``True``, use the
inference
 |      rules as during normal Series/DataFrame construction.  Then, if possible,
 |      convert to ``StringDtype``, ``BooleanDtype`` or an appropriate integer
 |      or floating extension type, otherwise leave as ``object``.
 |
 |      If the dtype is integer, convert to an appropriate integer extension type.
 |
 |      If the dtype is numeric, and consists of all integers, convert to an
 |      appropriate integer extension type. Otherwise, convert to an
 |      appropriate floating extension type.
 |
 |      .. versionchanged:: 1.2
 |          Starting with pandas 1.2, this method also converts float columns
 |          to the nullable floating extension type.
 |
 |      In the future, as new dtypes are added that support ``pd.NA``, the results
 |      of this method will change to support those new dtypes.
 |
 |      Examples
 |      --------
 |      >>> df = pd.DataFrame(
 |      ...     {
 |      ...         "a": pd.Series([1, 2, 3], dtype=np.dtype("int32")),
 |      ...         "b": pd.Series(["x", "y", "z"], dtype=np.dtype("O")),
 |      ...         "c": pd.Series([True, False, np.nan], dtype=np.dtype("O")),
 |      ...         "d": pd.Series(["h", "i", np.nan], dtype=np.dtype("O")),
 |      ...         "e": pd.Series([10, np.nan, 20], dtype=np.dtype("float")),
 |      ...         "f": pd.Series([np.nan, 100.5, 200], dtype=np.dtype("float")),
 |      ...     }
 |      ... )
 |
 |      Start with a DataFrame with default dtypes.
 |
 |      >>> df
 |         a  b      c    d     e      f
 |      0  1  x   True    h  10.0    NaN
 |      1  2  y  False    i   NaN  100.5
 |      2  3  z    NaN  NaN  20.0  200.0
 |
 |      >>> df.dtypes
 |      a      int32
 |      b     object
 |      c     object
 |      d     object
 |      e    float64
 |      f    float64
 |      dtype: object
 |
 |      Convert the DataFrame to use best possible dtypes.
 |
 |      >>> dfn = df.convert_dtypes()
 |      >>> dfn
 |         a  b      c     d     e      f
 |      0  1  x   True     h    10   <NA>
 |      1  2  y  False     i  <NA>  100.5
 |      2  3  z   <NA>  <NA>    20  200.0
 |
 |      >>> dfn.dtypes
 |      a      Int32
 |      b     string
 |      c    boolean
 |      d     string
 |      e      Int64
 |      f    Float64
 |      dtype: object
 |
 |      Start with a Series of strings and missing data represented by ``np.nan``.
 |
 |      >>> s = pd.Series(["a", "b", np.nan])
 |      >>> s
 |      0      a
 |      1      b
 |      2    NaN
 |      dtype: object
 |
 |      Obtain a Series with dtype ``StringDtype``.
 |
 |      >>> s.convert_dtypes()
 |      0       a
 |      1       b
 |      2    <NA>
 |      dtype: string
 |
 |  copy(self: 'FrameOrSeries', deep: 'bool_t' = True) -> 'FrameOrSeries'
 |      Make a copy of this object's indices and data.
 |
 |      When ``deep=True`` (default), a new object will be created with a
 |      copy of the calling object's data and indices. Modifications to
 |      the data or indices of the copy will not be reflected in the
 |      original object (see notes below).
 |
 |      When ``deep=False``, a new object will be created without copying
 |      the calling object's data or index (only references to the data
 |      and index are copied). Any changes to the data of the original
 |      will be reflected in the shallow copy (and vice versa).
 |
 |      Parameters
 |      ----------
 |      deep : bool, default True
 |          Make a deep copy, including a copy of the data and the indices.
 |          With ``deep=False`` neither the indices nor the data are copied.
 |
 |      Returns
 |      -------
 |      copy : Series or DataFrame
 |          Object type matches caller.
 |
 |      Notes
 |      -----
 |      When ``deep=True``, data is copied but actual Python objects
 |      will not be copied recursively, only the reference to the object.
 |      This is in contrast to `copy.deepcopy` in the Standard Library,
 |      which recursively copies object data (see examples below).
 |
 |      While ``Index`` objects are copied when ``deep=True``, the underlying
 |      numpy array is not copied for performance reasons. Since ``Index`` is
 |      immutable, the underlying data can be safely shared and a copy
 |      is not needed.
 |
 |      Examples
 |      --------
 |      >>> s = pd.Series([1, 2], index=["a", "b"])
 |      >>> s
 |      a    1
 |      b    2
 |      dtype: int64
 |
 |      >>> s_copy = s.copy()
 |      >>> s_copy
 |      a    1
 |      b    2
 |      dtype: int64
 |
 |      **Shallow copy versus default (deep) copy:**
 |
 |      >>> s = pd.Series([1, 2], index=["a", "b"])
 |      >>> deep = s.copy()
 |      >>> shallow = s.copy(deep=False)
 |
 |      Shallow copy shares data and index with original.
 |
 |      >>> s is shallow
 |      False
 |      >>> s.values is shallow.values and s.index is shallow.index
 |      True
 |
 |      Deep copy has own copy of data and index.
 |
 |      >>> s is deep
 |      False
 |      >>> s.values is deep.values or s.index is deep.index
 |      False
 |
 |      Updates to the data shared by shallow copy and original is reflected
 |      in both; deep copy remains unchanged.
 |
 |      >>> s[0] = 3
 |      >>> shallow[1] = 4
 |      >>> s
 |      a    3
 |      b    4
 |      dtype: int64
 |      >>> shallow
 |      a    3
 |      b    4
 |      dtype: int64
 |      >>> deep
 |      a    1
 |      b    2
 |      dtype: int64
 |
 |      Note that when copying an object containing Python objects, a deep copy
 |      will copy the data, but will not do so recursively. Updating a nested
 |      data object will be reflected in the deep copy.
 |
 |      >>> s = pd.Series([[1, 2], [3, 4]])
 |      >>> deep = s.copy()
 |      >>> s[0][0] = 10
 |      >>> s
 |      0    [10, 2]
 |      1     [3, 4]
 |      dtype: object
 |      >>> deep
 |      0    [10, 2]
 |      1     [3, 4]
 |      dtype: object
 |
 |  describe(self: 'FrameOrSeries', percentiles=None, include=None, exclude=None,
datetime_is_numeric=False) -> 'FrameOrSeries'
 |      Generate descriptive statistics.
 |
 |      Descriptive statistics include those that summarize the central
 |      tendency, dispersion and shape of a
 |      dataset's distribution, excluding ``NaN`` values.
 |
 |      Analyzes both numeric and object series, as well
 |      as ``DataFrame`` column sets of mixed data types. The output
 |      will vary depending on what is provided. Refer to the notes
 |      below for more detail.
 |
 |      Parameters
 |      ----------
 |      percentiles : list-like of numbers, optional
 |          The percentiles to include in the output. All should
 |          fall between 0 and 1. The default is
 |          ``[.25, .5, .75]``, which returns the 25th, 50th, and
 |          75th percentiles.
 |      include : 'all', list-like of dtypes or None (default), optional
 |          A white list of data types to include in the result. Ignored
 |          for ``Series``. Here are the options:
 |
 |          - 'all' : All columns of the input will be included in the output.
 |          - A list-like of dtypes : Limits the results to the
 |            provided data types.
 |            To limit the result to numeric types submit
 |            ``numpy.number``. To limit it instead to object columns submit
 |            the ``numpy.object`` data type. Strings
 |            can also be used in the style of
 |            ``select_dtypes`` (e.g. ``df.describe(include=['O'])``). To
 |            select pandas categorical columns, use ``'category'``
 |          - None (default) : The result will include all numeric columns.
 |      exclude : list-like of dtypes or None (default), optional,
 |          A black list of data types to omit from the result. Ignored
 |          for ``Series``. Here are the options:
 |
 |          - A list-like of dtypes : Excludes the provided data types
 |            from the result. To exclude numeric types submit
 |            ``numpy.number``. To exclude object columns submit the data
 |            type ``numpy.object``. Strings can also be used in the style of
 |            ``select_dtypes`` (e.g. ``df.describe(include=['O'])``). To
 |            exclude pandas categorical columns, use ``'category'``
 |          - None (default) : The result will exclude nothing.
 |      datetime_is_numeric : bool, default False
 |          Whether to treat datetime dtypes as numeric. This affects statistics
 |          calculated for the column. For DataFrame input, this also
 |          controls whether datetime columns are included by default.
 |
 |          .. versionadded:: 1.1.0
 |
 |      Returns
 |      -------
 |      Series or DataFrame
 |          Summary statistics of the Series or Dataframe provided.
 |
 |      See Also
 |      --------
 |      DataFrame.count: Count number of non-NA/null observations.
 |      DataFrame.max: Maximum of the values in the object.
 |      DataFrame.min: Minimum of the values in the object.
 |      DataFrame.mean: Mean of the values.
 |      DataFrame.std: Standard deviation of the observations.
 |      DataFrame.select_dtypes: Subset of a DataFrame including/excluding
 |          columns based on their dtype.
 |
 |      Notes
 |      -----
 |      For numeric data, the result's index will include ``count``,
 |      ``mean``, ``std``, ``min``, ``max`` as well as lower, ``50`` and
 |      upper percentiles. By default the lower percentile is ``25`` and the
 |      upper percentile is ``75``. The ``50`` percentile is the
 |      same as the median.
 |
 |      For object data (e.g. strings or timestamps), the result's index
 |      will include ``count``, ``unique``, ``top``, and ``freq``. The ``top``
 |      is the most common value. The ``freq`` is the most common value's
 |      frequency. Timestamps also include the ``first`` and ``last`` items.
 |
 |      If multiple object values have the highest count, then the
 |      ``count`` and ``top`` results will be arbitrarily chosen from
 |      among those with the highest count.
 |
 |      For mixed data types provided via a ``DataFrame``, the default is to
 |      return only an analysis of numeric columns. If the dataframe consists
 |      only of object and categorical data without any numeric columns, the
 |      default is to return an analysis of both the object and categorical
 |      columns. If ``include='all'`` is provided as an option, the result
 |      will include a union of attributes of each type.
 |
 |      The `include` and `exclude` parameters can be used to limit
 |      which columns in a ``DataFrame`` are analyzed for the output.
 |      The parameters are ignored when analyzing a ``Series``.
 |
 |      Examples
 |      --------
 |      Describing a numeric ``Series``.
 |
 |      >>> s = pd.Series([1, 2, 3])
 |      >>> s.describe()
 |      count    3.0
 |      mean     2.0
 |      std      1.0
 |      min      1.0
 |      25%      1.5
 |      50%      2.0
 |      75%      2.5
 |      max      3.0
 |      dtype: float64
 |
 |      Describing a categorical ``Series``.
 |
 |      >>> s = pd.Series(['a', 'a', 'b', 'c'])
 |      >>> s.describe()
 |      count     4
 |      unique    3
 |      top       a
 |      freq      2
 |      dtype: object
 |
 |      Describing a timestamp ``Series``.
 |
 |      >>> s = pd.Series([
 |      ...   np.datetime64("2000-01-01"),
 |      ...   np.datetime64("2010-01-01"),
 |      ...   np.datetime64("2010-01-01")
 |      ... ])
 |      >>> s.describe(datetime_is_numeric=True)
 |      count                      3
 |      mean     2006-09-01 08:00:00
 |      min      2000-01-01 00:00:00
 |      25%      2004-12-31 12:00:00
 |      50%      2010-01-01 00:00:00
 |      75%      2010-01-01 00:00:00
 |      max      2010-01-01 00:00:00
 |      dtype: object
 |
 |      Describing a ``DataFrame``. By default only numeric fields
 |      are returned.
 |
 |      >>> df = pd.DataFrame({'categorical': pd.Categorical(['d','e','f']),
 |      ...                    'numeric': [1, 2, 3],
 |      ...                    'object': ['a', 'b', 'c']
 |      ...                   })
 |      >>> df.describe()
 |             numeric
 |      count      3.0
 |      mean       2.0
 |      std        1.0
 |      min        1.0
 |      25%        1.5
 |      50%        2.0
 |      75%        2.5
 |      max        3.0
 |
 |      Describing all columns of a ``DataFrame`` regardless of data type.
 |
 |      >>> df.describe(include='all')  # doctest: +SKIP
 |             categorical  numeric object
 |      count            3      3.0      3
 |      unique           3      NaN      3
 |      top              f      NaN      a
 |      freq             1      NaN      1
 |      mean           NaN      2.0    NaN
 |      std            NaN      1.0    NaN
 |      min            NaN      1.0    NaN
 |      25%            NaN      1.5    NaN
 |      50%            NaN      2.0    NaN
 |      75%            NaN      2.5    NaN
 |      max            NaN      3.0    NaN
 |
 |      Describing a column from a ``DataFrame`` by accessing it as
 |      an attribute.
 |
 |      >>> df.numeric.describe()
 |      count    3.0
 |      mean     2.0
 |      std      1.0
 |      min      1.0
 |      25%      1.5
 |      50%      2.0
 |      75%      2.5
 |      max      3.0
 |      Name: numeric, dtype: float64
 |
 |      Including only numeric columns in a ``DataFrame`` description.
 |
 |      >>> df.describe(include=[np.number])
 |             numeric
 |      count      3.0
 |      mean       2.0
 |      std        1.0
 |      min        1.0
 |      25%        1.5
 |      50%        2.0
 |      75%        2.5
 |      max        3.0
 |
 |      Including only string columns in a ``DataFrame`` description.
 |
 |      >>> df.describe(include=[object])  # doctest: +SKIP
 |             object
 |      count       3
 |      unique      3
 |      top         a
 |      freq        1
 |
 |      Including only categorical columns from a ``DataFrame`` description.
 |
 |      >>> df.describe(include=['category'])
 |             categorical
 |      count            3
 |      unique           3
 |      top              d
 |      freq             1
 |
 |      Excluding numeric columns from a ``DataFrame`` description.
 |
 |      >>> df.describe(exclude=[np.number])  # doctest: +SKIP
 |             categorical object
 |      count            3      3
 |      unique           3      3
 |      top              f      a
 |      freq             1      1
 |      
 |      Excluding object columns from a ``DataFrame`` description.
 |      
 |      >>> df.describe(exclude=[object])  # doctest: +SKIP
 |             categorical  numeric
 |      count            3      3.0
 |      unique           3      NaN
 |      top              f      NaN
 |      freq             1      NaN
 |      mean           NaN      2.0
 |      std            NaN      1.0
 |      min            NaN      1.0
 |      25%            NaN      1.5
 |      50%            NaN      2.0
 |      75%            NaN      2.5
 |      max            NaN      3.0
 |  
 |  droplevel(self: 'FrameOrSeries', level, axis=0) -> 'FrameOrSeries'
 |      Return Series/DataFrame with requested index / column level(s) removed.
 |      
 |      Parameters
 |      ----------
 |      level : int, str, or list-like
 |          If a string is given, must be the name of a level
 |          If list-like, elements must be names or positional indexes
 |          of levels.
 |      
 |      axis : {0 or 'index', 1 or 'columns'}, default 0
 |          Axis along which the level(s) is removed:
 |      
 |          * 0 or 'index': remove level(s) in column.
 |          * 1 or 'columns': remove level(s) in row.
 |      
 |      Returns
 |      -------
 |      Series/DataFrame
 |          Series/DataFrame with requested index / column level(s) removed.
 |      
 |      Examples
 |      --------
 |      >>> df = pd.DataFrame([
 |      ...     [1, 2, 3, 4],
 |      ...     [5, 6, 7, 8],
 |      ...     [9, 10, 11, 12]
 |      ... ]).set_index([0, 1]).rename_axis(['a', 'b'])
 |      
 |      >>> df.columns = pd.MultiIndex.from_tuples([
 |      ...     ('c', 'e'), ('d', 'f')
 |      ... ], names=['level_1', 'level_2'])
 |      
 |      >>> df
 |      level_1   c   d
 |      level_2   e   f
 |      a b
 |      1 2      3   4
 |      5 6      7   8
 |      9 10    11  12
 |      
 |      >>> df.droplevel('a')
 |      level_1   c   d
 |      level_2   e   f
 |      b
 |      2        3   4
 |      6        7   8
 |      10      11  12
 |      
 |      >>> df.droplevel('level_2', axis=1)
 |      level_1   c   d
 |      a b
 |      1 2      3   4
 |      5 6      7   8
 |      9 10    11  12
 |  
 |  equals(self, other: 'object') -> 'bool_t'
 |      Test whether two objects contain the same elements.
 |      
 |      This function allows two Series or DataFrames to be compared against
 |      each other to see if they have the same shape and elements. NaNs in
 |      the same location are considered equal.
 |      
 |      The row/column index do not need to have the same type, as long
 |      as the values are considered equal. Corresponding columns must be of
 |      the same dtype.
 |      
 |      Parameters
 |      ----------
 |      other : Series or DataFrame
 |          The other Series or DataFrame to be compared with the first.
 |      
 |      Returns
 |      -------
 |      bool
 |          True if all elements are the same in both objects, False
 |          otherwise.
 |      
 |      See Also
 |      --------
 |      Series.eq : Compare two Series objects of the same length
 |          and return a Series where each element is True if the element
 |          in each Series is equal, False otherwise.
 |      DataFrame.eq : Compare two DataFrame objects of the same shape and
 |          return a DataFrame where each element is True if the respective
 |          element in each DataFrame is equal, False otherwise.
 |      testing.assert_series_equal : Raises an AssertionError if left and
 |          right are not equal. Provides an easy interface to ignore
 |          inequality in dtypes, indexes and precision among others.
 |      testing.assert_frame_equal : Like assert_series_equal, but targets
 |          DataFrames.
 |      numpy.array_equal : Return True if two arrays have the same shape
 |          and elements, False otherwise.
 |      
 |      Examples
 |      --------
 |      >>> df = pd.DataFrame({1: [10], 2: [20]})
 |      >>> df
 |          1   2
 |      0  10  20
 |      
 |      DataFrames df and exactly_equal have the same types and values for
 |      their elements and column labels, which will return True.
 |      
 |      >>> exactly_equal = pd.DataFrame({1: [10], 2: [20]})
 |      >>> exactly_equal
 |          1   2
 |      0  10  20
 |      >>> df.equals(exactly_equal)
 |      True
 |      
 |      DataFrames df and different_column_type have the same element
 |      types and values, but have different types for the column labels,
 |      which will still return True.
 |      
 |      >>> different_column_type = pd.DataFrame({1.0: [10], 2.0: [20]})
 |      >>> different_column_type
 |         1.0  2.0
 |      0   10   20
 |      >>> df.equals(different_column_type)
 |      True
 |      
 |      DataFrames df and different_data_type have different types for the
 |      same values for their elements, and will return False even though
 |      their column labels are the same values and types.
 |      
 |      >>> different_data_type = pd.DataFrame({1: [10.0], 2: [20.0]})
 |      >>> different_data_type
 |            1     2
 |      0  10.0  20.0
 |      >>> df.equals(different_data_type)
 |      False
 |  
 |  ewm(self, com: 'float | None' = None, span: 'float | None' = None, halflife: 
'float | TimedeltaConvertibleTypes | None' = None, alpha: 'float | None' = None, 
min_periods: 'int | None' = 0, adjust: 'bool_t' = True, ignore_na: 'bool_t' = False, 
axis: 'Axis' = 0, times: 'str | np.ndarray | FrameOrSeries | None' = None) -> 
'ExponentialMovingWindow'
 |      Provide exponential weighted (EW) functions.
 |      
 |      Available EW functions: ``mean()``, ``var()``, ``std()``, ``corr()``, 
``cov()``.
 |      
 |      Exactly one parameter: ``com``, ``span``, ``halflife``, or ``alpha`` must be
 |      provided.
 |      
 |      Parameters
 |      ----------
 |      com : float, optional
 |          Specify decay in terms of center of mass,
 |          :math:`\alpha = 1 / (1 + com)`, for :math:`com \geq 0`.
 |      span : float, optional
 |          Specify decay in terms of span,
 |          :math:`\alpha = 2 / (span + 1)`, for :math:`span \geq 1`.
 |      halflife : float, str, timedelta, optional
 |          Specify decay in terms of half-life,
 |          :math:`\alpha = 1 - \exp\left(-\ln(2) / halflife\right)`, for
 |          :math:`halflife > 0`.
 |
 |          If ``times`` is specified, the time unit (str or timedelta) over which an
 |          observation decays to half its value. Only applicable to ``mean()``
 |          and halflife value will not apply to the other functions.
 |
 |          .. versionadded:: 1.1.0
 |
 |      alpha : float, optional
 |          Specify smoothing factor :math:`\alpha` directly,
 |          :math:`0 < \alpha \leq 1`.
 |      min_periods : int, default 0
 |          Minimum number of observations in window required to have a value
 |          (otherwise result is NA).
 |      adjust : bool, default True
 |          Divide by decaying adjustment factor in beginning periods to account
 |          for imbalance in relative weightings (viewing EWMA as a moving average).
 |
 |          - When ``adjust=True`` (default), the EW function is calculated using
weights
 |            :math:`w_i = (1 - \alpha)^i`. For example, the EW moving average of the
series
 |            [:math:`x_0, x_1, ..., x_t`] would be:
 |
 |          .. math::
 |              y_t = \frac{x_t + (1 - \alpha)x_{t-1} + (1 - \alpha)^2 x_{t-2} + ...
+ (1 -
 |              \alpha)^t x_0}{1 + (1 - \alpha) + (1 - \alpha)^2 + ... + (1 -
\alpha)^t}
 |
 |          - When ``adjust=False``, the exponentially weighted function is
calculated
 |            recursively:
 |
 |          .. math::
 |              \begin{split}
 |                  y_0 &= x_0\\
 |                  y_t &= (1 - \alpha) y_{t-1} + \alpha x_t,
 |              \end{split}
 |      ignore_na : bool, default False
 |          Ignore missing values when calculating weights; specify ``True`` to
reproduce
 |          pre-0.15.0 behavior.
 |
 |          - When ``ignore_na=False`` (default), weights are based on absolute
positions.
 |            For example, the weights of :math:`x_0` and :math:`x_2` used in
calculating
 |            the final weighted average of [:math:`x_0`, None, :math:`x_2`] are
 |            :math:`(1-\alpha)^2` and :math:`1` if ``adjust=True``, and
 |            :math:`(1-\alpha)^2` and :math:`\alpha` if ``adjust=False``.
 |
 |          - When ``ignore_na=True`` (reproducing pre-0.15.0 behavior), weights are
based
 |            on relative positions. For example, the weights of :math:`x_0` and
:math:`x_2`
 |            used in calculating the final weighted average of
 |            [:math:`x_0`, None, :math:`x_2`] are :math:`1-\alpha` and :math:`1` if
 |            ``adjust=True``, and :math:`1-\alpha` and :math:`\alpha` if
``adjust=False``.
 |      axis : {0, 1}, default 0
 |          The axis to use. The value 0 identifies the rows, and 1
 |          identifies the columns.
 |      times : str, np.ndarray, Series, default None
 |
 |          .. versionadded:: 1.1.0
 |
 |          Times corresponding to the observations. Must be monotonically increasing
and
 |          ``datetime64[ns]`` dtype.
 |
 |          If str, the name of the column in the DataFrame representing the times.
 |
 |          If 1-D array like, a sequence with the same shape as the observations.
 |
 |          Only applicable to ``mean()``.
 |
 |      Returns
 |      -------
 |      DataFrame
 |          A Window sub-classed for the particular operation.
 |
 |
 |      See Also
 |      --------
 |      rolling : Provides rolling window calculations.
 |      expanding : Provides expanding transformations.
 |
 |      Notes
 |      -----
 |
 |      More details can be found at:
 |      :ref:`Exponentially weighted windows <window.exponentially_weighted>`.
 |
 |      Examples
 |      --------
 |      >>> df = pd.DataFrame({'B': [0, 1, 2, np.nan, 4]})
 |      >>> df
 |           B
 |      0  0.0
 |      1  1.0
 |      2  2.0
 |      3  NaN
 |      4  4.0
 |
 |      >>> df.ewm(com=0.5).mean()
 |                B
 |      0  0.000000
 |      1  0.750000
 |      2  1.615385
 |      3  1.615385
 |      4  3.670213
 |
 |      Specifying ``times`` with a timedelta ``halflife`` when computing mean.
 |
 |      >>> times = ['2020-01-01', '2020-01-03', '2020-01-10', '2020-01-15', '2020-
01-17']
 |      >>> df.ewm(halflife='4 days', times=pd.DatetimeIndex(times)).mean()
 |                B
 |      0  0.000000
 |      1  0.585786
 |      2  1.523889
 |      3  1.523889
 |      4  3.233686
 |
 |  expanding(self, min_periods: 'int' = 1, center: 'bool_t | None' = None, axis:
'Axis' = 0, method: 'str' = 'single') -> 'Expanding'
 |      Provide expanding transformations.
 |
 |      Parameters
 |      ----------
 |      min_periods : int, default 1
 |          Minimum number of observations in window required to have a value
 |          (otherwise result is NA).
 |      center : bool, default False
 |          Set the labels at the center of the window.
 |      axis : int or str, default 0
 |      method : str {'single', 'table'}, default 'single'
 |          Execute the rolling operation per single column or row (``'single'``)
 |          or over the entire object (``'table'``).
 |
 |          This argument is only implemented when specifying ``engine='numba'``
 |          in the method call.
 |
 |          .. versionadded:: 1.3.0
 |
 |      Returns
 |      -------
 |      a Window sub-classed for the particular operation
 |
 |      See Also
 |      --------
 |      rolling : Provides rolling window calculations.
 |      ewm : Provides exponential weighted functions.
 |
 |      Notes
 |      -----
 |      By default, the result is set to the right edge of the window. This can be
 |      changed to the center of the window by setting ``center=True``.
 |
 |      Examples
 |      --------
 |      >>> df = pd.DataFrame({"B": [0, 1, 2, np.nan, 4]})
 |      >>> df
 |           B
 |      0  0.0
 |      1  1.0
 |      2  2.0
 |      3  NaN
 |      4  4.0
 |
 |      >>> df.expanding(2).sum()
 |           B
 |      0  NaN
 |      1  1.0
 |      2  3.0
 |      3  3.0
 |      4  7.0
 |
 |  filter(self: 'FrameOrSeries', items=None, like: 'str | None' = None, regex: 'str
| None' = None, axis=None) -> 'FrameOrSeries'
 |      Subset the dataframe rows or columns according to the specified index labels.
 |
 |      Note that this routine does not filter a dataframe on its
 |      contents. The filter is applied to the labels of the index.
 |
 |      Parameters
 |      ----------
 |      items : list-like
 |          Keep labels from axis which are in items.
 |      like : str
 |          Keep labels from axis for which "like in label == True".
 |      regex : str (regular expression)
 |          Keep labels from axis for which re.search(regex, label) == True.
 |      axis : {0 or ‘index’, 1 or ‘columns’, None}, default None
 |          The axis to filter on, expressed either as an index (int)
 |          or axis name (str). By default this is the info axis,
 |          'index' for Series, 'columns' for DataFrame.
 |
 |      Returns
 |      -------
 |      same type as input object
 |
 |      See Also
 |      --------
 |      DataFrame.loc : Access a group of rows and columns
 |          by label(s) or a boolean array.
 |
 |      Notes
 |      -----
 |      The ``items``, ``like``, and ``regex`` parameters are
 |      enforced to be mutually exclusive.
 |
 |      ``axis`` defaults to the info axis that is used when indexing
 |      with ``[]``.
 |
 |      Examples
 |      --------
 |      >>> df = pd.DataFrame(np.array(([1, 2, 3], [4, 5, 6])),
 |      ...                   index=['mouse', 'rabbit'],
 |      ...                   columns=['one', 'two', 'three'])
 |      >>> df
 |              one  two  three
 |      mouse     1    2      3
 |      rabbit    4    5      6
 |
 |      >>> # select columns by name
 |      >>> df.filter(items=['one', 'three'])
 |               one  three
 |      mouse     1      3
 |      rabbit    4      6
 |
 |      >>> # select columns by regular expression
 |      >>> df.filter(regex='e$', axis=1)
 |               one  three
 |      mouse     1      3
 |      rabbit    4      6
 |
 |      >>> # select rows containing 'bbi'
 |      >>> df.filter(like='bbi', axis=0)
 |               one  two  three
 |      rabbit    4    5      6
 |
 |  first(self: 'FrameOrSeries', offset) -> 'FrameOrSeries'
 |      Select initial periods of time series data based on a date offset.
 |
 |      When having a DataFrame with dates as index, this function can
 |      select the first few rows based on a date offset.
 |
 |      Parameters
 |      ----------
 |      offset : str, DateOffset or dateutil.relativedelta
 |          The offset length of the data that will be selected. For instance,
 |          '1M' will display all the rows having their index within the first month.
 |
 |      Returns
 |      -------
 |      Series or DataFrame
 |          A subset of the caller.
 |
 |      Raises
 |      ------
 |      TypeError
 |          If the index is not  a :class:`DatetimeIndex`
 |
 |      See Also
 |      --------
 |      last : Select final periods of time series based on a date offset.
 |      at_time : Select values at a particular time of the day.
 |      between_time : Select values between particular times of the day.
 |
 |      Examples
 |      --------
 |      >>> i = pd.date_range('2018-04-09', periods=4, freq='2D')
 |      >>> ts = pd.DataFrame({'A': [1, 2, 3, 4]}, index=i)
 |      >>> ts
 |                  A
 |      2018-04-09  1
 |      2018-04-11  2
 |      2018-04-13  3
 |      2018-04-15  4
 |
 |      Get the rows for the first 3 days:
 |
 |      >>> ts.first('3D')
 |                  A
 |      2018-04-09  1
 |      2018-04-11  2
 |
 |      Notice the data for 3 first calendar days were returned, not the first
 |      3 days observed in the dataset, and therefore data for 2018-04-13 was
 |      not returned.
 |
 |  first_valid_index(self) -> 'Hashable | None'
 |      Return index for first non-NA value or None, if no NA value is found.
 |
 |      Returns
 |      -------
 |      scalar : type of index
 |
 |      Notes
 |      -----
 |      If all elements are non-NA/null, returns None.
 |      Also returns None for empty Series/DataFrame.
 |
 |  get(self, key, default=None)
 |      Get item from object for given key (ex: DataFrame column).
 |
 |      Returns default value if not found.
 |
 |      Parameters
 |      ----------
 |      key : object
 |
 |      Returns
 |      -------
 |      value : same type as items contained in object
 |
 |  head(self: 'FrameOrSeries', n: 'int' = 5) -> 'FrameOrSeries'
 |      Return the first `n` rows.
 |
 |      This function returns the first `n` rows for the object based
 |      on position. It is useful for quickly testing if your object
 |      has the right type of data in it.
 |
 |      For negative values of `n`, this function returns all rows except
 |      the last `n` rows, equivalent to ``df[:-n]``.
 |
 |      Parameters
 |      ----------
 |      n : int, default 5
 |          Number of rows to select.
 |
 |      Returns
 |      -------
 |      same type as caller
 |          The first `n` rows of the caller object.
 |
 |      See Also
 |      --------
 |      DataFrame.tail: Returns the last `n` rows.
 |
 |      Examples
```

```
 |      --------
 |      >>> df = pd.DataFrame({'animal': ['alligator', 'bee', 'falcon', 'lion',
 |      ...                    'monkey', 'parrot', 'shark', 'whale', 'zebra']})
 |      >>> df
 |            animal
 |      0  alligator
 |      1        bee
 |      2     falcon
 |      3       lion
 |      4     monkey
 |      5     parrot
 |      6      shark
 |      7      whale
 |      8      zebra
 |      
 |      Viewing the first 5 lines
 |      
 |      >>> df.head()
 |            animal
 |      0  alligator
 |      1        bee
 |      2     falcon
 |      3       lion
 |      4     monkey
 |      
 |      Viewing the first `n` lines (three in this case)
 |      
 |      >>> df.head(3)
 |            animal
 |      0  alligator
 |      1        bee
 |      2     falcon
 |      
 |      For negative values of `n`
 |      
 |      >>> df.head(-3)
 |            animal
 |      0  alligator
 |      1        bee
 |      2     falcon
 |      3       lion
 |      4     monkey
 |      5     parrot
 |  
 |  infer_objects(self: 'FrameOrSeries') -> 'FrameOrSeries'
 |      Attempt to infer better dtypes for object columns.
 |      
 |      Attempts soft conversion of object-dtyped
 |      columns, leaving non-object and unconvertible
 |      columns unchanged. The inference rules are the
 |      same as during normal Series/DataFrame construction.
 |      
 |      Returns
 |      -------
 |      converted : same type as input object
 |      
 |      See Also
 |      --------
 |      to_datetime : Convert argument to datetime.
 |      to_timedelta : Convert argument to timedelta.
 |      to_numeric : Convert argument to numeric type.
 |      convert_dtypes : Convert argument to best possible dtype.
 |      
 |      Examples
 |      --------
 |      >>> df = pd.DataFrame({"A": ["a", 1, 2, 3]})
 |      >>> df = df.iloc[1:]
 |      >>> df
 |         A
 |      1  1
 |      2  2
 |      3  3
 |      
 |      >>> df.dtypes
 |      A    object
 |      dtype: object
 |      
 |      >>> df.infer_objects().dtypes
 |      A    int64
 |      dtype: object
 |  
 |  keys(self)
 |      Get the 'info axis' (see Indexing for more).
 |      
 |      This is index for Series, columns for DataFrame.
 |      
 |      Returns
 |      -------
 |      Index
 |          Info axis.
 |  
 |  last(self: 'FrameOrSeries', offset) -> 'FrameOrSeries'
 |      Select final periods of time series data based on a date offset.
 |      
 |      For a DataFrame with a sorted DatetimeIndex, this function
 |      selects the last few rows based on a date offset.
 |      
 |      Parameters
 |      ----------
 |      offset : str, DateOffset, dateutil.relativedelta
 |          The offset length of the data that will be selected. For instance,
 |          '3D' will display all the rows having their index within the last 3 days.
 |      
 |      Returns
 |      -------
 |      Series or DataFrame
 |          A subset of the caller.
 |      
 |      Raises
 |      ------
 |      TypeError
 |          If the index is not  a :class:`DatetimeIndex`
 |      
 |      See Also
 |      --------
 |      first : Select initial periods of time series based on a date offset.
 |      at_time : Select values at a particular time of the day.
 |      between_time : Select values between particular times of the day.
 |      
 |      Examples
 |      --------
 |      >>> i = pd.date_range('2018-04-09', periods=4, freq='2D')
 |      >>> ts = pd.DataFrame({'A': [1, 2, 3, 4]}, index=i)
 |      >>> ts
 |                  A
 |      2018-04-09  1
 |      2018-04-11  2
 |      2018-04-13  3
 |      2018-04-15  4
 |      
 |      Get the rows for the last 3 days:
 |      
 |      >>> ts.last('3D')
 |                  A
 |      2018-04-13  3
 |      2018-04-15  4
 |      
 |      Notice the data for 3 last calendar days were returned, not the last
 |      3 observed days in the dataset, and therefore data for 2018-04-11 was
 |      not returned.
 |  
 |  last_valid_index(self) -> 'Hashable | None'
 |      Return index for last non-NA value or None, if no NA value is found.
 |      
 |      Returns
 |      -------
 |      scalar : type of index
 |      
 |      Notes
 |      -----
 |      If all elements are non-NA/null, returns None.
 |      Also returns None for empty Series/DataFrame.
 |  
 |  pad = ffill(self: 'FrameOrSeries', axis: 'None | Axis' = None, inplace: 'bool_t'
= False, limit: 'None | int' = None, downcast=None) -> 'FrameOrSeries | None'
 |      Synonym for :meth:`DataFrame.fillna` with ``method='ffill'``.
 |      
 |      Returns
 |      -------
 |      Series/DataFrame or None
 |          Object with missing values filled or None if ``inplace=True``.
 |  
 |  pct_change(self: 'FrameOrSeries', periods=1, fill_method='pad', limit=None,
freq=None, **kwargs) -> 'FrameOrSeries'
 |      Percentage change between the current and a prior element.
 |      
 |      Computes the percentage change from the immediately previous row by
 |      default. This is useful in comparing the percentage of change in a time
 |      series of elements.
 |      
 |      Parameters
 |      ----------
 |      periods : int, default 1
 |          Periods to shift for forming percent change.
 |      fill_method : str, default 'pad'
 |          How to handle NAs before computing percent changes.
 |      limit : int, default None
 |          The number of consecutive NAs to fill before stopping.
 |      freq : DateOffset, timedelta, or str, optional
 |          Increment to use from time series API (e.g. 'M' or BDay()).
 |      **kwargs
 |          Additional keyword arguments are passed into
 |          `DataFrame.shift` or `Series.shift`.
 |
 |      Returns
 |      -------
 |      chg : Series or DataFrame
 |          The same type as the calling object.
 |
 |      See Also
 |      --------
 |      Series.diff : Compute the difference of two elements in a Series.
 |      DataFrame.diff : Compute the difference of two elements in a DataFrame.
 |      Series.shift : Shift the index by some number of periods.
 |      DataFrame.shift : Shift the index by some number of periods.
 |
 |      Examples
 |      --------
 |      **Series**
 |
 |      >>> s = pd.Series([90, 91, 85])
 |      >>> s
 |      0    90
 |      1    91
 |      2    85
 |      dtype: int64
 |
 |      >>> s.pct_change()
 |      0         NaN
 |      1    0.011111
 |      2   -0.065934
 |      dtype: float64
 |
 |      >>> s.pct_change(periods=2)
 |      0         NaN
 |      1         NaN
 |      2   -0.055556
 |      dtype: float64
 |
 |      See the percentage change in a Series where filling NAs with last
 |      valid observation forward to next valid.
 |
 |      >>> s = pd.Series([90, 91, None, 85])
 |      >>> s
 |      0    90.0
 |      1    91.0
 |      2     NaN
 |      3    85.0
 |      dtype: float64
 |
 |      >>> s.pct_change(fill_method='ffill')
 |      0         NaN
 |      1    0.011111
 |      2    0.000000
 |      3   -0.065934
 |      dtype: float64
 |
 |      **DataFrame**
 |
 |      Percentage change in French franc, Deutsche Mark, and Italian lira from
 |      1980-01-01 to 1980-03-01.
 |
 |      >>> df = pd.DataFrame({
 |      ...     'FR': [4.0405, 4.0963, 4.3149],
 |      ...     'GR': [1.7246, 1.7482, 1.8519],
 |      ...     'IT': [804.74, 810.01, 860.13]},
 |      ...     index=['1980-01-01', '1980-02-01', '1980-03-01'])
 |      >>> df
 |                      FR      GR      IT
 |      1980-01-01  4.0405  1.7246  804.74
 |      1980-02-01  4.0963  1.7482  810.01
 |      1980-03-01  4.3149  1.8519  860.13
 |
 |      >>> df.pct_change()
 |                        FR        GR        IT
 |      1980-01-01       NaN       NaN       NaN
 |      1980-02-01  0.013810  0.013684  0.006549
 |      1980-03-01  0.053365  0.059318  0.061876
 |
 |      Percentage of change in GOOG and APPL stock volume. Shows computing
 |      the percentage change between columns.
```

```
 |
 |      >>> df = pd.DataFrame({
 |      ...     '2016': [1769950, 30586265],
 |      ...     '2015': [1500923, 40912316],
 |      ...     '2014': [1371819, 41403351]},
 |      ...     index=['GOOG', 'APPL'])
 |      >>> df
 |                2016      2015      2014
 |      GOOG   1769950   1500923   1371819
 |      APPL  30586265  40912316  41403351
 |
 |      >>> df.pct_change(axis='columns', periods=-1)
 |                2016      2015  2014
 |      GOOG  0.179241  0.094112   NaN
 |      APPL -0.252395 -0.011860   NaN
 |
 |  pipe(self, func: 'Callable[..., T] | tuple[Callable[..., T], str]', *args,
**kwargs) -> 'T'
 |      Apply func(self, \*args, \*\*kwargs).
 |
 |      Parameters
 |      ----------
 |      func : function
 |          Function to apply to the Series/DataFrame.
 |          ``args``, and ``kwargs`` are passed into ``func``.
 |          Alternatively a ``(callable, data_keyword)`` tuple where
 |          ``data_keyword`` is a string indicating the keyword of
 |          ``callable`` that expects the Series/DataFrame.
 |      args : iterable, optional
 |          Positional arguments passed into ``func``.
 |      kwargs : mapping, optional
 |          A dictionary of keyword arguments passed into ``func``.
 |
 |      Returns
 |      -------
 |      object : the return type of ``func``.
 |
 |      See Also
 |      --------
 |      DataFrame.apply : Apply a function along input axis of DataFrame.
 |      DataFrame.applymap : Apply a function elementwise on a whole DataFrame.
 |      Series.map : Apply a mapping correspondence on a
 |          :class:`~pandas.Series`.
 |
 |      Notes
 |      -----
 |      Use ``.pipe`` when chaining together functions that expect
 |      Series, DataFrames or GroupBy objects. Instead of writing
 |
 |      >>> func(g(h(df), arg1=a), arg2=b, arg3=c)  # doctest: +SKIP
 |
 |      You can write
 |
 |      >>> (df.pipe(h)
 |      ...    .pipe(g, arg1=a)
 |      ...    .pipe(func, arg2=b, arg3=c)
 |      ... )  # doctest: +SKIP
 |
 |      If you have a function that takes the data as (say) the second
 |      argument, pass a tuple indicating which keyword expects the
 |      data. For example, suppose ``f`` takes its data as ``arg2``:
 |
 |      >>> (df.pipe(h)
 |      ...    .pipe(g, arg1=a)
 |      ...    .pipe((func, 'arg2'), arg1=a, arg3=c)
 |      ...  )  # doctest: +SKIP
 |
 |  rank(self: 'FrameOrSeries', axis=0, method: 'str' = 'average', numeric_only:
'bool_t | None' = None, na_option: 'str' = 'keep', ascending: 'bool_t' = True, pct:
'bool_t' = False) -> 'FrameOrSeries'
 |      Compute numerical data ranks (1 through n) along axis.
 |
 |      By default, equal values are assigned a rank that is the average of the
 |      ranks of those values.
 |
 |      Parameters
 |      ----------
 |      axis : {0 or 'index', 1 or 'columns'}, default 0
 |          Index to direct ranking.
 |      method : {'average', 'min', 'max', 'first', 'dense'}, default 'average'
 |          How to rank the group of records that have the same value (i.e. ties):
 |
 |          * average: average rank of the group
 |          * min: lowest rank in the group
 |          * max: highest rank in the group
 |          * first: ranks assigned in order they appear in the array
 |          * dense: like 'min', but rank always increases by 1 between groups.
 |
 |      numeric_only : bool, optional
 |          For DataFrame objects, rank only numeric columns if set to True.
 |      na_option : {'keep', 'top', 'bottom'}, default 'keep'
 |          How to rank NaN values:
 |
 |          * keep: assign NaN rank to NaN values
 |          * top: assign lowest rank to NaN values
 |          * bottom: assign highest rank to NaN values
 |
 |      ascending : bool, default True
 |          Whether or not the elements should be ranked in ascending order.
 |      pct : bool, default False
 |          Whether or not to display the returned rankings in percentile
 |          form.
 |
 |      Returns
 |      -------
 |      same type as caller
 |          Return a Series or DataFrame with data ranks as values.
 |
 |      See Also
 |      --------
 |      core.groupby.GroupBy.rank : Rank of values within each group.
 |
 |      Examples
 |      --------
 |      >>> df = pd.DataFrame(data={'Animal': ['cat', 'penguin', 'dog',
 |      ...                                    'spider', 'snake'],
 |      ...                         'Number_legs': [4, 2, 4, 8, np.nan]})
 |      >>> df
 |          Animal  Number_legs
 |      0      cat          4.0
 |      1  penguin          2.0
 |      2      dog          4.0
 |      3   spider          8.0
 |      4    snake          NaN
 |
 |      The following example shows how the method behaves with the above
 |      parameters:
 |
 |      * default_rank: this is the default behaviour obtained without using
 |        any parameter.
 |      * max_rank: setting ``method = 'max'`` the records that have the
 |        same values are ranked using the highest rank (e.g.: since 'cat'
 |        and 'dog' are both in the 2nd and 3rd position, rank 3 is assigned.)
 |      * NA_bottom: choosing ``na_option = 'bottom'``, if there are records
 |        with NaN values they are placed at the bottom of the ranking.
 |      * pct_rank: when setting ``pct = True``, the ranking is expressed as
 |        percentile rank.
 |
 |      >>> df['default_rank'] = df['Number_legs'].rank()
 |      >>> df['max_rank'] = df['Number_legs'].rank(method='max')
 |      >>> df['NA_bottom'] = df['Number_legs'].rank(na_option='bottom')
 |      >>> df['pct_rank'] = df['Number_legs'].rank(pct=True)
 |      >>> df
 |          Animal  Number_legs  default_rank  max_rank  NA_bottom  pct_rank
 |      0      cat          4.0           2.5       3.0        2.5     0.625
 |      1  penguin          2.0           1.0       1.0        1.0     0.250
 |      2      dog          4.0           2.5       3.0        2.5     0.625
 |      3   spider          8.0           4.0       4.0        4.0     1.000
 |      4    snake          NaN           NaN       NaN        5.0       NaN
 |
 |  reindex_like(self: 'FrameOrSeries', other, method: 'str | None' = None, copy:
'bool_t' = True, limit=None, tolerance=None) -> 'FrameOrSeries'
 |      Return an object with matching indices as other object.
 |
 |      Conform the object to the same index on all axes. Optional
 |      filling logic, placing NaN in locations having no value
 |      in the previous index. A new object is produced unless the
 |      new index is equivalent to the current one and copy=False.
 |
 |      Parameters
 |      ----------
 |      other : Object of the same data type
 |          Its row and column indices are used to define the new indices
 |          of this object.
 |      method : {None, 'backfill'/'bfill', 'pad'/'ffill', 'nearest'}
 |          Method to use for filling holes in reindexed DataFrame.
 |          Please note: this is only applicable to DataFrames/Series with a
 |          monotonically increasing/decreasing index.
 |
 |          * None (default): don't fill gaps
 |          * pad / ffill: propagate last valid observation forward to next
 |            valid
 |          * backfill / bfill: use next valid observation to fill gap
 |          * nearest: use nearest valid observations to fill gap.
 |
 |
 |      copy : bool, default True
 |          Return a new object, even if the passed indexes are the same.
 |      limit : int, default None
 |          Maximum number of consecutive labels to fill for inexact matches.
 |      tolerance : optional
 |          Maximum distance between original and new labels for inexact
 |          matches. The values of the index at the matching locations must
 |          satisfy the equation ``abs(index[indexer] - target) <= tolerance``.
 |
 |          Tolerance may be a scalar value, which applies the same tolerance
 |          to all values, or list-like, which applies variable tolerance per
 |          element. List-like includes list, tuple, array, Series, and must be
 |          the same size as the index and its dtype must exactly match the
 |          index's type.
 |
 |      Returns
 |      -------
 |      Series or DataFrame
 |          Same type as caller, but with changed indices on each axis.
 |
 |      See Also
 |      --------
 |      DataFrame.set_index : Set row labels.
 |      DataFrame.reset_index : Remove row labels or move them to new columns.
 |      DataFrame.reindex : Change to new indices or expand indices.
 |
 |      Notes
 |      -----
 |      Same as calling
 |      ``.reindex(index=other.index, columns=other.columns,...)``.
 |
 |      Examples
 |      --------
 |      >>> df1 = pd.DataFrame([[24.3, 75.7, 'high'],
 |      ...                     [31, 87.8, 'high'],
 |      ...                     [22, 71.6, 'medium'],
 |      ...                     [35, 95, 'medium']],
 |      ...                    columns=['temp_celsius', 'temp_fahrenheit',
 |      ...                             'windspeed'],
 |      ...                    index=pd.date_range(start='2014-02-12',
 |      ...                                        end='2014-02-15', freq='D'))
 |
 |      >>> df1
 |                  temp_celsius  temp_fahrenheit windspeed
 |      2014-02-12          24.3             75.7      high
 |      2014-02-13          31.0             87.8      high
 |      2014-02-14          22.0             71.6    medium
 |      2014-02-15          35.0             95.0    medium
 |
 |      >>> df2 = pd.DataFrame([[28, 'low'],
 |      ...                     [30, 'low'],
 |      ...                     [35.1, 'medium']],
 |      ...                    columns=['temp_celsius', 'windspeed'],
 |      ...                    index=pd.DatetimeIndex(['2014-02-12', '2014-02-13',
 |      ...                                            '2014-02-15']))
 |
 |      >>> df2
 |                  temp_celsius windspeed
 |      2014-02-12          28.0       low
 |      2014-02-13          30.0       low
 |      2014-02-15          35.1    medium
 |
 |      >>> df2.reindex_like(df1)
 |                  temp_celsius  temp_fahrenheit windspeed
 |      2014-02-12          28.0              NaN       low
 |      2014-02-13          30.0              NaN       low
 |      2014-02-14           NaN              NaN       NaN
 |      2014-02-15          35.1              NaN    medium
 |
 |  rename_axis(self, mapper=None, index=None, columns=None, axis=None, copy=True,
inplace=False)
 |      Set the name of the axis for the index or columns.
 |
 |      Parameters
 |      ----------
 |      mapper : scalar, list-like, optional
 |          Value to set the axis name attribute.
 |      index, columns : scalar, list-like, dict-like or function, optional
 |          A scalar, list-like, dict-like or functions transformations to
 |          apply to that axis' values.
 |          Note that the ``columns`` parameter is not allowed if the
 |          object is a Series. This parameter only apply for DataFrame
 |          type objects.
 |
 |          Use either ``mapper`` and ``axis`` to
 |          specify the axis to target with ``mapper``, or ``index``
 |          and/or ``columns``.
 |      axis : {0 or 'index', 1 or 'columns'}, default 0
 |          The axis to rename.
 |      copy : bool, default True
 |          Also copy underlying data.
 |      inplace : bool, default False
 |          Modifies the object directly, instead of creating a new Series
 |          or DataFrame.
 |
 |      Returns
 |      -------
 |      Series, DataFrame, or None
 |          The same type as the caller or None if ``inplace=True``.
 |
 |      See Also
 |      --------
 |      Series.rename : Alter Series index labels or name.
 |      DataFrame.rename : Alter DataFrame index labels or name.
 |      Index.rename : Set new names on index.
 |
 |      Notes
 |      -----
 |      ``DataFrame.rename_axis`` supports two calling conventions
 |
 |      * ``(index=index_mapper, columns=columns_mapper, ...)``
 |      * ``(mapper, axis={'index', 'columns'}, ...)``
 |
 |      The first calling convention will only modify the names of
 |      the index and/or the names of the Index object that is the columns.
 |      In this case, the parameter ``copy`` is ignored.
 |
 |      The second calling convention will modify the names of the
 |      corresponding index if mapper is a list or a scalar.
 |      However, if mapper is dict-like or a function, it will use the
 |      deprecated behavior of modifying the axis *labels*.
 |
 |      We *highly* recommend using keyword arguments to clarify your
 |      intent.
 |
 |      Examples
 |      --------
 |      **Series**
 |
 |      >>> s = pd.Series(["dog", "cat", "monkey"])
 |      >>> s
 |      0       dog
 |      1       cat
 |      2    monkey
 |      dtype: object
 |      >>> s.rename_axis("animal")
 |      animal
 |      0    dog
 |      1    cat
 |      2    monkey
 |      dtype: object
 |
 |      **DataFrame**
 |
 |      >>> df = pd.DataFrame({"num_legs": [4, 4, 2],
 |      ...                    "num_arms": [0, 0, 2]},
 |      ...                   ["dog", "cat", "monkey"])
 |      >>> df
 |              num_legs  num_arms
 |      dog            4         0
 |      cat            4         0
 |      monkey         2         2
 |      >>> df = df.rename_axis("animal")
 |      >>> df
 |              num_legs  num_arms
 |      animal
 |      dog            4         0
 |      cat            4         0
 |      monkey         2         2
 |      >>> df = df.rename_axis("limbs", axis="columns")
 |      >>> df
 |      limbs   num_legs  num_arms
 |      animal
 |      dog            4         0
 |      cat            4         0
 |      monkey         2         2
 |
 |      **MultiIndex**
 |
 |      >>> df.index = pd.MultiIndex.from_product([['mammal'],
 |      ...                                        ['dog', 'cat', 'monkey']],
 |      ...                                       names=['type', 'name'])
 |      >>> df
 |      limbs          num_legs  num_arms
 |      type   name
 |      mammal dog            4         0
 |             cat            4         0
 |             monkey         2         2
 |
 |      >>> df.rename_axis(index={'type': 'class'})
 |      limbs          num_legs  num_arms
 |      class  name
 |      mammal dog            4         0
 |             cat            4         0
 |             monkey         2         2
 |
 |      >>> df.rename_axis(columns=str.upper)
 |      LIMBS          num_legs  num_arms
 |      type   name
 |      mammal dog            4         0
 |             cat            4         0
 |             monkey         2         2
 |
 |  rolling(self, window: 'int | timedelta | BaseOffset | BaseIndexer', min_periods:
'int | None' = None, center: 'bool_t' = False, win_type: 'str | None' = None, on:
'str | None' = None, axis: 'Axis' = 0, closed: 'str | None' = None, method: 'str' =
'single')
 |      Provide rolling window calculations.
 |
 |      Parameters
 |      ----------
 |      window : int, offset, or BaseIndexer subclass
 |          Size of the moving window. This is the number of observations used for
 |          calculating the statistic. Each window will be a fixed size.
 |
 |          If its an offset then this will be the time period of each window. Each
 |          window will be a variable sized based on the observations included in
 |          the time-period. This is only valid for datetimelike indexes.
 |
 |          If a BaseIndexer subclass is passed, calculates the window boundaries
 |          based on the defined ``get_window_bounds`` method. Additional rolling
 |          keyword arguments, namely `min_periods`, `center`, and
 |          `closed` will be passed to `get_window_bounds`.
 |      min_periods : int, default None
 |          Minimum number of observations in window required to have a value
 |          (otherwise result is NA). For a window that is specified by an offset,
 |          `min_periods` will default to 1. Otherwise, `min_periods` will default
 |          to the size of the window.
 |      center : bool, default False
 |          Set the labels at the center of the window.
 |      win_type : str, default None
 |          Provide a window type. If ``None``, all points are evenly weighted.
 |          See the notes below for further information.
 |      on : str, optional
 |          For a DataFrame, a datetime-like column or Index level on which
 |          to calculate the rolling window, rather than the DataFrame's index.
 |          Provided integer column is ignored and excluded from result since
 |          an integer index is not used to calculate the rolling window.
 |      axis : int or str, default 0
 |      closed : str, default None
 |          Make the interval closed on the 'right', 'left', 'both' or
 |          'neither' endpoints. Defaults to 'right'.
 |
 |          .. versionchanged:: 1.2.0
 |
 |              The closed parameter with fixed windows is now supported.
 |      method : str {'single', 'table'}, default 'single'
 |          Execute the rolling operation per single column or row (``'single'``)
 |          or over the entire object (``'table'``).
 |
 |          This argument is only implemented when specifying ``engine='numba'``
 |          in the method call.
 |
 |          .. versionadded:: 1.3.0
 |
 |      Returns
 |      -------
 |      a Window or Rolling sub-classed for the particular operation
 |
 |      See Also
 |      --------
 |      expanding : Provides expanding transformations.
 |      ewm : Provides exponential weighted functions.
 |
 |      Notes
 |      -----
 |      By default, the result is set to the right edge of the window. This can be
 |      changed to the center of the window by setting ``center=True``.
 |
 |      To learn more about the offsets & frequency strings, please see `this link
 |      <https://pandas.pydata.org/pandas-
docs/stable/user_guide/timeseries.html#offset-aliases>`__.
 |
 |      If ``win_type=None``, all points are evenly weighted; otherwise, ``win_type``
 |      can accept a string of any `scipy.signal window function
 |      <https://docs.scipy.org/doc/scipy/reference/signal.windows.html#module-
scipy.signal.windows>`__.
 |
 |      Certain Scipy window types require additional parameters to be passed
 |      in the aggregation function. The additional parameters must match
 |      the keywords specified in the Scipy window type method signature.
 |      Please see the third example below on how to add the additional parameters.
 |
 |      Examples
 |      --------
 |      >>> df = pd.DataFrame({'B': [0, 1, 2, np.nan, 4]})
 |      >>> df
 |           B
 |      0  0.0
 |      1  1.0
 |      2  2.0
 |      3  NaN
 |      4  4.0
 |
 |      Rolling sum with a window length of 2, using the 'triang'
 |      window type.
 |
 |      >>> df.rolling(2, win_type='triang').sum()
 |           B
 |      0  NaN
 |      1  0.5
 |      2  1.5
 |      3  NaN
 |      4  NaN
 |
 |      Rolling sum with a window length of 2, using the 'gaussian'
 |      window type (note how we need to specify std).
 |
 |      >>> df.rolling(2, win_type='gaussian').sum(std=3)
 |                B
 |      0       NaN
 |      1  0.986207
 |      2  2.958621
 |      3       NaN
 |      4       NaN
 |
 |      Rolling sum with a window length of 2, min_periods defaults
 |      to the window length.
 |
 |      >>> df.rolling(2).sum()
 |           B
 |      0  NaN
 |      1  1.0
 |      2  3.0
 |      3  NaN
 |      4  NaN
 |
 |      Same as above, but explicitly set the min_periods
 |
 |      >>> df.rolling(2, min_periods=1).sum()
 |           B
 |      0  0.0
 |      1  1.0
 |      2  3.0
 |      3  2.0
 |      4  4.0
 |
 |      Same as above, but with forward-looking windows
 |
 |      >>> indexer = pd.api.indexers.FixedForwardWindowIndexer(window_size=2)
 |      >>> df.rolling(window=indexer, min_periods=1).sum()
 |           B
 |      0  1.0
 |      1  3.0
 |      2  2.0
 |      3  4.0
 |      4  4.0
 |
 |      A ragged (meaning not-a-regular frequency), time-indexed DataFrame
 |
 |      >>> df = pd.DataFrame({'B': [0, 1, 2, np.nan, 4]},
 |      ...                   index = [pd.Timestamp('20130101 09:00:00'),
 |      ...                            pd.Timestamp('20130101 09:00:02'),
 |      ...                            pd.Timestamp('20130101 09:00:03'),
 |      ...                            pd.Timestamp('20130101 09:00:05'),
 |      ...                            pd.Timestamp('20130101 09:00:06')])
 |
 |      >>> df
 |                             B
 |      2013-01-01 09:00:00  0.0
 |      2013-01-01 09:00:02  1.0
 |      2013-01-01 09:00:03  2.0
 |      2013-01-01 09:00:05  NaN
 |      2013-01-01 09:00:06  4.0
 |
 |      Contrasting to an integer rolling window, this will roll a variable
 |      length window corresponding to the time period.
 |      The default for min_periods is 1.
 |
 |      >>> df.rolling('2s').sum()
 |                             B
 |      2013-01-01 09:00:00  0.0
 |      2013-01-01 09:00:02  1.0
 |      2013-01-01 09:00:03  3.0
 |      2013-01-01 09:00:05  NaN
 |      2013-01-01 09:00:06  4.0
 |
 |  sample(self: 'FrameOrSeries', n=None, frac: 'float | None' = None, replace:
'bool_t' = False, weights=None, random_state=None, axis: 'Axis | None' = None,
ignore_index: 'bool_t' = False) -> 'FrameOrSeries'
 |      Return a random sample of items from an axis of object.
 |
 |      You can use `random_state` for reproducibility.
 |
 |      Parameters
 |      ----------
 |      n : int, optional
 |          Number of items from axis to return. Cannot be used with `frac`.
 |          Default = 1 if `frac` = None.
 |      frac : float, optional
 |          Fraction of axis items to return. Cannot be used with `n`.
 |      replace : bool, default False
 |          Allow or disallow sampling of the same row more than once.
 |      weights : str or ndarray-like, optional
 |          Default 'None' results in equal probability weighting.
 |          If passed a Series, will align with target object on index. Index
 |          values in weights not found in sampled object will be ignored and
 |          index values in sampled object not in weights will be assigned
 |          weights of zero.
 |          If called on a DataFrame, will accept the name of a column
 |          when axis = 0.
 |          Unless weights are a Series, weights must be same length as axis
 |          being sampled.
 |          If weights do not sum to 1, they will be normalized to sum to 1.
 |          Missing values in the weights column will be treated as zero.
 |          Infinite values not allowed.
 |      random_state : int, array-like, BitGenerator, np.random.RandomState, optional
 |          If int, array-like, or BitGenerator (NumPy>=1.17), seed for
 |          random number generator
 |          If np.random.RandomState, use as numpy RandomState object.
 |
 |          .. versionchanged:: 1.1.0
 |
 |              array-like and BitGenerator (for NumPy>=1.17) object now passed to
 |              np.random.RandomState() as seed
 |
 |      axis : {0 or ‘index’, 1 or ‘columns’, None}, default None
 |          Axis to sample. Accepts axis number or name. Default is stat axis
 |          for given data type (0 for Series and DataFrames).
 |      ignore_index : bool, default False
 |          If True, the resulting index will be labeled 0, 1, …, n - 1.
 |
 |          .. versionadded:: 1.3.0
 |
 |      Returns
 |      -------
 |      Series or DataFrame
 |          A new object of same type as caller containing `n` items randomly
 |          sampled from the caller object.
 |
 |      See Also
 |      --------
 |      DataFrameGroupBy.sample: Generates random samples from each group of a
 |          DataFrame object.
 |      SeriesGroupBy.sample: Generates random samples from each group of a
 |          Series object.
 |      numpy.random.choice: Generates a random sample from a given 1-D numpy
 |          array.
 |
 |      Notes
 |      -----
 |      If `frac` > 1, `replacement` should be set to `True`.
 |
 |      Examples
 |      --------
 |      >>> df = pd.DataFrame({'num_legs': [2, 4, 8, 0],
 |      ...                    'num_wings': [2, 0, 0, 0],
 |      ...                    'num_specimen_seen': [10, 2, 1, 8]},
 |      ...                   index=['falcon', 'dog', 'spider', 'fish'])
 |      >>> df
 |              num_legs  num_wings  num_specimen_seen
 |      falcon         2          2                 10
 |      dog            4          0                  2
 |      spider         8          0                  1
 |      fish           0          0                  8
 |      
 |      Extract 3 random elements from the ``Series`` ``df['num_legs']``:
 |      Note that we use `random_state` to ensure the reproducibility of
 |      the examples.
 |      
 |      >>> df['num_legs'].sample(n=3, random_state=1)
 |      fish      0
 |      spider    8
 |      falcon    2
 |      Name: num_legs, dtype: int64
 |      
 |      A random 50% sample of the ``DataFrame`` with replacement:
 |      
 |      >>> df.sample(frac=0.5, replace=True, random_state=1)
 |            num_legs  num_wings  num_specimen_seen
 |      dog          4          0                  2
 |      fish         0          0                  8
 |      
 |      An upsample sample of the ``DataFrame`` with replacement:
 |      Note that `replace` parameter has to be `True` for `frac` parameter > 1.
 |      
 |      >>> df.sample(frac=2, replace=True, random_state=1)
 |              num_legs  num_wings  num_specimen_seen
 |      dog            4          0                  2
 |      fish           0          0                  8
 |      falcon         2          2                 10
 |      falcon         2          2                 10
 |      fish           0          0                  8
 |      dog            4          0                  2
 |      fish           0          0                  8
 |      dog            4          0                  2
 |      
 |      Using a DataFrame column as weights. Rows with larger value in the
 |      `num_specimen_seen` column are more likely to be sampled.
 |      
 |      >>> df.sample(n=2, weights='num_specimen_seen', random_state=1)
 |              num_legs  num_wings  num_specimen_seen
 |      falcon         2          2                 10
 |      fish           0          0                  8
 |  
 |  set_flags(self: 'FrameOrSeries', *, copy: 'bool_t' = False, 
allows_duplicate_labels: 'bool_t | None' = None) -> 'FrameOrSeries'
 |      Return a new object with updated flags.
 |      
 |      Parameters
 |      ----------
 |      allows_duplicate_labels : bool, optional
 |          Whether the returned object allows duplicate labels.
 |      
 |      Returns
 |      -------
 |      Series or DataFrame
 |          The same type as the caller.
 |      
 |      See Also
 |      --------
 |      DataFrame.attrs : Global metadata applying to this dataset.
 |      DataFrame.flags : Global flags applying to this object.
 |      
 |      Notes
 |      -----
 |      This method returns a new object that's a view on the same data
 |      as the input. Mutating the input or the output values will be reflected
 |      in the other.
 |      
 |      This method is intended to be used in method chains.
 |      
 |      "Flags" differ from "metadata". Flags reflect properties of the
 |      pandas object (the Series or DataFrame). Metadata refer to properties
 |      of the dataset, and should be stored in :attr:`DataFrame.attrs`.
 |      
 |      Examples
 |      --------
 |      >>> df = pd.DataFrame({"A": [1, 2]})
 |      >>> df.flags.allows_duplicate_labels
 |      True
 |      >>> df2 = df.set_flags(allows_duplicate_labels=False)
 |      >>> df2.flags.allows_duplicate_labels
 |      False
 |
 |  slice_shift(self: 'FrameOrSeries', periods: 'int' = 1, axis=0) -> 'FrameOrSeries'
 |      Equivalent to `shift` without copying data.
 |      The shifted data will not include the dropped periods and the
 |      shifted axis will be smaller than the original.
 |
 |      .. deprecated:: 1.2.0
 |          slice_shift is deprecated,
 |          use DataFrame/Series.shift instead.
 |
 |      Parameters
 |      ----------
 |      periods : int
 |          Number of periods to move, can be positive or negative.
 |
 |      Returns
 |      -------
 |      shifted : same type as caller
 |
 |      Notes
 |      -----
 |      While the `slice_shift` is faster than `shift`, you may pay for it
 |      later during alignment.
 |
 |  squeeze(self, axis=None)
 |      Squeeze 1 dimensional axis objects into scalars.
 |
 |      Series or DataFrames with a single element are squeezed to a scalar.
 |      DataFrames with a single column or a single row are squeezed to a
 |      Series. Otherwise the object is unchanged.
 |
 |      This method is most useful when you don't know if your
 |      object is a Series or DataFrame, but you do know it has just a single
 |      column. In that case you can safely call `squeeze` to ensure you have a
 |      Series.
 |
 |      Parameters
 |      ----------
 |      axis : {0 or 'index', 1 or 'columns', None}, default None
 |          A specific axis to squeeze. By default, all length-1 axes are
 |          squeezed.
 |
 |      Returns
 |      -------
 |      DataFrame, Series, or scalar
 |          The projection after squeezing `axis` or all the axes.
 |
 |      See Also
 |      --------
 |      Series.iloc : Integer-location based indexing for selecting scalars.
 |      DataFrame.iloc : Integer-location based indexing for selecting Series.
 |      Series.to_frame : Inverse of DataFrame.squeeze for a
 |          single-column DataFrame.
 |
 |      Examples
 |      --------
 |      >>> primes = pd.Series([2, 3, 5, 7])
 |
 |      Slicing might produce a Series with a single value:
 |
 |      >>> even_primes = primes[primes % 2 == 0]
 |      >>> even_primes
 |      0    2
 |      dtype: int64
 |
 |      >>> even_primes.squeeze()
 |      2
 |
 |      Squeezing objects with more than one value in every axis does nothing:
 |
 |      >>> odd_primes = primes[primes % 2 == 1]
 |      >>> odd_primes
 |      1    3
 |      2    5
 |      3    7
 |      dtype: int64
 |
 |      >>> odd_primes.squeeze()
 |      1    3
 |      2    5
 |      3    7
 |      dtype: int64
 |
 |      Squeezing is even more effective when used with DataFrames.
```

```
|
|      >>> df = pd.DataFrame([[1, 2], [3, 4]], columns=['a', 'b'])
|      >>> df
|         a  b
|      0  1  2
|      1  3  4
|
|      Slicing a single column will produce a DataFrame with the columns
|      having only one value:
|
|      >>> df_a = df[['a']]
|      >>> df_a
|         a
|      0  1
|      1  3
|
|      So the columns can be squeezed down, resulting in a Series:
|
|      >>> df_a.squeeze('columns')
|      0    1
|      1    3
|      Name: a, dtype: int64
|
|      Slicing a single row from a single column will produce a single
|      scalar DataFrame:
|
|      >>> df_0a = df.loc[df.index < 1, ['a']]
|      >>> df_0a
|         a
|      0  1
|
|      Squeezing the rows produces a single scalar Series:
|
|      >>> df_0a.squeeze('rows')
|      a    1
|      Name: 0, dtype: int64
|
|      Squeezing all axes will project directly into a scalar:
|
|      >>> df_0a.squeeze()
|      1
|
|  swapaxes(self: 'FrameOrSeries', axis1, axis2, copy=True) -> 'FrameOrSeries'
|      Interchange axes and swap values axes appropriately.
|
|      Returns
|      -------
|      y : same as input
|
|  tail(self: 'FrameOrSeries', n: 'int' = 5) -> 'FrameOrSeries'
|      Return the last `n` rows.
|
|      This function returns last `n` rows from the object based on
|      position. It is useful for quickly verifying data, for example,
|      after sorting or appending rows.
|
|      For negative values of `n`, this function returns all rows except
|      the first `n` rows, equivalent to ``df[n:]``.
|
|      Parameters
|      ----------
|      n : int, default 5
|          Number of rows to select.
|
|      Returns
|      -------
|      type of caller
|          The last `n` rows of the caller object.
|
|      See Also
|      --------
|      DataFrame.head : The first `n` rows of the caller object.
|
|      Examples
|      --------
|      >>> df = pd.DataFrame({'animal': ['alligator', 'bee', 'falcon', 'lion',
|      ...                    'monkey', 'parrot', 'shark', 'whale', 'zebra']})
|      >>> df
|            animal
|      0  alligator
|      1        bee
|      2     falcon
|      3       lion
|      4     monkey
|      5     parrot
|      6      shark
|      7      whale
```

```
 |      8      zebra
 |
 |      Viewing the last 5 lines
 |
 |      >>> df.tail()
 |         animal
 |      4  monkey
 |      5  parrot
 |      6   shark
 |      7   whale
 |      8   zebra
 |
 |      Viewing the last `n` lines (three in this case)
 |
 |      >>> df.tail(3)
 |        animal
 |      6  shark
 |      7  whale
 |      8  zebra
 |
 |      For negative values of `n`
 |
 |      >>> df.tail(-3)
 |         animal
 |      3    lion
 |      4  monkey
 |      5  parrot
 |      6   shark
 |      7   whale
 |      8   zebra
 |
 |  take(self: 'FrameOrSeries', indices, axis=0, is_copy: 'bool_t | None' = None,
**kwargs) -> 'FrameOrSeries'
 |      Return the elements in the given *positional* indices along an axis.
 |
 |      This means that we are not indexing according to actual values in
 |      the index attribute of the object. We are indexing according to the
 |      actual position of the element in the object.
 |
 |      Parameters
 |      ----------
 |      indices : array-like
 |          An array of ints indicating which positions to take.
 |      axis : {0 or 'index', 1 or 'columns', None}, default 0
 |          The axis on which to select elements. ``0`` means that we are
 |          selecting rows, ``1`` means that we are selecting columns.
 |      is_copy : bool
 |          Before pandas 1.0, ``is_copy=False`` can be specified to ensure
 |          that the return value is an actual copy. Starting with pandas 1.0,
 |          ``take`` always returns a copy, and the keyword is therefore
 |          deprecated.
 |
 |          .. deprecated:: 1.0.0
 |      **kwargs
 |          For compatibility with :meth:`numpy.take`. Has no effect on the
 |          output.
 |
 |      Returns
 |      -------
 |      taken : same type as caller
 |          An array-like containing the elements taken from the object.
 |
 |      See Also
 |      --------
 |      DataFrame.loc : Select a subset of a DataFrame by labels.
 |      DataFrame.iloc : Select a subset of a DataFrame by positions.
 |      numpy.take : Take elements from an array along an axis.
 |
 |      Examples
 |      --------
 |      >>> df = pd.DataFrame([('falcon', 'bird', 389.0),
 |      ...                    ('parrot', 'bird', 24.0),
 |      ...                    ('lion', 'mammal', 80.5),
 |      ...                    ('monkey', 'mammal', np.nan)],
 |      ...                   columns=['name', 'class', 'max_speed'],
 |      ...                   index=[0, 2, 3, 1])
 |      >>> df
 |           name   class  max_speed
 |      0  falcon    bird      389.0
 |      2  parrot    bird       24.0
 |      3    lion  mammal       80.5
 |      1  monkey  mammal        NaN
 |
 |      Take elements at positions 0 and 3 along the axis 0 (default).
 |
 |      Note how the actual indices selected (0 and 1) do not correspond to
 |      our selected indices 0 and 3. That's because we are selecting the 0th
 |      and 3rd rows, not rows whose indices equal 0 and 3.
 |
 |      >>> df.take([0, 3])
 |           name   class  max_speed
 |      0  falcon    bird      389.0
 |      1  monkey  mammal        NaN
 |
 |      Take elements at indices 1 and 2 along the axis 1 (column selection).
 |
 |      >>> df.take([1, 2], axis=1)
 |          class  max_speed
 |      0    bird      389.0
 |      2    bird       24.0
 |      3  mammal       80.5
 |      1  mammal        NaN
 |
 |      We may take elements using negative integers for positive indices,
 |      starting from the end of the object, just like with Python lists.
 |
 |      >>> df.take([-1, -2])
 |           name   class  max_speed
 |      1  monkey  mammal        NaN
 |      3    lion  mammal       80.5
 |
 |  to_clipboard(self, excel: 'bool_t' = True, sep: 'str | None' = None, **kwargs) ->
'None'
 |      Copy object to the system clipboard.
 |
 |      Write a text representation of object to the system clipboard.
 |      This can be pasted into Excel, for example.
 |
 |      Parameters
 |      ----------
 |      excel : bool, default True
 |          Produce output in a csv format for easy pasting into excel.
 |
 |          - True, use the provided separator for csv pasting.
 |          - False, write a string representation of the object to the clipboard.
 |
 |      sep : str, default ``'\t'``
 |          Field delimiter.
 |      **kwargs
 |          These parameters will be passed to DataFrame.to_csv.
 |
 |      See Also
 |      --------
 |      DataFrame.to_csv : Write a DataFrame to a comma-separated values
 |          (csv) file.
 |      read_clipboard : Read text from clipboard and pass to read_table.
 |
 |      Notes
 |      -----
 |      Requirements for your platform.
 |
 |        - Linux : `xclip`, or `xsel` (with `PyQt4` modules)
 |        - Windows : none
 |        - OS X : none
 |
 |      Examples
 |      --------
 |      Copy the contents of a DataFrame to the clipboard.
 |
 |      >>> df = pd.DataFrame([[1, 2, 3], [4, 5, 6]], columns=['A', 'B', 'C'])
 |
 |      >>> df.to_clipboard(sep=',')  # doctest: +SKIP
 |      ... # Wrote the following to the system clipboard:
 |      ... # ,A,B,C
 |      ... # 0,1,2,3
 |      ... # 1,4,5,6
 |
 |      We can omit the index by passing the keyword `index` and setting
 |      it to false.
 |
 |      >>> df.to_clipboard(sep=',', index=False)  # doctest: +SKIP
 |      ... # Wrote the following to the system clipboard:
 |      ... # A,B,C
 |      ... # 1,2,3
 |      ... # 4,5,6
 |
 |  to_csv(self, path_or_buf: 'FilePathOrBuffer[AnyStr] | None' = None, sep: 'str' =
',', na_rep: 'str' = '', float_format: 'str | None' = None, columns:
'Sequence[Hashable] | None' = None, header: 'bool_t | list[str]' = True, index:
'bool_t' = True, index_label: 'IndexLabel | None' = None, mode: 'str' = 'w',
encoding: 'str | None' = None, compression: 'CompressionOptions' = 'infer', quoting:
'int | None' = None, quotechar: 'str' = '"', line_terminator: 'str | None' = None,
chunksize: 'int | None' = None, date_format: 'str | None' = None, doublequote:
'bool_t' = True, escapechar: 'str | None' = None, decimal: 'str' = '.', errors: 'str'
= 'strict', storage_options: 'StorageOptions' = None) -> 'str | None'
 |      Write object to a comma-separated values (csv) file.
 |      
 |      Parameters
 |      ----------
 |      path_or_buf : str or file handle, default None
 |          File path or object, if None is provided the result is returned as
 |          a string.  If a non-binary file object is passed, it should be opened
 |          with `newline=''`, disabling universal newlines. If a binary
 |          file object is passed, `mode` might need to contain a `'b'`.
 |      
 |          .. versionchanged:: 1.2.0
 |      
 |             Support for binary file objects was introduced.
 |      
 |      sep : str, default ','
 |          String of length 1. Field delimiter for the output file.
 |      na_rep : str, default ''
 |          Missing data representation.
 |      float_format : str, default None
 |          Format string for floating point numbers.
 |      columns : sequence, optional
 |          Columns to write.
 |      header : bool or list of str, default True
 |          Write out the column names. If a list of strings is given it is
 |          assumed to be aliases for the column names.
 |      index : bool, default True
 |          Write row names (index).
 |      index_label : str or sequence, or False, default None
 |          Column label for index column(s) if desired. If None is given, and
 |          `header` and `index` are True, then the index names are used. A
 |          sequence should be given if the object uses MultiIndex. If
 |          False do not print fields for index names. Use index_label=False
 |          for easier importing in R.
 |      mode : str
 |          Python write mode, default 'w'.
 |      encoding : str, optional
 |          A string representing the encoding to use in the output file,
 |          defaults to 'utf-8'. `encoding` is not supported if `path_or_buf`
 |          is a non-binary file object.
 |      compression : str or dict, default 'infer'
 |          If str, represents compression mode. If dict, value at 'method' is
 |          the compression mode. Compression mode may be any of the following
 |          possible values: {'infer', 'gzip', 'bz2', 'zip', 'xz', None}. If
 |          compression mode is 'infer' and `path_or_buf` is path-like, then
 |          detect compression mode from the following extensions: '.gz',
 |          '.bz2', '.zip' or '.xz'. (otherwise no compression). If dict given
 |          and mode is one of {'zip', 'gzip', 'bz2'}, or inferred as
 |          one of the above, other entries passed as
 |          additional compression options.
 |      
 |          .. versionchanged:: 1.0.0
 |      
 |             May now be a dict with key 'method' as compression mode
 |             and other entries as additional compression options if
 |             compression mode is 'zip'.
 |      
 |          .. versionchanged:: 1.1.0
 |      
 |             Passing compression options as keys in dict is
 |             supported for compression modes 'gzip' and 'bz2'
 |             as well as 'zip'.
 |      
 |          .. versionchanged:: 1.2.0
 |      
 |              Compression is supported for binary file objects.
 |      
 |          .. versionchanged:: 1.2.0
 |      
 |              Previous versions forwarded dict entries for 'gzip' to
 |              `gzip.open` instead of `gzip.GzipFile` which prevented
 |              setting `mtime`.
 |      
 |      quoting : optional constant from csv module
 |          Defaults to csv.QUOTE_MINIMAL. If you have set a `float_format`
 |          then floats are converted to strings and thus csv.QUOTE_NONNUMERIC
 |          will treat them as non-numeric.
 |      quotechar : str, default '\"'
 |          String of length 1. Character used to quote fields.
 |      line_terminator : str, optional
 |          The newline character or character sequence to use in the output
 |          file. Defaults to `os.linesep`, which depends on the OS in which
 |          this method is called ('\\n' for linux, '\\r\\n' for Windows, i.e.).
 |      chunksize : int or None
 |          Rows to write at a time.
 |      date_format : str, default None
 |          Format string for datetime objects.
 |      doublequote : bool, default True
 |          Control quoting of `quotechar` inside a field.
 |      escapechar : str, default None
 |          String of length 1. Character used to escape `sep` and `quotechar`
 |          when appropriate.
 |      decimal : str, default '.'
 |          Character recognized as decimal separator. E.g. use ',' for
 |          European data.
 |      errors : str, default 'strict'
 |          Specifies how encoding and decoding errors are to be handled.
 |          See the errors argument for :func:`open` for a full list
 |          of options.
 |
 |          .. versionadded:: 1.1.0
 |
 |      storage_options : dict, optional
 |          Extra options that make sense for a particular storage connection, e.g.
 |          host, port, username, password, etc. For HTTP(S) URLs the key-value pairs
 |          are forwarded to ``urllib`` as header options. For other URLs (e.g.
 |          starting with "s3://", and "gcs://") the key-value pairs are forwarded to
 |          ``fsspec``. Please see ``fsspec`` and ``urllib`` for more details.
 |
 |          .. versionadded:: 1.2.0
 |
 |      Returns
 |      -------
 |      None or str
 |          If path_or_buf is None, returns the resulting csv format as a
 |          string. Otherwise returns None.
 |
 |      See Also
 |      --------
 |      read_csv : Load a CSV file into a DataFrame.
 |      to_excel : Write DataFrame to an Excel file.
 |
 |      Examples
 |      --------
 |      >>> df = pd.DataFrame({'name': ['Raphael', 'Donatello'],
 |      ...                    'mask': ['red', 'purple'],
 |      ...                    'weapon': ['sai', 'bo staff']})
 |      >>> df.to_csv(index=False)
 |      'name,mask,weapon\nRaphael,red,sai\nDonatello,purple,bo staff\n'
 |
 |      Create 'out.zip' containing 'out.csv'
 |
 |      >>> compression_opts = dict(method='zip',
 |      ...                         archive_name='out.csv')  # doctest: +SKIP
 |      >>> df.to_csv('out.zip', index=False,
 |      ...           compression=compression_opts)  # doctest: +SKIP
 |
 |  to_excel(self, excel_writer, sheet_name: 'str' = 'Sheet1', na_rep: 'str' = '',
float_format: 'str | None' = None, columns=None, header=True, index=True,
index_label=None, startrow=0, startcol=0, engine=None, merge_cells=True,
encoding=None, inf_rep='inf', verbose=True, freeze_panes=None, storage_options:
'StorageOptions' = None) -> 'None'
 |      Write object to an Excel sheet.
 |
 |      To write a single object to an Excel .xlsx file it is only necessary to
 |      specify a target file name. To write to multiple sheets it is necessary to
 |      create an `ExcelWriter` object with a target file name, and specify a sheet
 |      in the file to write to.
 |
 |      Multiple sheets may be written to by specifying unique `sheet_name`.
 |      With all data written to the file it is necessary to save the changes.
 |      Note that creating an `ExcelWriter` object with a file name that already
 |      exists will result in the contents of the existing file being erased.
 |
 |      Parameters
 |      ----------
 |      excel_writer : path-like, file-like, or ExcelWriter object
 |          File path or existing ExcelWriter.
 |      sheet_name : str, default 'Sheet1'
 |          Name of sheet which will contain DataFrame.
 |      na_rep : str, default ''
 |          Missing data representation.
 |      float_format : str, optional
 |          Format string for floating point numbers. For example
 |          ``float_format="%.2f"`` will format 0.1234 to 0.12.
 |      columns : sequence or list of str, optional
 |          Columns to write.
 |      header : bool or list of str, default True
 |          Write out the column names. If a list of string is given it is
 |          assumed to be aliases for the column names.
 |      index : bool, default True
 |          Write row names (index).
 |      index_label : str or sequence, optional
 |          Column label for index column(s) if desired. If not specified, and
 |              `header` and `index` are True, then the index names are used. A
 |              sequence should be given if the DataFrame uses MultiIndex.
 |          startrow : int, default 0
 |              Upper left cell row to dump data frame.
 |          startcol : int, default 0
 |              Upper left cell column to dump data frame.
 |          engine : str, optional
 |              Write engine to use, 'openpyxl' or 'xlsxwriter'. You can also set this
 |              via the options ``io.excel.xlsx.writer``, ``io.excel.xls.writer``, and
 |              ``io.excel.xlsm.writer``.
 |
 |              .. deprecated:: 1.2.0
 |
 |                  As the `xlwt <https://pypi.org/project/xlwt/>`__ package is no longer
 |                  maintained, the ``xlwt`` engine will be removed in a future version
 |                  of pandas.
 |
 |          merge_cells : bool, default True
 |              Write MultiIndex and Hierarchical Rows as merged cells.
 |          encoding : str, optional
 |              Encoding of the resulting excel file. Only necessary for xlwt,
 |              other writers support unicode natively.
 |          inf_rep : str, default 'inf'
 |              Representation for infinity (there is no native representation for
 |              infinity in Excel).
 |          verbose : bool, default True
 |              Display more information in the error logs.
 |          freeze_panes : tuple of int (length 2), optional
 |              Specifies the one-based bottommost row and rightmost column that
 |              is to be frozen.
 |          storage_options : dict, optional
 |              Extra options that make sense for a particular storage connection, e.g.
 |              host, port, username, password, etc. For HTTP(S) URLs the key-value pairs
 |              are forwarded to ``urllib`` as header options. For other URLs (e.g.
 |              starting with "s3://", and "gcs://") the key-value pairs are forwarded to
 |              ``fsspec``. Please see ``fsspec`` and ``urllib`` for more details.
 |
 |              .. versionadded:: 1.2.0
 |
 |          See Also
 |          --------
 |          to_csv : Write DataFrame to a comma-separated values (csv) file.
 |          ExcelWriter : Class for writing DataFrame objects into excel sheets.
 |          read_excel : Read an Excel file into a pandas DataFrame.
 |          read_csv : Read a comma-separated values (csv) file into DataFrame.
 |
 |          Notes
 |          -----
 |          For compatibility with :meth:`~DataFrame.to_csv`,
 |          to_excel serializes lists and dicts to strings before writing.
 |
 |          Once a workbook has been saved it is not possible to write further
 |          data without rewriting the whole workbook.
 |
 |          Examples
 |          --------
 |
 |          Create, write to and save a workbook:
 |
 |          >>> df1 = pd.DataFrame([['a', 'b'], ['c', 'd']],
 |          ...                    index=['row 1', 'row 2'],
 |          ...                    columns=['col 1', 'col 2'])
 |          >>> df1.to_excel("output.xlsx")  # doctest: +SKIP
 |
 |          To specify the sheet name:
 |
 |          >>> df1.to_excel("output.xlsx",
 |          ...              sheet_name='Sheet_name_1')  # doctest: +SKIP
 |
 |          If you wish to write to more than one sheet in the workbook, it is
 |          necessary to specify an ExcelWriter object:
 |
 |          >>> df2 = df1.copy()
 |          >>> with pd.ExcelWriter('output.xlsx') as writer:  # doctest: +SKIP
 |          ...     df1.to_excel(writer, sheet_name='Sheet_name_1')
 |          ...     df2.to_excel(writer, sheet_name='Sheet_name_2')
 |
 |          ExcelWriter can also be used to append to an existing Excel file:
 |
 |          >>> with pd.ExcelWriter('output.xlsx',
 |          ...                     mode='a') as writer:  # doctest: +SKIP
 |          ...     df.to_excel(writer, sheet_name='Sheet_name_3')
 |
 |          To set the library that is used to write the Excel file,
 |          you can pass the `engine` keyword (the default engine is
 |          automatically chosen depending on the file extension):
 |
 |      >>> df1.to_excel('output1.xlsx', engine='xlsxwriter')  # doctest: +SKIP
 |  
 |  to_hdf(self, path_or_buf, key: 'str', mode: 'str' = 'a', complevel: 'int | None'
= None, complib: 'str | None' = None, append: 'bool_t' = False, format: 'str | None'
= None, index: 'bool_t' = True, min_itemsize: 'int | dict[str, int] | None' = None,
nan_rep=None, dropna: 'bool_t | None' = None, data_columns: 'bool_t | list[str] |
None' = None, errors: 'str' = 'strict', encoding: 'str' = 'UTF-8') -> 'None'
 |      Write the contained data to an HDF5 file using HDFStore.
 |      
 |      Hierarchical Data Format (HDF) is self-describing, allowing an
 |      application to interpret the structure and contents of a file with
 |      no outside information. One HDF file can hold a mix of related objects
 |      which can be accessed as a group or as individual objects.
 |      
 |      In order to add another DataFrame or Series to an existing HDF file
 |      please use append mode and a different a key.
 |      
 |      .. warning::
 |      
 |         One can store a subclass of ``DataFrame`` or ``Series`` to HDF5,
 |         but the type of the subclass is lost upon storing.
 |      
 |      For more information see the :ref:`user guide <io.hdf5>`.
 |      
 |      Parameters
 |      ----------
 |      path_or_buf : str or pandas.HDFStore
 |          File path or HDFStore object.
 |      key : str
 |          Identifier for the group in the store.
 |      mode : {'a', 'w', 'r+'}, default 'a'
 |          Mode to open file:
 |      
 |          - 'w': write, a new file is created (an existing file with
 |            the same name would be deleted).
 |          - 'a': append, an existing file is opened for reading and
 |            writing, and if the file does not exist it is created.
 |          - 'r+': similar to 'a', but the file must already exist.
 |      complevel : {0-9}, optional
 |          Specifies a compression level for data.
 |          A value of 0 disables compression.
 |      complib : {'zlib', 'lzo', 'bzip2', 'blosc'}, default 'zlib'
 |          Specifies the compression library to be used.
 |          As of v0.20.2 these additional compressors for Blosc are supported
 |          (default if no compressor specified: 'blosc:blosclz'):
 |          {'blosc:blosclz', 'blosc:lz4', 'blosc:lz4hc', 'blosc:snappy',
 |          'blosc:zlib', 'blosc:zstd'}.
 |          Specifying a compression library which is not available issues
 |          a ValueError.
 |      append : bool, default False
 |          For Table formats, append the input data to the existing.
 |      format : {'fixed', 'table', None}, default 'fixed'
 |          Possible values:
 |      
 |          - 'fixed': Fixed format. Fast writing/reading. Not-appendable,
 |            nor searchable.
 |          - 'table': Table format. Write as a PyTables Table structure
 |            which may perform worse but allow more flexible operations
 |            like searching / selecting subsets of the data.
 |          - If None, pd.get_option('io.hdf.default_format') is checked,
 |            followed by fallback to "fixed"
 |      errors : str, default 'strict'
 |          Specifies how encoding and decoding errors are to be handled.
 |          See the errors argument for :func:`open` for a full list
 |          of options.
 |      encoding : str, default "UTF-8"
 |      min_itemsize : dict or int, optional
 |          Map column names to minimum string sizes for columns.
 |      nan_rep : Any, optional
 |          How to represent null values as str.
 |          Not allowed with append=True.
 |      data_columns : list of columns or True, optional
 |          List of columns to create as indexed data columns for on-disk
 |          queries, or True to use all columns. By default only the axes
 |          of the object are indexed. See :ref:`io.hdf5-query-data-columns`.
 |          Applicable only to format='table'.
 |      
 |      See Also
 |      --------
 |      read_hdf : Read from HDF file.
 |      DataFrame.to_parquet : Write a DataFrame to the binary parquet format.
 |      DataFrame.to_sql : Write to a SQL table.
 |      DataFrame.to_feather : Write out feather-format for DataFrames.
 |      DataFrame.to_csv : Write out to a csv file.
 |      
 |      Examples
 |      --------
 |      >>> df = pd.DataFrame({'A': [1, 2, 3], 'B': [4, 5, 6]},
 |      ...                   index=['a', 'b', 'c'])
 |      >>> df.to_hdf('data.h5', key='df', mode='w')
 |
 |      We can add another object to the same file:
 |
 |      >>> s = pd.Series([1, 2, 3, 4])
 |      >>> s.to_hdf('data.h5', key='s')
 |
 |      Reading from HDF file:
 |
 |      >>> pd.read_hdf('data.h5', 'df')
 |      A  B
 |      a  1  4
 |      b  2  5
 |      c  3  6
 |      >>> pd.read_hdf('data.h5', 's')
 |      0    1
 |      1    2
 |      2    3
 |      3    4
 |      dtype: int64
 |
 |      Deleting file with data:
 |
 |      >>> import os
 |      >>> os.remove('data.h5')
 |
 |  to_json(self, path_or_buf: 'FilePathOrBuffer | None' = None, orient: 'str | None'
= None, date_format: 'str | None' = None, double_precision: 'int' = 10, force_ascii:
'bool_t' = True, date_unit: 'str' = 'ms', default_handler: 'Callable[[Any],
JSONSerializable] | None' = None, lines: 'bool_t' = False, compression:
'CompressionOptions' = 'infer', index: 'bool_t' = True, indent: 'int | None' = None,
storage_options: 'StorageOptions' = None) -> 'str | None'
 |      Convert the object to a JSON string.
 |
 |      Note NaN's and None will be converted to null and datetime objects
 |      will be converted to UNIX timestamps.
 |
 |      Parameters
 |      ----------
 |      path_or_buf : str or file handle, optional
 |          File path or object. If not specified, the result is returned as
 |          a string.
 |      orient : str
 |          Indication of expected JSON string format.
 |
 |          * Series:
 |
 |              - default is 'index'
 |              - allowed values are: {'split', 'records', 'index', 'table'}.
 |
 |          * DataFrame:
 |
 |              - default is 'columns'
 |              - allowed values are: {'split', 'records', 'index', 'columns',
 |                'values', 'table'}.
 |
 |          * The format of the JSON string:
 |
 |              - 'split' : dict like {'index' -> [index], 'columns' -> [columns],
 |                'data' -> [values]}
 |              - 'records' : list like [{column -> value}, ... , {column -> value}]
 |              - 'index' : dict like {index -> {column -> value}}
 |              - 'columns' : dict like {column -> {index -> value}}
 |              - 'values' : just the values array
 |              - 'table' : dict like {'schema': {schema}, 'data': {data}}
 |
 |              Describing the data, where data component is like
``orient='records'``.
 |
 |      date_format : {None, 'epoch', 'iso'}
 |          Type of date conversion. 'epoch' = epoch milliseconds,
 |          'iso' = ISO8601. The default depends on the `orient`. For
 |          ``orient='table'``, the default is 'iso'. For all other orients,
 |          the default is 'epoch'.
 |      double_precision : int, default 10
 |          The number of decimal places to use when encoding
 |          floating point values.
 |      force_ascii : bool, default True
 |          Force encoded string to be ASCII.
 |      date_unit : str, default 'ms' (milliseconds)
 |          The time unit to encode to, governs timestamp and ISO8601
 |          precision.  One of 's', 'ms', 'us', 'ns' for second, millisecond,
 |          microsecond, and nanosecond respectively.
 |      default_handler : callable, default None
 |          Handler to call if object cannot otherwise be converted to a
 |          suitable format for JSON. Should receive a single argument which is
 |          the object to convert and return a serialisable object.
 |      lines : bool, default False
 |          If 'orient' is 'records' write out line-delimited json format. Will
 |          throw ValueError if incorrect 'orient' since others are not
 |          list-like.
 |      
 |      compression : {'infer', 'gzip', 'bz2', 'zip', 'xz', None}
 |      
 |          A string representing the compression to use in the output file,
 |          only used when the first argument is a filename. By default, the
 |          compression is inferred from the filename.
 |      index : bool, default True
 |          Whether to include the index values in the JSON string. Not
 |          including the index (``index=False``) is only supported when
 |          orient is 'split' or 'table'.
 |      indent : int, optional
 |         Length of whitespace used to indent each record.
 |      
 |         .. versionadded:: 1.0.0
 |      
 |      storage_options : dict, optional
 |          Extra options that make sense for a particular storage connection, e.g.
 |          host, port, username, password, etc. For HTTP(S) URLs the key-value pairs
 |          are forwarded to ``urllib`` as header options. For other URLs (e.g.
 |          starting with "s3://", and "gcs://") the key-value pairs are forwarded to
 |          ``fsspec``. Please see ``fsspec`` and ``urllib`` for more details.
 |      
 |          .. versionadded:: 1.2.0
 |      
 |      Returns
 |      -------
 |      None or str
 |          If path_or_buf is None, returns the resulting json format as a
 |          string. Otherwise returns None.
 |      
 |      See Also
 |      --------
 |      read_json : Convert a JSON string to pandas object.
 |      
 |      Notes
 |      -----
 |      The behavior of ``indent=0`` varies from the stdlib, which does not
 |      indent the output but does insert newlines. Currently, ``indent=0``
 |      and the default ``indent=None`` are equivalent in pandas, though this
 |      may change in a future release.
 |      
 |      ``orient='table'`` contains a 'pandas_version' field under 'schema'.
 |      This stores the version of `pandas` used in the latest revision of the
 |      schema.
 |      
 |      Examples
 |      --------
 |      >>> import json
 |      >>> df = pd.DataFrame(
 |      ...     [["a", "b"], ["c", "d"]],
 |      ...     index=["row 1", "row 2"],
 |      ...     columns=["col 1", "col 2"],
 |      ... )
 |      
 |      >>> result = df.to_json(orient="split")
 |      >>> parsed = json.loads(result)
 |      >>> json.dumps(parsed, indent=4)  # doctest: +SKIP
 |      {
 |          "columns": [
 |              "col 1",
 |              "col 2"
 |          ],
 |          "index": [
 |              "row 1",
 |              "row 2"
 |          ],
 |          "data": [
 |              [
 |                  "a",
 |                  "b"
 |              ],
 |              [
 |                  "c",
 |                  "d"
 |              ]
 |          ]
 |      }
 |      
 |      Encoding/decoding a Dataframe using ``'records'`` formatted JSON.
 |      Note that index labels are not preserved with this encoding.
 |
 |      >>> result = df.to_json(orient="records")
 |      >>> parsed = json.loads(result)
 |      >>> json.dumps(parsed, indent=4)  # doctest: +SKIP
 |      [
 |          {
 |              "col 1": "a",
 |              "col 2": "b"
 |          },
 |          {
 |              "col 1": "c",
 |              "col 2": "d"
 |          }
 |      ]
 |
 |      Encoding/decoding a Dataframe using ``'index'`` formatted JSON:
 |
 |      >>> result = df.to_json(orient="index")
 |      >>> parsed = json.loads(result)
 |      >>> json.dumps(parsed, indent=4)  # doctest: +SKIP
 |      {
 |          "row 1": {
 |              "col 1": "a",
 |              "col 2": "b"
 |          },
 |          "row 2": {
 |              "col 1": "c",
 |              "col 2": "d"
 |          }
 |      }
 |
 |      Encoding/decoding a Dataframe using ``'columns'`` formatted JSON:
 |
 |      >>> result = df.to_json(orient="columns")
 |      >>> parsed = json.loads(result)
 |      >>> json.dumps(parsed, indent=4)  # doctest: +SKIP
 |      {
 |          "col 1": {
 |              "row 1": "a",
 |              "row 2": "c"
 |          },
 |          "col 2": {
 |              "row 1": "b",
 |              "row 2": "d"
 |          }
 |      }
 |
 |      Encoding/decoding a Dataframe using ``'values'`` formatted JSON:
 |
 |      >>> result = df.to_json(orient="values")
 |      >>> parsed = json.loads(result)
 |      >>> json.dumps(parsed, indent=4)  # doctest: +SKIP
 |      [
 |          [
 |              "a",
 |              "b"
 |          ],
 |          [
 |              "c",
 |              "d"
 |          ]
 |      ]
 |
 |      Encoding with Table Schema:
 |
 |      >>> result = df.to_json(orient="table")
 |      >>> parsed = json.loads(result)
 |      >>> json.dumps(parsed, indent=4)  # doctest: +SKIP
 |      {
 |          "schema": {
 |              "fields": [
 |                  {
 |                      "name": "index",
 |                      "type": "string"
 |                  },
 |                  {
 |                      "name": "col 1",
 |                      "type": "string"
 |                  },
 |                  {
 |                      "name": "col 2",
 |                      "type": "string"
 |                  }
 |              ],
 |              "primaryKey": [
 |                  "index"
 |              ],
 |              "pandas_version": "0.20.0"
 |          },
 |          "data": [
 |              {
 |                  "index": "row 1",
 |                  "col 1": "a",
 |                  "col 2": "b"
 |              },
 |              {
 |                  "index": "row 2",
 |                  "col 1": "c",
 |                  "col 2": "d"
 |              }
 |          ]
 |      }
 |
 |  to_latex(self, buf=None, columns=None, col_space=None, header=True, index=True,
na_rep='NaN', formatters=None, float_format=None, sparsify=None, index_names=True,
bold_rows=False, column_format=None, longtable=None, escape=None, encoding=None,
decimal='.', multicolumn=None, multicolumn_format=None, multirow=None, caption=None,
label=None, position=None)
 |      Render object to a LaTeX tabular, longtable, or nested table/tabular.
 |
 |      Requires ``\usepackage{booktabs}``.  The output can be copy/pasted
 |      into a main LaTeX document or read from an external file
 |      with ``\input{table.tex}``.
 |
 |      .. versionchanged:: 1.0.0
 |         Added caption and label arguments.
 |
 |      .. versionchanged:: 1.2.0
 |         Added position argument, changed meaning of caption argument.
 |
 |      Parameters
 |      ----------
 |      buf : str, Path or StringIO-like, optional, default None
 |          Buffer to write to. If None, the output is returned as a string.
 |      columns : list of label, optional
 |          The subset of columns to write. Writes all columns by default.
 |      col_space : int, optional
 |          The minimum width of each column.
 |      header : bool or list of str, default True
 |          Write out the column names. If a list of strings is given,
 |          it is assumed to be aliases for the column names.
 |      index : bool, default True
 |          Write row names (index).
 |      na_rep : str, default 'NaN'
 |          Missing data representation.
 |      formatters : list of functions or dict of {str: function}, optional
 |          Formatter functions to apply to columns' elements by position or
 |          name. The result of each function must be a unicode string.
 |          List must be of length equal to the number of columns.
 |      float_format : one-parameter function or str, optional, default None
 |          Formatter for floating point numbers. For example
 |          ``float_format="%.2f"`` and ``float_format="{:0.2f}".format`` will
 |          both result in 0.1234 being formatted as 0.12.
 |      sparsify : bool, optional
 |          Set to False for a DataFrame with a hierarchical index to print
 |          every multiindex key at each row. By default, the value will be
 |          read from the config module.
 |      index_names : bool, default True
 |          Prints the names of the indexes.
 |      bold_rows : bool, default False
 |          Make the row labels bold in the output.
 |      column_format : str, optional
 |          The columns format as specified in `LaTeX table format
 |          <https://en.wikibooks.org/wiki/LaTeX/Tables>`__ e.g. 'rcl' for 3
 |          columns. By default, 'l' will be used for all columns except
 |          columns of numbers, which default to 'r'.
 |      longtable : bool, optional
 |          By default, the value will be read from the pandas config
 |          module. Use a longtable environment instead of tabular. Requires
 |          adding a \usepackage{longtable} to your LaTeX preamble.
 |      escape : bool, optional
 |          By default, the value will be read from the pandas config
 |          module. When set to False prevents from escaping latex special
 |          characters in column names.
 |      encoding : str, optional
 |          A string representing the encoding to use in the output file,
 |          defaults to 'utf-8'.
 |      decimal : str, default '.'
 |          Character recognized as decimal separator, e.g. ',' in Europe.
 |      multicolumn : bool, default True
 |          Use \multicolumn to enhance MultiIndex columns.
 |          The default will be read from the config module.
 |      multicolumn_format : str, default 'l'
 |          The alignment for multicolumns, similar to `column_format`
 |          The default will be read from the config module.
 |      multirow : bool, default False
 |          Use \multirow to enhance MultiIndex rows. Requires adding a
 |          \usepackage{multirow} to your LaTeX preamble. Will print
 |          centered labels (instead of top-aligned) across the contained
 |          rows, separating groups via clines. The default will be read
 |          from the pandas config module.
 |      caption : str or tuple, optional
 |          Tuple (full_caption, short_caption),
 |          which results in ``\caption[short_caption]{full_caption}``;
 |          if a single string is passed, no short caption will be set.
 |
 |          .. versionadded:: 1.0.0
 |
 |          .. versionchanged:: 1.2.0
 |             Optionally allow caption to be a tuple ``(full_caption,
short_caption)``.
 |
 |      label : str, optional
 |          The LaTeX label to be placed inside ``\label{}`` in the output.
 |          This is used with ``\ref{}`` in the main ``.tex`` file.
 |
 |          .. versionadded:: 1.0.0
 |      position : str, optional
 |          The LaTeX positional argument for tables, to be placed after
 |          ``\begin{}`` in the output.
 |
 |          .. versionadded:: 1.2.0
 |
 |              Returns
 |              -------
 |              str or None
 |                  If buf is None, returns the result as a string. Otherwise returns
 |                  None.
 |
 |      See Also
 |      --------
 |      DataFrame.to_string : Render a DataFrame to a console-friendly
 |          tabular output.
 |      DataFrame.to_html : Render a DataFrame as an HTML table.
 |
 |      Examples
 |      --------
 |      >>> df = pd.DataFrame(dict(name=['Raphael', 'Donatello'],
 |      ...                   mask=['red', 'purple'],
 |      ...                   weapon=['sai', 'bo staff']))
 |      >>> print(df.to_latex(index=False))  # doctest: +NORMALIZE_WHITESPACE
 |      \begin{tabular}{lll}
 |       \toprule
 |             name &    mask &    weapon \\
 |       \midrule
 |          Raphael &     red &       sai \\
 |        Donatello &  purple &  bo staff \\
 |      \bottomrule
 |      \end{tabular}
 |
 |  to_pickle(self, path, compression: 'CompressionOptions' = 'infer', protocol:
'int' = 4, storage_options: 'StorageOptions' = None) -> 'None'
 |      Pickle (serialize) object to file.
 |
 |      Parameters
 |      ----------
 |      path : str
 |          File path where the pickled object will be stored.
 |      compression : {'infer', 'gzip', 'bz2', 'zip', 'xz', None},         default
'infer'
 |          A string representing the compression to use in the output file. By
 |          default, infers from the file extension in specified path.
 |          Compression mode may be any of the following possible
 |          values: {‘infer’, ‘gzip’, ‘bz2’, ‘zip’, ‘xz’, None}. If compression
 |          mode is ‘infer’ and path_or_buf is path-like, then detect
 |          compression mode from the following extensions:
 |          ‘.gz’, ‘.bz2’, ‘.zip’ or ‘.xz’. (otherwise no compression).
 |          If dict given and mode is ‘zip’ or inferred as ‘zip’, other entries
 |          passed as additional compression options.
 |      protocol : int
 |          Int which indicates which protocol should be used by the pickler,
 |          default HIGHEST_PROTOCOL (see [1]_ paragraph 12.1.2). The possible
 |          values are 0, 1, 2, 3, 4, 5. A negative value for the protocol
 |          parameter is equivalent to setting its value to HIGHEST_PROTOCOL.
 |
 |          .. [1] https://docs.python.org/3/library/pickle.html.
 |
 |      storage_options : dict, optional
 |          Extra options that make sense for a particular storage connection, e.g.
 |          host, port, username, password, etc. For HTTP(S) URLs the key-value pairs
 |          are forwarded to ``urllib`` as header options. For other URLs (e.g.
 |          starting with "s3://", and "gcs://") the key-value pairs are forwarded to
 |          ``fsspec``. Please see ``fsspec`` and ``urllib`` for more details.
 |
 |          .. versionadded:: 1.2.0
 |
 |      See Also
 |      --------
 |      read_pickle : Load pickled pandas object (or any object) from file.
 |      DataFrame.to_hdf : Write DataFrame to an HDF5 file.
 |      DataFrame.to_sql : Write DataFrame to a SQL database.
 |      DataFrame.to_parquet : Write a DataFrame to the binary parquet format.
 |
 |      Examples
 |      --------
 |      >>> original_df = pd.DataFrame({"foo": range(5), "bar": range(5, 10)})
 |      >>> original_df
 |         foo  bar
 |      0    0    5
 |      1    1    6
 |      2    2    7
 |      3    3    8
 |      4    4    9
 |      >>> original_df.to_pickle("./dummy.pkl")
 |
 |      >>> unpickled_df = pd.read_pickle("./dummy.pkl")
 |      >>> unpickled_df
 |         foo  bar
 |      0    0    5
 |      1    1    6
 |      2    2    7
 |      3    3    8
 |      4    4    9
 |
 |      >>> import os
 |      >>> os.remove("./dummy.pkl")
 |
 |  to_sql(self, name: 'str', con, schema=None, if_exists: 'str' = 'fail', index:
'bool_t' = True, index_label=None, chunksize=None, dtype: 'DtypeArg | None' = None,
method=None) -> 'None'
 |      Write records stored in a DataFrame to a SQL database.
 |
 |      Databases supported by SQLAlchemy [1]_ are supported. Tables can be
 |      newly created, appended to, or overwritten.
 |
 |      Parameters
 |      ----------
 |      name : str
 |          Name of SQL table.
 |      con : sqlalchemy.engine.(Engine or Connection) or sqlite3.Connection
 |          Using SQLAlchemy makes it possible to use any DB supported by that
 |          library. Legacy support is provided for sqlite3.Connection objects. The
user
 |          is responsible for engine disposal and connection closure for the
SQLAlchemy
 |          connectable See `here
<https://docs.sqlalchemy.org/en/13/core/connections.html>`_.
 |
 |      schema : str, optional
 |          Specify the schema (if database flavor supports this). If None, use
 |          default schema.
 |      if_exists : {'fail', 'replace', 'append'}, default 'fail'
 |          How to behave if the table already exists.
 |
 |          * fail: Raise a ValueError.
 |          * replace: Drop the table before inserting new values.
 |          * append: Insert new values to the existing table.
 |
 |      index : bool, default True
 |          Write DataFrame index as a column. Uses `index_label` as the column
 |          name in the table.
 |      index_label : str or sequence, default None
 |          Column label for index column(s). If None is given (default) and
 |          `index` is True, then the index names are used.
 |          A sequence should be given if the DataFrame uses MultiIndex.
 |      chunksize : int, optional
 |          Specify the number of rows in each batch to be written at a time.
 |          By default, all rows will be written at once.
 |      dtype : dict or scalar, optional
 |          Specifying the datatype for columns. If a dictionary is used, the
 |          keys should be the column names and the values should be the
 |          SQLAlchemy types or strings for the sqlite3 legacy mode. If a
 |          scalar is provided, it will be applied to all columns.
 |      method : {None, 'multi', callable}, optional
 |          Controls the SQL insertion clause used:
 |
 |          * None : Uses standard SQL ``INSERT`` clause (one per row).
 |          * 'multi': Pass multiple values in a single ``INSERT`` clause.
 |          * callable with signature ``(pd_table, conn, keys, data_iter)``.
 |
 |          Details and a sample callable implementation can be found in the
 |          section :ref:`insert method <io.sql.method>`.
 |
 |      Raises
 |      ------
 |      ValueError
 |          When the table already exists and `if_exists` is 'fail' (the
 |          default).
 |
 |      See Also
 |      --------
 |      read_sql : Read a DataFrame from a table.
 |
 |      Notes
 |      -----
 |      Timezone aware datetime columns will be written as
 |      ``Timestamp with timezone`` type with SQLAlchemy if supported by the
 |      database. Otherwise, the datetimes will be stored as timezone unaware
 |      timestamps local to the original timezone.
 |
 |      References
 |      ----------
 |      .. [1] https://docs.sqlalchemy.org
 |      .. [2] https://www.python.org/dev/peps/pep-0249/
 |
 |      Examples
 |      --------
 |      Create an in-memory SQLite database.
 |
 |      >>> from sqlalchemy import create_engine
 |      >>> engine = create_engine('sqlite://', echo=False)
 |
 |      Create a table from scratch with 3 rows.
 |
 |      >>> df = pd.DataFrame({'name' : ['User 1', 'User 2', 'User 3']})
 |      >>> df
 |           name
 |      0  User 1
 |      1  User 2
 |      2  User 3
 |
 |      >>> df.to_sql('users', con=engine)
 |      >>> engine.execute("SELECT * FROM users").fetchall()
 |      [(0, 'User 1'), (1, 'User 2'), (2, 'User 3')]
 |
 |      An `sqlalchemy.engine.Connection` can also be passed to `con`:
 |
 |      >>> with engine.begin() as connection:
 |      ...     df1 = pd.DataFrame({'name' : ['User 4', 'User 5']})
 |      ...     df1.to_sql('users', con=connection, if_exists='append')
 |
 |      This is allowed to support operations that require that the same
 |      DBAPI connection is used for the entire operation.
 |
 |      >>> df2 = pd.DataFrame({'name' : ['User 6', 'User 7']})
 |      >>> df2.to_sql('users', con=engine, if_exists='append')
 |      >>> engine.execute("SELECT * FROM users").fetchall()
 |      [(0, 'User 1'), (1, 'User 2'), (2, 'User 3'),
 |       (0, 'User 4'), (1, 'User 5'), (0, 'User 6'),
 |       (1, 'User 7')]
 |
 |      Overwrite the table with just ``df2``.
 |
 |      >>> df2.to_sql('users', con=engine, if_exists='replace',
 |      ...            index_label='id')
 |      >>> engine.execute("SELECT * FROM users").fetchall()
 |      [(0, 'User 6'), (1, 'User 7')]
 |
 |      Specify the dtype (especially useful for integers with missing values).
 |      Notice that while pandas is forced to store the data as floating point,
 |      the database supports nullable integers. When fetching the data with
 |      Python, we get back integer scalars.
 |
 |      >>> df = pd.DataFrame({"A": [1, None, 2]})
 |      >>> df
 |           A
 |      0  1.0
 |      1  NaN
 |      2  2.0
 |
 |      >>> from sqlalchemy.types import Integer
 |      >>> df.to_sql('integers', con=engine, index=False,
 |      ...           dtype={"A": Integer()})
 |
 |      >>> engine.execute("SELECT * FROM integers").fetchall()
 |      [(1,), (None,), (2,)]
 |  
 |  to_xarray(self)
 |      Return an xarray object from the pandas object.
 |      
 |      Returns
 |      -------
 |      xarray.DataArray or xarray.Dataset
 |          Data in the pandas structure converted to Dataset if the object is
 |          a DataFrame, or a DataArray if the object is a Series.
 |      
 |      See Also
 |      --------
 |      DataFrame.to_hdf : Write DataFrame to an HDF5 file.
 |      DataFrame.to_parquet : Write a DataFrame to the binary parquet format.
 |      
 |      Notes
 |      -----
 |      See the `xarray docs <https://xarray.pydata.org/en/stable/>`__
 |      
 |      Examples
 |      --------
 |      >>> df = pd.DataFrame([('falcon', 'bird', 389.0, 2),
 |      ...                    ('parrot', 'bird', 24.0, 2),
 |      ...                    ('lion', 'mammal', 80.5, 4),
 |      ...                    ('monkey', 'mammal', np.nan, 4)],
 |      ...                   columns=['name', 'class', 'max_speed',
 |      ...                            'num_legs'])
 |      >>> df
 |           name   class  max_speed  num_legs
 |      0  falcon    bird      389.0         2
 |      1  parrot    bird       24.0         2
 |      2    lion  mammal       80.5         4
 |      3  monkey  mammal        NaN         4
 |      
 |      >>> df.to_xarray()
 |      <xarray.Dataset>
 |      Dimensions:    (index: 4)
 |      Coordinates:
 |        * index      (index) int64 0 1 2 3
 |      Data variables:
 |          name       (index) object 'falcon' 'parrot' 'lion' 'monkey'
 |          class      (index) object 'bird' 'bird' 'mammal' 'mammal'
 |          max_speed  (index) float64 389.0 24.0 80.5 nan
 |          num_legs   (index) int64 2 2 4 4
 |      
 |      >>> df['max_speed'].to_xarray()
 |      <xarray.DataArray 'max_speed' (index: 4)>
 |      array([389. ,  24. ,  80.5,   nan])
 |      Coordinates:
 |        * index    (index) int64 0 1 2 3
 |      
 |      >>> dates = pd.to_datetime(['2018-01-01', '2018-01-01',
 |      ...                         '2018-01-02', '2018-01-02'])
 |      >>> df_multiindex = pd.DataFrame({'date': dates,
 |      ...                               'animal': ['falcon', 'parrot',
 |      ...                                          'falcon', 'parrot'],
 |      ...                               'speed': [350, 18, 361, 15]})
 |      >>> df_multiindex = df_multiindex.set_index(['date', 'animal'])
 |      
 |      >>> df_multiindex
 |                         speed
 |      date       animal
 |      2018-01-01 falcon    350
 |                 parrot     18
 |      2018-01-02 falcon    361
 |                 parrot     15
 |      
 |      >>> df_multiindex.to_xarray()
 |      <xarray.Dataset>
 |      Dimensions:  (animal: 2, date: 2)
 |      Coordinates:
 |        * date     (date) datetime64[ns] 2018-01-01 2018-01-02
 |        * animal   (animal) object 'falcon' 'parrot'
 |      Data variables:
 |          speed    (date, animal) int64 350 18 361 15
 |  
 |  truncate(self: 'FrameOrSeries', before=None, after=None, axis=None, copy:
'bool_t' = True) -> 'FrameOrSeries'
 |      Truncate a Series or DataFrame before and after some index value.
 |      
 |      This is a useful shorthand for boolean indexing based on index
 |      values above or below certain thresholds.
 |      
 |      Parameters
 |      ----------
 |      before : date, str, int
 |          Truncate all rows before this index value.
 |      after : date, str, int
 |          Truncate all rows after this index value.
 |      axis : {0 or 'index', 1 or 'columns'}, optional
 |          Axis to truncate. Truncates the index (rows) by default.
 |      copy : bool, default is True,
 |          Return a copy of the truncated section.
 |
 |      Returns
 |      -------
 |      type of caller
 |          The truncated Series or DataFrame.
 |
 |      See Also
 |      --------
 |      DataFrame.loc : Select a subset of a DataFrame by label.
 |      DataFrame.iloc : Select a subset of a DataFrame by position.
 |
 |      Notes
 |      -----
 |      If the index being truncated contains only datetime values,
 |      `before` and `after` may be specified as strings instead of
 |      Timestamps.
 |
 |      Examples
 |      --------
 |      >>> df = pd.DataFrame({'A': ['a', 'b', 'c', 'd', 'e'],
 |      ...                    'B': ['f', 'g', 'h', 'i', 'j'],
 |      ...                    'C': ['k', 'l', 'm', 'n', 'o']},
 |      ...                   index=[1, 2, 3, 4, 5])
 |      >>> df
 |         A  B  C
 |      1  a  f  k
 |      2  b  g  l
 |      3  c  h  m
 |      4  d  i  n
 |      5  e  j  o
 |
 |      >>> df.truncate(before=2, after=4)
 |         A  B  C
 |      2  b  g  l
 |      3  c  h  m
 |      4  d  i  n
 |
 |      The columns of a DataFrame can be truncated.
 |
 |      >>> df.truncate(before="A", after="B", axis="columns")
 |         A  B
 |      1  a  f
 |      2  b  g
 |      3  c  h
 |      4  d  i
 |      5  e  j
 |
 |      For Series, only rows can be truncated.
 |
 |      >>> df['A'].truncate(before=2, after=4)
 |      2    b
 |      3    c
 |      4    d
 |      Name: A, dtype: object
 |
 |      The index values in ``truncate`` can be datetimes or string
 |      dates.
 |
 |      >>> dates = pd.date_range('2016-01-01', '2016-02-01', freq='s')
 |      >>> df = pd.DataFrame(index=dates, data={'A': 1})
 |      >>> df.tail()
 |                           A
 |      2016-01-31 23:59:56  1
 |      2016-01-31 23:59:57  1
 |      2016-01-31 23:59:58  1
 |      2016-01-31 23:59:59  1
 |      2016-02-01 00:00:00  1
 |
 |      >>> df.truncate(before=pd.Timestamp('2016-01-05'),
 |      ...             after=pd.Timestamp('2016-01-10')).tail()
 |                           A
 |      2016-01-09 23:59:56  1
 |      2016-01-09 23:59:57  1
 |      2016-01-09 23:59:58  1
 |      2016-01-09 23:59:59  1
 |      2016-01-10 00:00:00  1
 |
 |      Because the index is a DatetimeIndex containing only dates, we can
 |      specify `before` and `after` as strings. They will be coerced to
 |      Timestamps before truncation.
 |
 |      >>> df.truncate('2016-01-05', '2016-01-10').tail()
 |                           A
 |      2016-01-09 23:59:56  1
 |      2016-01-09 23:59:57  1
 |      2016-01-09 23:59:58  1
 |      2016-01-09 23:59:59  1
 |      2016-01-10 00:00:00  1
 |      
 |      Note that ``truncate`` assumes a 0 value for any unspecified time
 |      component (midnight). This differs from partial string slicing, which
 |      returns any partially matching dates.
 |      
 |      >>> df.loc['2016-01-05':'2016-01-10', :].tail()
 |                           A
 |      2016-01-10 23:59:55  1
 |      2016-01-10 23:59:56  1
 |      2016-01-10 23:59:57  1
 |      2016-01-10 23:59:58  1
 |      2016-01-10 23:59:59  1
 |  
 |  tshift(self: 'FrameOrSeries', periods: 'int' = 1, freq=None, axis: 'Axis' = 0) -> 
'FrameOrSeries'
 |      Shift the time index, using the index's frequency if available.
 |      
 |      .. deprecated:: 1.1.0
 |          Use `shift` instead.
 |      
 |      Parameters
 |      ----------
 |      periods : int
 |          Number of periods to move, can be positive or negative.
 |      freq : DateOffset, timedelta, or str, default None
 |          Increment to use from the tseries module
 |          or time rule expressed as a string (e.g. 'EOM').
 |      axis : {0 or ‘index’, 1 or ‘columns’, None}, default 0
 |          Corresponds to the axis that contains the Index.
 |      
 |      Returns
 |      -------
 |      shifted : Series/DataFrame
 |      
 |      Notes
 |      -----
 |      If freq is not specified then tries to use the freq or inferred_freq
 |      attributes of the index. If neither of those attributes exist, a
 |      ValueError is thrown
 |  
 |  tz_convert(self: 'FrameOrSeries', tz, axis=0, level=None, copy: 'bool_t' = True) 
-> 'FrameOrSeries'
 |      Convert tz-aware axis to target time zone.
 |      
 |      Parameters
 |      ----------
 |      tz : str or tzinfo object
 |      axis : the axis to convert
 |      level : int, str, default None
 |          If axis is a MultiIndex, convert a specific level. Otherwise
 |          must be None.
 |      copy : bool, default True
 |          Also make a copy of the underlying data.
 |      
 |      Returns
 |      -------
 |      {klass}
 |          Object with time zone converted axis.
 |      
 |      Raises
 |      ------
 |      TypeError
 |          If the axis is tz-naive.
 |  
 |  tz_localize(self: 'FrameOrSeries', tz, axis=0, level=None, copy: 'bool_t' = True, 
ambiguous='raise', nonexistent: 'str' = 'raise') -> 'FrameOrSeries'
 |      Localize tz-naive index of a Series or DataFrame to target time zone.
 |      
 |      This operation localizes the Index. To localize the values in a
 |      timezone-naive Series, use :meth:`Series.dt.tz_localize`.
 |      
 |      Parameters
 |      ----------
 |      tz : str or tzinfo
 |      axis : the axis to localize
 |      level : int, str, default None
 |          If axis ia a MultiIndex, localize a specific level. Otherwise
 |          must be None.
 |      copy : bool, default True
 |          Also make a copy of the underlying data.
 |      ambiguous : 'infer', bool-ndarray, 'NaT', default 'raise'
 |          When clocks moved backward due to DST, ambiguous times may arise.
 |          For example in Central European Time (UTC+01), when going from
 |          03:00 DST to 02:00 non-DST, 02:30:00 local time occurs both at
 |          00:30:00 UTC and at 01:30:00 UTC. In such a situation, the
 |          `ambiguous` parameter dictates how ambiguous times should be
 |          handled.
 |
 |          - 'infer' will attempt to infer fall dst-transition hours based on
 |            order
 |          - bool-ndarray where True signifies a DST time, False designates
 |            a non-DST time (note that this flag is only applicable for
 |            ambiguous times)
 |          - 'NaT' will return NaT where there are ambiguous times
 |          - 'raise' will raise an AmbiguousTimeError if there are ambiguous
 |            times.
 |      nonexistent : str, default 'raise'
 |          A nonexistent time does not exist in a particular timezone
 |          where clocks moved forward due to DST. Valid values are:
 |
 |          - 'shift_forward' will shift the nonexistent time forward to the
 |            closest existing time
 |          - 'shift_backward' will shift the nonexistent time backward to the
 |            closest existing time
 |          - 'NaT' will return NaT where there are nonexistent times
 |          - timedelta objects will shift nonexistent times by the timedelta
 |          - 'raise' will raise an NonExistentTimeError if there are
 |            nonexistent times.
 |
 |      Returns
 |      -------
 |      Series or DataFrame
 |          Same type as the input.
 |
 |      Raises
 |      ------
 |      TypeError
 |          If the TimeSeries is tz-aware and tz is not None.
 |
 |      Examples
 |      --------
 |      Localize local times:
 |
 |      >>> s = pd.Series([1],
 |      ...               index=pd.DatetimeIndex(['2018-09-15 01:30:00']))
 |      >>> s.tz_localize('CET')
 |      2018-09-15 01:30:00+02:00    1
 |      dtype: int64
 |
 |      Be careful with DST changes. When there is sequential data, pandas
 |      can infer the DST time:
 |
 |      >>> s = pd.Series(range(7),
 |      ...               index=pd.DatetimeIndex(['2018-10-28 01:30:00',
 |      ...                                       '2018-10-28 02:00:00',
 |      ...                                       '2018-10-28 02:30:00',
 |      ...                                       '2018-10-28 02:00:00',
 |      ...                                       '2018-10-28 02:30:00',
 |      ...                                       '2018-10-28 03:00:00',
 |      ...                                       '2018-10-28 03:30:00']))
 |      >>> s.tz_localize('CET', ambiguous='infer')
 |      2018-10-28 01:30:00+02:00    0
 |      2018-10-28 02:00:00+02:00    1
 |      2018-10-28 02:30:00+02:00    2
 |      2018-10-28 02:00:00+01:00    3
 |      2018-10-28 02:30:00+01:00    4
 |      2018-10-28 03:00:00+01:00    5
 |      2018-10-28 03:30:00+01:00    6
 |      dtype: int64
 |
 |      In some cases, inferring the DST is impossible. In such cases, you can
 |      pass an ndarray to the ambiguous parameter to set the DST explicitly
 |
 |      >>> s = pd.Series(range(3),
 |      ...               index=pd.DatetimeIndex(['2018-10-28 01:20:00',
 |      ...                                       '2018-10-28 02:36:00',
 |      ...                                       '2018-10-28 03:46:00']))
 |      >>> s.tz_localize('CET', ambiguous=np.array([True, True, False]))
 |      2018-10-28 01:20:00+02:00    0
 |      2018-10-28 02:36:00+02:00    1
 |      2018-10-28 03:46:00+01:00    2
 |      dtype: int64
 |
 |      If the DST transition causes nonexistent times, you can shift these
 |      dates forward or backward with a timedelta object or `'shift_forward'`
 |      or `'shift_backward'`.
 |
 |      >>> s = pd.Series(range(2),
 |      ...               index=pd.DatetimeIndex(['2015-03-29 02:30:00',
 |      ...                                       '2015-03-29 03:30:00']))
 |      >>> s.tz_localize('Europe/Warsaw', nonexistent='shift_forward')
 |      2015-03-29 03:00:00+02:00    0
 |      2015-03-29 03:30:00+02:00    1
 |      dtype: int64
 |      >>> s.tz_localize('Europe/Warsaw', nonexistent='shift_backward')
 |      2015-03-29 01:59:59.999999999+01:00    0
 |      2015-03-29 03:30:00+02:00              1
 |      dtype: int64
 |      >>> s.tz_localize('Europe/Warsaw', nonexistent=pd.Timedelta('1H'))
 |      2015-03-29 03:30:00+02:00    0
 |      2015-03-29 03:30:00+02:00    1
 |      dtype: int64
 |  
 |  xs(self, key, axis=0, level=None, drop_level: 'bool_t' = True)
 |      Return cross-section from the Series/DataFrame.
 |      
 |      This method takes a `key` argument to select data at a particular
 |      level of a MultiIndex.
 |      
 |      Parameters
 |      ----------
 |      key : label or tuple of label
 |          Label contained in the index, or partially in a MultiIndex.
 |      axis : {0 or 'index', 1 or 'columns'}, default 0
 |          Axis to retrieve cross-section on.
 |      level : object, defaults to first n levels (n=1 or len(key))
 |          In case of a key partially contained in a MultiIndex, indicate
 |          which levels are used. Levels can be referred by label or position.
 |      drop_level : bool, default True
 |          If False, returns object with same levels as self.
 |      
 |      Returns
 |      -------
 |      Series or DataFrame
 |          Cross-section from the original Series or DataFrame
 |          corresponding to the selected index levels.
 |      
 |      See Also
 |      --------
 |      DataFrame.loc : Access a group of rows and columns
 |          by label(s) or a boolean array.
 |      DataFrame.iloc : Purely integer-location based indexing
 |          for selection by position.
 |      
 |      Notes
 |      -----
 |      `xs` can not be used to set values.
 |      
 |      MultiIndex Slicers is a generic way to get/set values on
 |      any level or levels.
 |      It is a superset of `xs` functionality, see
 |      :ref:`MultiIndex Slicers <advanced.mi_slicers>`.
 |      
 |      Examples
 |      --------
 |      >>> d = {'num_legs': [4, 4, 2, 2],
 |      ...      'num_wings': [0, 0, 2, 2],
 |      ...      'class': ['mammal', 'mammal', 'mammal', 'bird'],
 |      ...      'animal': ['cat', 'dog', 'bat', 'penguin'],
 |      ...      'locomotion': ['walks', 'walks', 'flies', 'walks']}
 |      >>> df = pd.DataFrame(data=d)
 |      >>> df = df.set_index(['class', 'animal', 'locomotion'])
 |      >>> df
 |                                 num_legs  num_wings
 |      class  animal  locomotion
 |      mammal cat     walks              4          0
 |             dog     walks              4          0
 |             bat     flies              2          2
 |      bird   penguin walks              2          2
 |      
 |      Get values at specified index
 |      
 |      >>> df.xs('mammal')
 |                         num_legs  num_wings
 |      animal locomotion
 |      cat    walks              4          0
 |      dog    walks              4          0
 |      bat    flies              2          2
 |      
 |      Get values at several indexes
 |      
 |      >>> df.xs(('mammal', 'dog'))
 |                  num_legs  num_wings
 |      locomotion
```

```
|      walks              4          0
|
|      Get values at specified index and level
|
|      >>> df.xs('cat', level=1)
|                         num_legs  num_wings
|      class  locomotion
|      mammal walks              4          0
|
|      Get values at several indexes and levels
|
|      >>> df.xs(('bird', 'walks'),
|      ...       level=[0, 'locomotion'])
|               num_legs  num_wings
|      animal
|      penguin         2          2
|
|      Get values at specified column and axis
|
|      >>> df.xs('num_wings', axis=1)
|      class   animal   locomotion
|      mammal  cat      walks         0
|              dog      walks         0
|              bat      flies         2
|      bird    penguin  walks         2
|      Name: num_wings, dtype: int64
|
|  ----------------------------------------------------------------------
|  Data descriptors inherited from pandas.core.generic.NDFrame:
|
|  attrs
|      Dictionary of global attributes of this dataset.
|
|      .. warning::
|
|         attrs is experimental and may change without warning.
|
|      See Also
|      --------
|      DataFrame.flags : Global flags applying to this object.
|
|  dtypes
|      Return the dtypes in the DataFrame.
|
|      This returns a Series with the data type of each column.
|      The result's index is the original DataFrame's columns. Columns
|      with mixed types are stored with the ``object`` dtype. See
|      :ref:`the User Guide <basics.dtypes>` for more.
|
|      Returns
|      -------
|      pandas.Series
|          The data type of each column.
|
|      Examples
|      --------
|      >>> df = pd.DataFrame({'float': [1.0],
|      ...                    'int': [1],
|      ...                    'datetime': [pd.Timestamp('20180310')],
|      ...                    'string': ['foo']})
|      >>> df.dtypes
|      float              float64
|      int                  int64
|      datetime    datetime64[ns]
|      string              object
|      dtype: object
|
|  empty
|      Indicator whether DataFrame is empty.
|
|      True if DataFrame is entirely empty (no items), meaning any of the
|      axes are of length 0.
|
|      Returns
|      -------
|      bool
|          If DataFrame is empty, return True, if not return False.
|
|      See Also
|      --------
|      Series.dropna : Return series without null values.
|      DataFrame.dropna : Return DataFrame with labels on given axis omitted
|          where (all or any) data are missing.
|
|      Notes
|      -----
|      If DataFrame contains only NaNs, it is still not considered empty. See
|      the example below.
|
|      Examples
|      --------
|      An example of an actual empty DataFrame. Notice the index is empty:
|
|      >>> df_empty = pd.DataFrame({'A' : []})
|      >>> df_empty
|      Empty DataFrame
|      Columns: [A]
|      Index: []
|      >>> df_empty.empty
|      True
|
|      If we only have NaNs in our DataFrame, it is not considered empty! We
|      will need to drop the NaNs to make the DataFrame empty:
|
|      >>> df = pd.DataFrame({'A' : [np.nan]})
|      >>> df
|          A
|      0 NaN
|      >>> df.empty
|      False
|      >>> df.dropna().empty
|      True
|
|  flags
|      Get the properties associated with this pandas object.
|
|      The available flags are
|
|      * :attr:`Flags.allows_duplicate_labels`
|
|      See Also
|      --------
|      Flags : Flags that apply to pandas objects.
|      DataFrame.attrs : Global metadata applying to this dataset.
|
|      Notes
|      -----
|      "Flags" differ from "metadata". Flags reflect properties of the
|      pandas object (the Series or DataFrame). Metadata refer to properties
|      of the dataset, and should be stored in :attr:`DataFrame.attrs`.
|
|      Examples
|      --------
|      >>> df = pd.DataFrame({"A": [1, 2]})
|      >>> df.flags
|      <Flags(allows_duplicate_labels=True)>
|
|      Flags can be get or set using ``.``
|
|      >>> df.flags.allows_duplicate_labels
|      True
|      >>> df.flags.allows_duplicate_labels = False
|
|      Or by slicing with a key
|
|      >>> df.flags["allows_duplicate_labels"]
|      False
|      >>> df.flags["allows_duplicate_labels"] = True
|
|  ndim
|      Return an int representing the number of axes / array dimensions.
|
|      Return 1 if Series. Otherwise return 2 if DataFrame.
|
|      See Also
|      --------
|      ndarray.ndim : Number of array dimensions.
|
|      Examples
|      --------
|      >>> s = pd.Series({'a': 1, 'b': 2, 'c': 3})
|      >>> s.ndim
|      1
|
|      >>> df = pd.DataFrame({'col1': [1, 2], 'col2': [3, 4]})
|      >>> df.ndim
|      2
|
|  size
|      Return an int representing the number of elements in this object.
|
|      Return the number of rows if Series. Otherwise return the number of
|      rows times number of columns if DataFrame.
|
|      See Also
|      --------
|      ndarray.size : Number of elements in the array.
|
|      Examples
|      --------
|      >>> s = pd.Series({'a': 1, 'b': 2, 'c': 3})
|      >>> s.size
|      3
|
|      >>> df = pd.DataFrame({'col1': [1, 2], 'col2': [3, 4]})
|      >>> df.size
|      4
|
|  ----------------------------------------------------------------------
|  Data and other attributes inherited from pandas.core.generic.NDFrame:
|
|  __array_priority__ = 1000
|
|  ----------------------------------------------------------------------
|  Methods inherited from pandas.core.base.PandasObject:
|
|  __sizeof__(self) -> 'int'
|      Generates the total memory usage for an object that returns
|      either a value or Series of values
|
|  ----------------------------------------------------------------------
|  Methods inherited from pandas.core.accessor.DirNamesMixin:
|
|  __dir__(self) -> 'list[str]'
|      Provide method name lookup and completion.
|
|      Notes
|      -----
|      Only provide 'public' methods.
|
|  ----------------------------------------------------------------------
|  Data descriptors inherited from pandas.core.accessor.DirNamesMixin:
|
|  __dict__
|      dictionary for instance variables (if defined)
|
|  __weakref__
|      list of weak references to the object (if defined)
|
|  ----------------------------------------------------------------------
|  Data descriptors inherited from pandas.core.indexing.IndexingMixin:
|
|  at
|      Access a single value for a row/column label pair.
|
|      Similar to ``loc``, in that both provide label-based lookups. Use
|      ``at`` if you only need to get or set a single value in a DataFrame
|      or Series.
|
|      Raises
|      ------
|      KeyError
|          If 'label' does not exist in DataFrame.
|
|      See Also
|      --------
|      DataFrame.iat : Access a single value for a row/column pair by integer
|          position.
|      DataFrame.loc : Access a group of rows and columns by label(s).
|      Series.at : Access a single value using a label.
|
|      Examples
|      --------
|      >>> df = pd.DataFrame([[0, 2, 3], [0, 4, 1], [10, 20, 30]],
|      ...                   index=[4, 5, 6], columns=['A', 'B', 'C'])
|      >>> df
|          A   B   C
|      4   0   2   3
|      5   0   4   1
|      6  10  20  30
|
|      Get value at specified row/column pair
|
|      >>> df.at[4, 'B']
|      2
|
|      Set value at specified row/column pair
|
|      >>> df.at[4, 'B'] = 10
|      >>> df.at[4, 'B']
|      10
```

```
|
|      Get value within a Series
|
|      >>> df.loc[5].at['B']
|      4
|
|  iat
|      Access a single value for a row/column pair by integer position.
|
|      Similar to ``iloc``, in that both provide integer-based lookups. Use
|      ``iat`` if you only need to get or set a single value in a DataFrame
|      or Series.
|
|      Raises
|      ------
|      IndexError
|          When integer position is out of bounds.
|
|      See Also
|      --------
|      DataFrame.at : Access a single value for a row/column label pair.
|      DataFrame.loc : Access a group of rows and columns by label(s).
|      DataFrame.iloc : Access a group of rows and columns by integer position(s).
|
|      Examples
|      --------
|      >>> df = pd.DataFrame([[0, 2, 3], [0, 4, 1], [10, 20, 30]],
|      ...                   columns=['A', 'B', 'C'])
|      >>> df
|          A   B   C
|      0   0   2   3
|      1   0   4   1
|      2  10  20  30
|
|      Get value at specified row/column pair
|
|      >>> df.iat[1, 2]
|      1
|
|      Set value at specified row/column pair
|
|      >>> df.iat[1, 2] = 10
|      >>> df.iat[1, 2]
|      10
|
|      Get value within a series
|
|      >>> df.loc[0].iat[1]
|      2
|
|  iloc
|      Purely integer-location based indexing for selection by position.
|
|      ``.iloc[]`` is primarily integer position based (from ``0`` to
|      ``length-1`` of the axis), but may also be used with a boolean
|      array.
|
|      Allowed inputs are:
|
|      - An integer, e.g. ``5``.
|      - A list or array of integers, e.g. ``[4, 3, 0]``.
|      - A slice object with ints, e.g. ``1:7``.
|      - A boolean array.
|      - A ``callable`` function with one argument (the calling Series or
|        DataFrame) and that returns valid output for indexing (one of the above).
|        This is useful in method chains, when you don't have a reference to the
|        calling object, but would like to base your selection on some value.
|
|      ``.iloc`` will raise ``IndexError`` if a requested indexer is
|      out-of-bounds, except *slice* indexers which allow out-of-bounds
|      indexing (this conforms with python/numpy *slice* semantics).
|
|      See more at :ref:`Selection by Position <indexing.integer>`.
|
|      See Also
|      --------
|      DataFrame.iat : Fast integer location scalar accessor.
|      DataFrame.loc : Purely label-location based indexer for selection by label.
|      Series.iloc : Purely integer-location based indexing for
|                     selection by position.
|
|      Examples
|      --------
|      >>> mydict = [{'a': 1, 'b': 2, 'c': 3, 'd': 4},
|      ...           {'a': 100, 'b': 200, 'c': 300, 'd': 400},
|      ...           {'a': 1000, 'b': 2000, 'c': 3000, 'd': 4000 }]
|      >>> df = pd.DataFrame(mydict)
|      >>> df
|            a     b     c     d
|      0     1     2     3     4
|      1   100   200   300   400
|      2  1000  2000  3000  4000
|
|      **Indexing just the rows**
|
|      With a scalar integer.
|
|      >>> type(df.iloc[0])
|      <class 'pandas.core.series.Series'>
|      >>> df.iloc[0]
|      a    1
|      b    2
|      c    3
|      d    4
|      Name: 0, dtype: int64
|
|      With a list of integers.
|
|      >>> df.iloc[[0]]
|         a  b  c  d
|      0  1  2  3  4
|      >>> type(df.iloc[[0]])
|      <class 'pandas.core.frame.DataFrame'>
|
|      >>> df.iloc[[0, 1]]
|           a    b    c    d
|      0    1    2    3    4
|      1  100  200  300  400
|
|      With a `slice` object.
|
|      >>> df.iloc[:3]
|            a     b     c     d
|      0     1     2     3     4
|      1   100   200   300   400
|      2  1000  2000  3000  4000
|
|      With a boolean mask the same length as the index.
|
|      >>> df.iloc[[True, False, True]]
|            a     b     c     d
|      0     1     2     3     4
|      2  1000  2000  3000  4000
|
|      With a callable, useful in method chains. The `x` passed
|      to the ``lambda`` is the DataFrame being sliced. This selects
|      the rows whose index label even.
|
|      >>> df.iloc[lambda x: x.index % 2 == 0]
|            a     b     c     d
|      0     1     2     3     4
|      2  1000  2000  3000  4000
|
|      **Indexing both axes**
|
|      You can mix the indexer types for the index and columns. Use ``:`` to
|      select the entire axis.
|
|      With scalar integers.
|
|      >>> df.iloc[0, 1]
|      2
|
|      With lists of integers.
|
|      >>> df.iloc[[0, 2], [1, 3]]
|            b     d
|      0     2     4
|      2  2000  4000
|
|      With `slice` objects.
|
|      >>> df.iloc[1:3, 0:3]
|            a     b     c
|      1   100   200   300
|      2  1000  2000  3000
|
|      With a boolean array whose length matches the columns.
|
|      >>> df.iloc[:, [True, False, True, False]]
|            a     c
|      0     1     3
|      1   100   300
|      2  1000  3000
```

```
 |
 |      With a callable function that expects the Series or DataFrame.
 |
 |      >>> df.iloc[:, lambda df: [0, 2]]
 |            a     c
 |      0     1     3
 |      1   100   300
 |      2  1000  3000
 |
 |  loc
 |      Access a group of rows and columns by label(s) or a boolean array.
 |
 |      ``.loc[]`` is primarily label based, but may also be used with a
 |      boolean array.
 |
 |      Allowed inputs are:
 |
 |      - A single label, e.g. ``5`` or ``'a'``, (note that ``5`` is
 |        interpreted as a *label* of the index, and **never** as an
 |        integer position along the index).
 |      - A list or array of labels, e.g. ``['a', 'b', 'c']``.
 |      - A slice object with labels, e.g. ``'a':'f'``.
 |
 |        .. warning:: Note that contrary to usual python slices, **both** the
 |            start and the stop are included
 |
 |      - A boolean array of the same length as the axis being sliced,
 |        e.g. ``[True, False, True]``.
 |      - An alignable boolean Series. The index of the key will be aligned before
 |        masking.
 |      - An alignable Index. The Index of the returned selection will be the input.
 |      - A ``callable`` function with one argument (the calling Series or
 |        DataFrame) and that returns valid output for indexing (one of the above)
 |
 |      See more at :ref:`Selection by Label <indexing.label>`.
 |
 |      Raises
 |      ------
 |      KeyError
 |          If any items are not found.
 |      IndexingError
 |          If an indexed key is passed and its index is unalignable to the frame
index.
 |
 |      See Also
 |      --------
 |      DataFrame.at : Access a single value for a row/column label pair.
 |      DataFrame.iloc : Access group of rows and columns by integer position(s).
 |      DataFrame.xs : Returns a cross-section (row(s) or column(s)) from the
 |          Series/DataFrame.
 |      Series.loc : Access group of values using labels.
 |
 |      Examples
 |      --------
 |      **Getting values**
 |
 |      >>> df = pd.DataFrame([[1, 2], [4, 5], [7, 8]],
 |      ...      index=['cobra', 'viper', 'sidewinder'],
 |      ...      columns=['max_speed', 'shield'])
 |      >>> df
 |                  max_speed  shield
 |      cobra               1       2
 |      viper               4       5
 |      sidewinder          7       8
 |
 |      Single label. Note this returns the row as a Series.
 |
 |      >>> df.loc['viper']
 |      max_speed    4
 |      shield       5
 |      Name: viper, dtype: int64
 |
 |      List of labels. Note using ``[[]]`` returns a DataFrame.
 |
 |      >>> df.loc[['viper', 'sidewinder']]
 |                  max_speed  shield
 |      viper               4       5
 |      sidewinder          7       8
 |
 |      Single label for row and column
 |
 |      >>> df.loc['cobra', 'shield']
 |      2
 |
 |      Slice with labels for row and single label for column. As mentioned
 |      above, note that both the start and stop of the slice are included.
 |
 |      >>> df.loc['cobra':'viper', 'max_speed']
 |      cobra    1
 |      viper    4
 |      Name: max_speed, dtype: int64
 |
 |      Boolean list with the same length as the row axis
 |
 |      >>> df.loc[[False, False, True]]
 |                  max_speed  shield
 |      sidewinder          7       8
 |
 |      Alignable boolean Series:
 |
 |      >>> df.loc[pd.Series([False, True, False],
 |      ...        index=['viper', 'sidewinder', 'cobra'])]
 |                  max_speed  shield
 |      sidewinder          7       8
 |
 |      Index (same behavior as ``df.reindex``)
 |
 |      >>> df.loc[pd.Index(["cobra", "viper"], name="foo")]
 |             max_speed  shield
 |      foo
 |      cobra          1       2
 |      viper          4       5
 |
 |      Conditional that returns a boolean Series
 |
 |      >>> df.loc[df['shield'] > 6]
 |                  max_speed  shield
 |      sidewinder          7       8
 |
 |      Conditional that returns a boolean Series with column labels specified
 |
 |      >>> df.loc[df['shield'] > 6, ['max_speed']]
 |                  max_speed
 |      sidewinder          7
 |
 |      Callable that returns a boolean Series
 |
 |      >>> df.loc[lambda df: df['shield'] == 8]
 |                  max_speed  shield
 |      sidewinder          7       8
 |
 |      **Setting values**
 |
 |      Set value for all items matching the list of labels
 |
 |      >>> df.loc[['viper', 'sidewinder'], ['shield']] = 50
 |      >>> df
 |                  max_speed  shield
 |      cobra               1       2
 |      viper               4      50
 |      sidewinder          7      50
 |
 |      Set value for an entire row
 |
 |      >>> df.loc['cobra'] = 10
 |      >>> df
 |                  max_speed  shield
 |      cobra              10      10
 |      viper               4      50
 |      sidewinder          7      50
 |
 |      Set value for an entire column
 |
 |      >>> df.loc[:, 'max_speed'] = 30
 |      >>> df
 |                  max_speed  shield
 |      cobra              30      10
 |      viper              30      50
 |      sidewinder         30      50
 |
 |      Set value for rows matching callable condition
 |
 |      >>> df.loc[df['shield'] > 35] = 0
 |      >>> df
 |                  max_speed  shield
 |      cobra              30      10
 |      viper               0       0
 |      sidewinder          0       0
 |
 |      **Getting values on a DataFrame with an index that has integer labels**
 |
 |      Another example using integers for the index
 |
 |      >>> df = pd.DataFrame([[1, 2], [4, 5], [7, 8]],
 |      ...      index=[7, 8, 9], columns=['max_speed', 'shield'])
 |      >>> df
 |         max_speed  shield
 |      7          1       2
 |      8          4       5
 |      9          7       8
 |
 |      Slice with integer labels for rows. As mentioned above, note that both
 |      the start and stop of the slice are included.
 |
 |      >>> df.loc[7:9]
 |         max_speed  shield
 |      7          1       2
 |      8          4       5
 |      9          7       8
 |
 |      **Getting values with a MultiIndex**
 |
 |      A number of examples using a DataFrame with a MultiIndex
 |
 |      >>> tuples = [
 |      ...    ('cobra', 'mark i'), ('cobra', 'mark ii'),
 |      ...    ('sidewinder', 'mark i'), ('sidewinder', 'mark ii'),
 |      ...    ('viper', 'mark ii'), ('viper', 'mark iii')
 |      ... ]
 |      >>> index = pd.MultiIndex.from_tuples(tuples)
 |      >>> values = [[12, 2], [0, 4], [10, 20],
 |      ...         [1, 4], [7, 1], [16, 36]]
 |      >>> df = pd.DataFrame(values, columns=['max_speed', 'shield'], index=index)
 |      >>> df
 |                           max_speed  shield
 |      cobra      mark i           12       2
 |                 mark ii           0       4
 |      sidewinder mark i           10      20
 |                 mark ii           1       4
 |      viper      mark ii           7       1
 |                 mark iii         16      36
 |
 |      Single label. Note this returns a DataFrame with a single index.
 |
 |      >>> df.loc['cobra']
 |               max_speed  shield
 |      mark i          12       2
 |      mark ii          0       4
 |
 |      Single index tuple. Note this returns a Series.
 |
 |      >>> df.loc[('cobra', 'mark ii')]
 |      max_speed    0
 |      shield       4
 |      Name: (cobra, mark ii), dtype: int64
 |
 |      Single label for row and column. Similar to passing in a tuple, this
 |      returns a Series.
 |
 |      >>> df.loc['cobra', 'mark i']
 |      max_speed    12
 |      shield        2
 |      Name: (cobra, mark i), dtype: int64
 |
 |      Single tuple. Note using ``[[]]`` returns a DataFrame.
 |
 |      >>> df.loc[[('cobra', 'mark ii')]]
 |                     max_speed  shield
 |      cobra mark ii          0       4
 |
 |      Single tuple for the index with a single label for the column
 |
 |      >>> df.loc[('cobra', 'mark i'), 'shield']
 |      2
 |
 |      Slice from index tuple to single label
 |
 |      >>> df.loc[('cobra', 'mark i'):'viper']
 |                           max_speed  shield
 |      cobra      mark i           12       2
 |                 mark ii           0       4
 |      sidewinder mark i           10      20
 |                 mark ii           1       4
 |      viper      mark ii           7       1
 |                 mark iii         16      36
 |
 |      Slice from index tuple to index tuple
 |
 |      >>> df.loc[('cobra', 'mark i'):('viper', 'mark ii')]
 |                          max_speed  shield
 |      cobra      mark i          12       2
 |                  mark ii          0       4
 |      sidewinder mark i          10      20
 |                  mark ii          1       4
 |      viper      mark ii          7       1
 |
 |  ----------------------------------------------------------------------
 |  Methods inherited from pandas.core.arraylike.OpsMixin:
 |
 |  __add__(self, other)
 |
 |  __and__(self, other)
 |
 |  __eq__(self, other)
 |      Return self==value.
 |
 |  __floordiv__(self, other)
 |
 |  __ge__(self, other)
 |      Return self>=value.
 |
 |  __gt__(self, other)
 |      Return self>value.
 |
 |  __le__(self, other)
 |      Return self<=value.
 |
 |  __lt__(self, other)
 |      Return self<value.
 |
 |  __mod__(self, other)
 |
 |  __mul__(self, other)
 |
 |  __ne__(self, other)
 |      Return self!=value.
 |
 |  __or__(self, other)
 |
 |  __pow__(self, other)
 |
 |  __radd__(self, other)
 |
 |  __rand__(self, other)
 |
 |  __rfloordiv__(self, other)
 |
 |  __rmod__(self, other)
 |
 |  __rmul__(self, other)
 |
 |  __ror__(self, other)
 |
 |  __rpow__(self, other)
 |
 |  __rsub__(self, other)
 |
 |  __rtruediv__(self, other)
 |
 |  __rxor__(self, other)
 |
 |  __sub__(self, other)
 |
 |  __truediv__(self, other)
 |
 |  __xor__(self, other)
 |
 |  ----------------------------------------------------------------------
 |  Data and other attributes inherited from pandas.core.arraylike.OpsMixin:
 |
 |  __hash__ = None
```

> **Try it yourself!**
>
> Make a list of the shape of all of the tables on the syllabus Achievements page.

```
achievements_url =
'https://rhodyprog4ds.github.io/BrownFall21/syllabus/achievements.html'
```

```
[(14, 3), (15, 5), (15, 15), (15, 6)]
```

This solution uses a list comprehension which allows us to compress a loop. It’s equivalent to the following with a for loop

```
[(14, 3), (15, 5), (15, 15), (15, 6)]
```

5.6. Lambdas and Dictionaries for switching

Note

Python does have ternary operators but the dictionary is a more common way to achieve this and goal and more common patterns are better for readability

What if we want to print out the first column for the dataFrame if it has more than 3 columns and the whole thing if it has 3 or less columns?

Two ways of writing a function

```python
# typical way
def first_col_f(d):
    return d[d.columns[0]]

first_col_l = lambda d: d[d.columns[0]]

first_col_f(help_df) == first_col_l(help_df)
```

```
0    True
1    True
2    True
3    True
4    True
5    True
Name: Day, dtype: bool
```

We can put functions in dictionaries, or even define a lambda right in the dictionary.

```python
df_display = {True: lambda d: d[d.columns[0]],
       False: lambda d: d}

for df in pd.read_html(achievements_url):
    _, n_cols = df.shape
    print(df_display[n_cols>3](df))
```

```
   Unnamed: 0_level_0                                             topics  \
                 week                                 Unnamed: 1_level_1
0                   1                             [admin, python review]
1                   2                        Loading data, Python review
2                   3                          Exploratory Data Analysis
3                   4                                      Data Cleaning
4                   5                      Databases, Merging DataFrames
5                   6  Modeling, Naive Bayes, classification performa...
6                   7                   decision trees, cross validation
7                   8                                         Regression
8                   9                                         Clustering
9                  10                              SVM, parameter tuning
10                 11                              KNN, Model comparison
11                 12                                      Text Analysis
12                 13                                    Images Analysis
13                 14                                      Deep Learning

                               skills
                   Unnamed: 2_level_1
0                             process
1        [access, prepare, summarize]
2              [summarize, visualize]
3     [prepare, summarize, visualize]
4      [access, construct, summarize]
5          [classification, evaluate]
6          [classification, evaluate]
7              [regression, evaluate]
8              [clustering, evaluate]
9                   [optimize, tools]
10                   [compare, tools]
11                     [unstructured]
12              [unstructured, tools]
13                   [tools, compare]
0             python
1            process
2             access
3          construct
4          summarize
5          visualize
6            prepare
7     classification
8         regression
9         clustering
10          evaluate
11          optimize
12           compare
13      unstructured
14          workflow
Name: (Unnamed: 0_level_0, keyword), dtype: object
0             python
1            process
2             access
3          construct
4          summarize
5          visualize
6            prepare
7     classification
8         regression
9         clustering
10          evaluate
11          optimize
12           compare
13      unstructured
14          workflow
Name: (Unnamed: 0_level_0, keyword), dtype: object
0             python
1            process
2             access
3          construct
4          summarize
5          visualize
6            prepare
7     classification
8         regression
9         clustering
10          evaluate
11          optimize
12           compare
13      unstructured
14          workflow
Name: (Unnamed: 0_level_0, keyword), dtype: object
```

> **Try it Yourself!**
>
> What does the _ do?

5.7. Questions after class

> **Note**
>
> add a question with a pull request; earn 1-2 ram tokens for submitting a question with the answer (with sources)

5.8. More Practice

1. What type is the shape of a pandas.DataFrame?
2. use a list comprehension to create a list that you could use as column names for data that consists of N measurements. Set N=5 for now, but you suspect that the number might change.

1. create a list of items with different types, then Create a dictionary with the types as keys using a dictionary comprehension. Dictionary comprehensions are similar to list comprehensions, in their form.

```
  File "/tmp/ipykernel_1694/1953963119.py", line 1
    about_prof_brown_dict = {type(fact):fact} for fact in about_prof_brown_list}
                                                ^
SyntaxError: invalid syntax
```

1. Create a lambda funcation to print return the first 2 rows of a data frame

1. Portfolio Setup, Data Science, and Python

Due: 2020-09-12

1.1. Objective & Evaluation

This assignment is an opportunity to earn level 2 achievements for the process and python and confirm that you have all of your tools setup, including your portfolio.

1.2. To Do

> **Important**
>
> If you have trouble, check the GitHub FAQ on the left before e-mailing

> **Note**
>
> If you get stuck on any of this after accepting the assignment and creating a repository, you can create an issue on your repository, describing what you're stuck on and tag us: @rhodypro4dg/fall21instructors
>
> To do this click Issues at the top, the green “New Issue” button and then type away.

~~~
```{warning}
If you have trouble with the (*)d steps, don't worry, we can help work around these later. To help
us out, document the errors as bugs on your repository.
```
~~~

Your task is to:

1. Install required software from the Tools & Resource page
2. Create your portfolio, by [accepting the assignment](#)
3. Learn about your portfolio from the README file on your repository.
4. edit _config.yml to set your name as author and change the logo if you wish
5. Fill in about/index.md with information about yourself(not evaluated, but useful) and your own definition of data science (graded for **level 1 process**)

6. (*) Install some additional python packages with: `pip install pip install -r requirements.txt` (this is a python operation, so use anaconda prompt on Windows, if the pip version doesn't work, try it with conda: `conda install --file requirements.txt`) form inside the portfolio folder
7. (*) Configure precommit to help keep your repo clean with `pre-commit install`. If this step doesn't work, see the portfolio README under "Using your Jupyter Book Portfolio"
8. Add a Jupyter notebook called `grading.ipynb` to the `about` folder and write a function that computes a grade for this course, with the following docstring. Include:
    - a Markdown cell with a heading
    - your function called `compute_grade`
    - three calls to your function that verify it returns the correct value for different number of badges that produce at three different letter grades.
    - a basic function that uses conditionals in python will earn **level 1 python**
    - to earn **level 2 python** use pythonic code to write a loop that tests your function's correctness, by iterating over a list or dictionary. Remember you will have many chances to earn level 2 achievement in python
9. Add the line `  - file: about/grading` in your `_toc.yml` file.

> **Important**
>
> remember to add, commit, and push your changes so we can see them

```
    '''
    Computes a grade for CSC/DSP310 from numbers of achievements at each level

    Parameters:
    ------------
    num_level1 : int
      number of level 1 achievements earned
    num_level2 : int
      number of level 2 achievements earned
    num_level3 : int
      number of level 3 achievements earned

    Returns:
    --------
    letter_grade : string
      letter grade with possible modifier (+/-)
    '''
```

Here are some sample tests you could run to confirm that your function works correctly:

```
assert compute_grade(15,15,15) == 'A'

assert compute_grade(15,15,13) == 'A-'

assert compute_grade(15,14,14) == 'B-'

assert compute_grade(14,14,14) == 'C-'

assert compute_grade(4,3,1) == 'D'

assert compute_grade(15,15,6) =='B+'
```

> **Warning**
>
> your function can have a different name than `compute_grade`, but make sure it's your function name, with those parameter values in your tests.

> **Note**
>
> when the value of the expression after `assert` is `True`, it will look like nothing happened. `assert` is used for testing

1.3. Submission Instructions

Create a Jupyter Notebook with your function in your portfolio folder commit and push the changes.

In your browser, view the `gh-pages` branch to see your compiled submission, as `portfolio.pdf` or by viewing your website.

There will be a pull request on your repository that is made by GitHub classroom, request a review from @rhodypro4dg/fall21instructors.

2. Assignment 2: Practicing Python and Accessing Data

due : 2020-09-21

2.1. Objective & Evaluation

This assignment is an opportunity to earn level 1 and 2 achievements in `python` and `access` and begin working toward level 1 for `summarize`. You can also earn level 1 for `process`.

In this assignment, you'll practice/ review python skills by manipulating datasets and extracting

Task	Skills (max level)
identify possible uses for data in a data science pipeline	[process (1)]
load data from one file format	[access (1)]
load data from at least two of (.csv, .tsv, .dat, database, .json)	[access (2)]
compare the data formats	[access (2)]
complete the assignment in python	python (1)]
use python data types (eg dictionaries) to prepare information about datasets	[python (2)]
use informative variable names, pythonic iteration, and other common PEP 8 conventions	[python (2)]
display DataFrame properties	[summarize (1)]

Table 2.1 practice python by manipulating data files, load datasets of different types

First, accept the assignment . It contains a notebook with some template structure (and will set you up for grading).

2.2. Find Datasets

Find 3 datasets of interest to you that are provided in at least two different file formats. Choose datasets that are not too big, so that they do not take more than a few second to load. At least one dataset, must have non numerical (eg string or boolean) data in at least 1 column.

In your notebook, create a markdown cell for each notebook that includes:

- heading of the dataset's name
- a link to where someone can learn about the dataset
- a 1-2 sentence summary of what the dataset contains and why it was collected
- 1-2 questions you would like to answer with that dataset.

2.3. Store them for loading

Create a list of dictionaries in `datasets.py`, so that there is one dictionary for each dataset with the url, a name, and what function should be used to load the data into a `pandas.DataFrame`.

2.4. Make a dataset about your datasets

Import the list fomr the `datasets` module you created in the step above. Then terate over the list of dictionaries, and:

1. save it to a local csv using the short name you provided for the dataset as the file name, without writing the index column to the file.
2. record attributes about the dataset as in the table below in a list of lists:
3. Use that to create a DataFrame with the following columns:

Tip

Urls are strings. The `string` class in python has a lot of helpful methods for manipulating strings, like `split`.

name	a short name for the dataset
source	a url to where you found the data
num_rows	number of rows in the dataset
num_columns	number of columns in the dataset
num_numerical	number of numerical variables in the dataset

Table 2.2 Meta Data Description of the DataFrame to build

2.5. Manipulate your datasets

For one dataset that includes nonnumerical data:

- display the heading and the last 4 rows
- make and display a new data frame with only the numerical columns (select these programmatically)

For any other dataset:

- display the heading and the first three rows
- display the datatype for each column
- Are there any variables where pandas may have read in the data as a datatype that's not what you expect (eg a numerical column mistaken for strings)? If so, investigate and try to figure out why.

For the third dataset:

- display the first 3 odd rows (eg 1,3,5) of the data for two columns of your choice

2.6. Exploring data files

For each dataset, in a separate section of your notebook titled `When things go wrong`:

- try reading in data with the wrong `read_` function and make notes about what happens.
- was the format that the data was provided in a good format? why or why not?
- try to read in the `.csv` file that's included in the template repository (), use the error messages you get to try to fix the file manualy (any text editor, including jupyter can edit a `.csv`), making notes about what changes you made in a markdown cell.

2.7. Thinking ahead

> ⚠ **Warning**
>
> his section is not required, but is intended to help you get started thinking about ideas for your portflio. If you complete it, we'll give your feedback to help shape your ideas to get to level 3 achievements. If you want to focus only on level 2 at this moment in time, feel free to skip this part.

1. When might you prefer one datatype over another?
2. How does PEP 8 standard code help you be collaborative?
3. Learn about Datasheets for Datasets eg this google scholar result How could something like this impact your work as a datascientist?

Portfolio Dates and Key Facts

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
/tmp/ipykernel_1719/311973014.py in <module>
      5 
      6 rubric_df = pd.concat([rubric_df,
----> 7                       assignment_dummies,
      8                       portfolio_dummies],axis=1)
      9 

NameError: name 'assignment_dummies' is not defined
```

This section of the site has a set of portfolio prompts and this page has instructions for portfolio submissions.

Starting in week 3 it is recommended that you spend some time each week working on items for your portfolio, that way when it's time to submit you only have a little bit to add before submission. The portfolio is your only chance to earn Level 3 achievements, however, if you have not earned a level 2 for any of the skills in a given check, you could earn level 2 then instead. The prompts provide a starting point, but remember that to earn achievements, you'll be evaluated by the rubric. You can see the full rubric for all portfolios in the syllabus. Your portfolio is also an opportunity to be creative, explore things, and answer your own questions that we haven't answered in class to dig deeper on the topics we're covering. Use the feedback you get on assignments to inspire your portfolio.

Each submission should include an introduction and a number of 'chapters'. The grade will be based on both that you demonstrate skills through your chapters that are inspired by the prompts and that your summary demonstrates that you *know* you learned the skills. See the formatting tips for advice on how to structure files.

On each chapter(for a file) of your portfolio, you should identify which skills by their keyword, you are applying.

You can view a (fake) example in this repository as a pdf or as a rendered website

Current

Check 1

The first portfolio check will be due October 15

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
/tmp/ipykernel_1719/2819809228.py in <module>
----> 1 portfolio_df[portfolio_df['P1']==1]

NameError: name 'portfolio_df' is not defined
```

Upcoming Checks

Check 2: November 12 Check 3: December 5 Check 4: December 20

Formatting Tips

> ⚠ **Warning**
>
> This is all based on you having accepted the portfolio assignment on github and having a cloned copy of the template. If you are not enrolled or the initial assignment has not been issued, you can view the template on GitHub

Your portfolio is a jupyter book. This means a few things:

- it uses myst markdown
- it will run and compile Jupyter notebooks

This page will cover a few basic tips.

Organization

The summary of for the `part` or whole submission, should match the skills to the chapters. Which prompt you're addressing is not important, the prompts are a *starting point* not the end goal of your portfolio.

Data Files

Also note that for your portfolio to build, you will have to:

- include the data files in the repository and use a relative path OR
- load via url

using a full local path(eg that starts with `///file:`) **will not work** and will render your portfolio unreadable.

Structure of plain markdown

Use a heading like this:

```
# Heading of page
## Heading 2
### Heading 3
```

in the file and it will appear in the sidebar.

You can also make text *italic* or **bold** with either `*asterics*` or `__underscores__` with `_one for italic_` or `**two for bold**` in either case

File Naming

It is best practice to name files without spaces. Each `chapter` or file should have a descriptive file name (`with_no_spaces`) and descriptive title for it.

Syncing markdown and ipynb files

If you have the precommit hook working, git will call a script and convert your notebook files from the ipynb format (which is json like) to Myst Markdown, which is more plain text with some header information. The markdown format works better with version control, largely because it doesn't contain the outputs.

If you don't get the precommit hook working, but you do get jupytext installed, you can set each file to sync.

Adding annotations with formatting or margin notes

You can either install jupytext and convert locally or upload /push a notebook to your repository and let GitHub convert. Then edit the .md file with a text editor of your choice. You can run by uploading if you don't have jupytext installed, or locally if you have installed jupytext or jupyterbook.

In your .md file use backticks to mark special content blocks

~~~
```{note}
Here is a note!
```
~~~

~~~
```{warning}
Here is a warning!
```
~~~

~~~
```{tip}
Here is a tip!
```
~~~

~~~
```{margin}
Here is a margin note!
```
~~~

For a complete list of options, see the `sphinx-book-theme` documentation.

Links

Markdown syntax for links

```
[text to show](path/or/url)
```

Configurations

Things like the menus and links at the top are controlled as settings, in `_config.yml`. The following are some things that you might change in your configuration file.

Show errors and continue

To show errors and continue running the rest, add the following to your configuration file:

```
# Execution settings
execute:
  allow_errors              : true
```

Using additional packages

You'll have to add any additional packages you use (beyond pandas and seaborn) to the `requirements.txt` file in your portfolio.

FAQ

This section will grow as questions are asked and new content is introduced to the site. You can submit questions:

- via e-mail to Dr. Brown (brownsarahm) or Beibhinn (beibhinn)
- via Prismia.chat during class
- by creating an issue

Syllabus FAQ

How much does assignment x, class participation, or a portfolio check weigh in my grade?

Can I submit this assignment late if …?

Git and GitHub FAQ

The content I added to my portfolio isn't in the pdf

My command line says I cannot use a password

My .ipynb file isn't showing in the staging area or didn't push

My portfolio won't compile

Help! I accidentally merged the Feedback Pull Request before my assignment was graded

Common Debugging Issues

Key Error

<bound method

Glossary

DataFrame
a data structure provided by pandas for tabular data in python.

git
a version control tool; it’s a fully open source and always free tool, that can be hosted by anyone or used without a host, locally only.

GitHub
a hosting service for git repositories

interpreter
the translator from human readable python code to something the computer can run. An interpreted language means you can work with python interactively

kernel
in the jupyter environment, the kernel is a language specific computational engine

PEP 8
Python Enhancement Proposal 8, the Style Guide for Python Code

repository
a project folder with tracking information in it in the form of a .git file

TraceBack
an error message in python that traces back from the line of code that had caused the exception back through all of the functions that called other functions to reach that line. This is sometimes call tracing back through the stack

General Tips and Resources

This section is for materials that are not specific to this course, but are likely useful. They are not generally required readings or installs, but are options or advice I provide frequently.

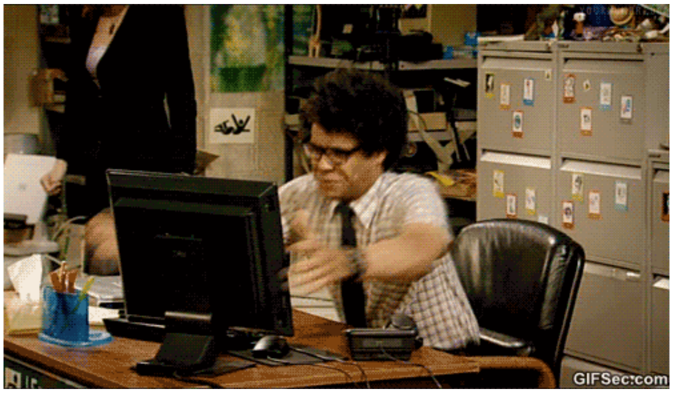
on email

- how to e-mail professors

How to Study in this class

This is a programming intensive course and it’s about data science. This course is designed to help you learn how to program for data science and in the process build general skills in programming and using data to understand the world. Learning two things at once is more complex. In this page, I break down how I expect learning to work for this class.

Remember the goal is to avoid this:



Why this way?

Learning to program requires iterative practice. It does not require memorizing all of the specific commands, but instead learning the basic patterns.
Using reference materials frequently is a built in part of programming, most languages have built in help as a part of the language for this reason.

A new book that might be of interest if you find programming classes hard is the Programmers Brain As of 2020-09-07, it is available for free by clicking on chapters at that linked table of contents section.

> ℹ **Where are your help tools?**
>
> In Python and Jupyter notebooks, what help tools do you have?

Learning in class

> ❗ **Important**
>
> My goal is to use class time so that you can be successful with *minimal frustration* while working outside of class time.

Programming requires both practical skills and abstract concepts. During class time, we will cover the practical aspects and introduce the basic concepts. You will get to see the basic practical details and real examples of debugging during class sessions. Learning to debug something you've never encountered before and setting up your programming envrionment, for example, are *high frustration* activities, when you're learning, because you don't know what you don't know. On the other hand, diving deeper into options and more complex applications of what you have already seen in class, while challenging, is something I'm confident that you can all be successful at with minimal frustration once you've seen basic ideas in class. My goal is that you can repeat the patterns and processes we use in class outside of class to complete assignments, while acknowledging that you will definitely have to look things up and read documentation outside of class.

Each class will open with some time to review what was covered in the last session before adding new material.

To get the most out of class sessions, you should have a laptop with you. During class you should be following along with Dr. Brown, typing and running the same code. You'll answer questions on Prismia chat, when you do so, you should try running necessary code to answer those questions. If you encounter errors, share them via prismia chat so that we can see and help you.

After class

After class, you should practice with the concepts introduced.
This means reviewing the notes: both yours from class and the annotated notes posted to the course website.
When you review the notes, you should be adding comments on tricky aspects of the code and narrative text between code blocks in markdown cells. While you review your notes and the annotated course notes, you should also read the documentation for new modules, libraries, or functions introduced that day.

In the annotated notes, there will often be extra questions or ideas on how to extend and practice the concepts. Try these out.

If you find anything hard to understand or unclear, write it down to bring to class the next day.

There will be additional drills posted to Prismia chat for you to try.

Assignments

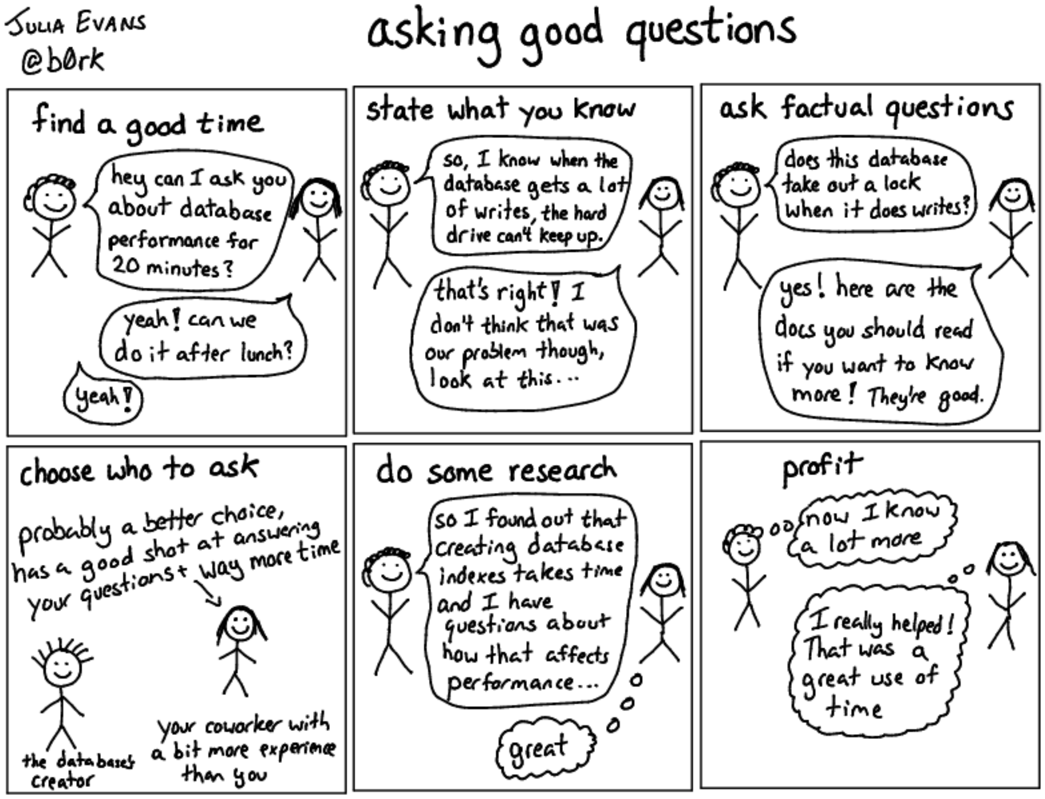
In assignments, you will be asked to practice with specific concepts at an intermediate level. Assignments will apply the concepts from class with minimal extensions. You will probably need to use help funcitons and read documentation to complete assignments, but mostly to look up things you saw in class and make minor variations. Most of what you need for assignments will be in the class notes, which is another reason to read them after class.

Portfolios

In portfolios, your goal is to extend and apply the concepts taught in class and practiced in assignments to solve more realistic problms. You may also reflect on your learning in order to demonstrate deep understanding. These will require significant reading beyond what we cover in class.

Getting Help with Programming

Asking Questions


JULIA EVANS
@b0rk
asking good questions
find a good time
hey can I ask you about database performance for 20 minutes?
yeah! can we do it after lunch?
yeah!
state what you know
so, I know when the database gets a lot of writes, the hard drive can't keep up.
that's right! I don't think that was our problem though, look at this...
ask factual questions
does this database take out a lock when it does writes?
yes! here are the docs you should read if you want to know more! They're good.
choose who to ask
probably a better choice, has a good shot at answering your questions + way more time
the database's creator
your coworker with a bit more experience than you
do some research
so I found out that creating database indexes takes time and I have questions about how that affects performance...
great
profit
now I know a lot more
I really helped! That was a great use of time

One of my favorite resources that describes how to ask good questions is this blog post by Julia Evans, a developer who writes comics about the things she learns in the course of her work and publisher of wizard zines.

Describing what you have so far

Stackoverflow is a common place for programmers to post and answer questions.
As such, they have written a good guide on creating a minimal, reproducible example.

Note

A fun version of this is rubber duck debugging

Creating a minimal reproducible example may even help you debug your own code, but if it does not, it will definitely make it easier for another person to understand what you have, what your goal is, and what’s working.

Understanding Errors

Error messages from the compiler are not always straight forward.

The TraceBack can be a really long list of errors that seem like they are not even from your code. It will trace back to all of the places that the error occurred. It is often about how you called the functions from a library, but the compiler cannot tell that.

To understand what the traceback is, how to read one, and common examples, see this post on Real Python.

One thing to try, is friendly traceback a python package that is designed to make that error message text more clear and help you figure out what to do next.

References on Python

- Course Text

Cheatsheet

Patterns and examples of how to use common tips in class

Axes

First build a small dataset that's just enough to display

```
data = [[1,0],[5,4],[1,4]]
df = pd.DataFrame(data = data,
  columns = ['A','B'])

df
```

	A	B
0	1	0
1	5	4
2	1	4

```
df.sum(axis=0)
```

```
A    7
B    8
dtype: int64
```

```
df.sum(axis=1)
```

```
0    1
1    9
2    5
dtype: int64
```

```
df.apply(sum,axis=0)
```

```
A    7
B    8
dtype: int64
```

```
df.apply(sum,axis=1)
```

```
0    1
1    9
2    5
dtype: int64
```

```
df['A'][1]
```

```
5
```

```
df.iloc[0][1]
```

```
0
```

Data Sources

Best for loading directly into a notebook

- Tidy Tuesday inside the folder for each year there is a README file with list of the datasets. These are .csv files
- Json Datasets
- National Center for Education Statistics Digest 2019 These data tables are available for download as excel and visible on the page.
- Lots of wikipedia pages have tables in them.

Requires some more work

- Stackoverflow Developer Survey This data comes with readme info all packaged together in a .zip. You'll need to unzip it first.
- Google Dataset Search
- Kaggle most Kaggle datasets will require you to download and unzip them first and then you can copy them into your repo folder.
- UCI Data Repository

Databases

- SQLite Databases

If you have others please share by creating a pull request or issue on this repo (from the GitHub logo at the top right, `suggest edit`).

By Professor Sarah M Brown