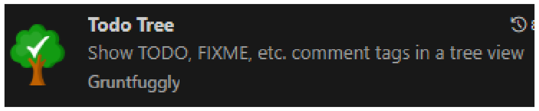
Commenting and DocStrings

By: Karl Johnson – karl.johnson@lambweston.com

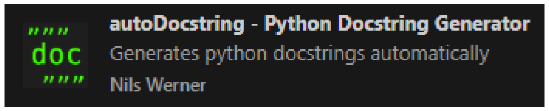
Extensions

I use VS Code as my general purpose IDE, and there are a few extensions that I use to help me with commenting and documentation. I'll discuss these in more detail later on, but in case you want to install them:

- Todo Tree

  
  Todo Tree
  Show TODO, FIXME, etc. comment tags in a tree view
  Gruntfuggly

  - Searches your comments for predefined tags like "TODO", "BUG", etc., and highlights them (inline and minimap).
  - Also adds an icon in the activity bar to view a structured tree, broken down by folder/file with clickable links to all the detected tags.
- AutoDocstring

  
  autoDocstring - Python Docstring Generator
  Generates python docstrings automatically
  Nils Werner

  - Generates docstring templates for your functions/classes in any of a number of docstring markdown formats.
  - Auto-fills arg names and types (if you remembered your type hints!) Does return type, too.

Without further ado…

Personal Comment Philosophy

Comments are written with two audiences in mind. The first audience is myself, as reminders for things like unusual coding patterns, new functionality, or logic that I found confusing in the moment that I am likely to be puzzled by when attempting to understand later.

The second audience is other developers. It is important to remember that someone else may need to parse your code later, or you might need them when debugging code that you've long-since forgotten how it was written, or why you chose to write it that way. Good candidates for these kinds of comments are the same as the first audience: describe unusual patterns, obscure logic, and basically anything that might confuse future reviewers. If it was enigmatic when you wrote it, it will probably be a challenge for the next developer to comprehend, too. This is especially useful when writing decidedly "pythonic" code. Python has a lot of neat tricks to make your code more elegant, but elegance often comes at the expense of readability.

Keep comments brief. Long-form descriptions are for docstrings and external documentation.

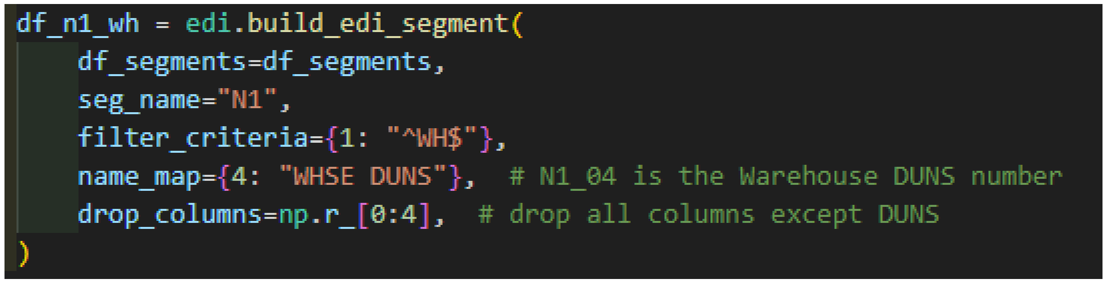
Inline comments should be limited to definitions and brief descriptions. Especially when commenting args in large function calls or objects in a collection. Think: "what is this thing (and what is it used for)?"

Block comments should be limited to describing the immediate action. They will often take the form of "[VERB] the [THING] (optional: so that [OUTCOME] happens.)"
Examples:

- `# set up dataframe to append to later`
- `# pop row 0 to remove headers`

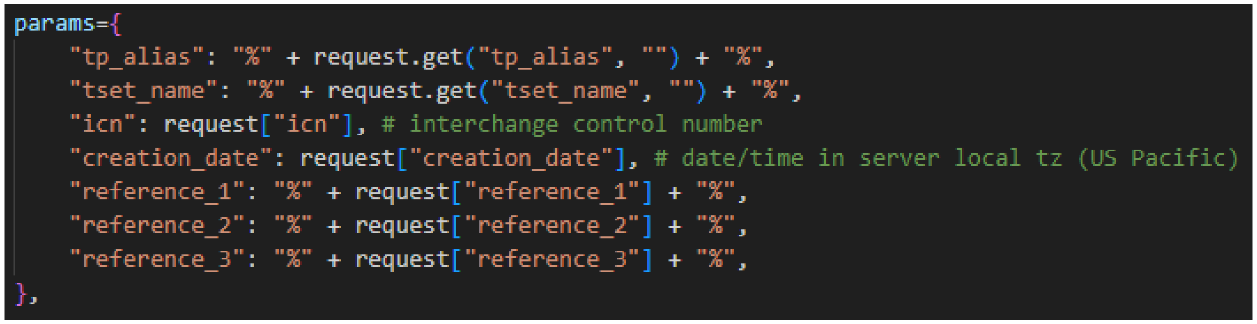
Examples of inline and block comments:

1. Inline:
   a. Short, pithy comments at the end of lines, helpful when documenting the purpose of multiple args or elements in a collection (dictionary, etc.)
      i. Args:



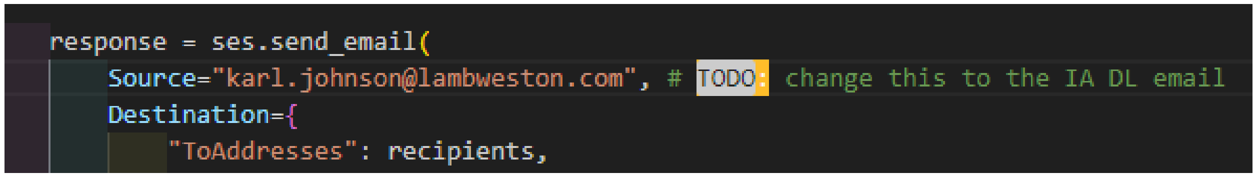
```python
df_n1_wh = edi.build_edi_segment(
    df_segments=df_segments,
    seg_name="N1",
    filter_criteria={1: "^WH$"},
    name_map={4: "WHSE DUNS"},  # N1_04 is the Warehouse DUNS number
    drop_columns=np.r_[0:4],  # drop all columns except DUNS
)
```

      ii. Collections:



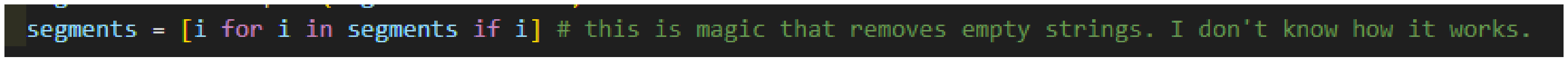
```python
params={
    "tp_alias": "%" + request.get("tp_alias", "") + "%",
    "tset_name": "%" + request.get("tset_name", "") + "%",
    "icn": request["icn"], # interchange control number
    "creation_date": request["creation_date"], # date/time in server local tz (US Pacific)
    "reference_1": "%" + request["reference_1"] + "%",
    "reference_2": "%" + request["reference_2"] + "%",
    "reference_3": "%" + request["reference_3"] + "%",
},
```

   b. Small To-Do's or reminders for obscure/cryptic lines of code (see: Todo Tree Extension):
      i. A reminder to remove/alter a line after testing:



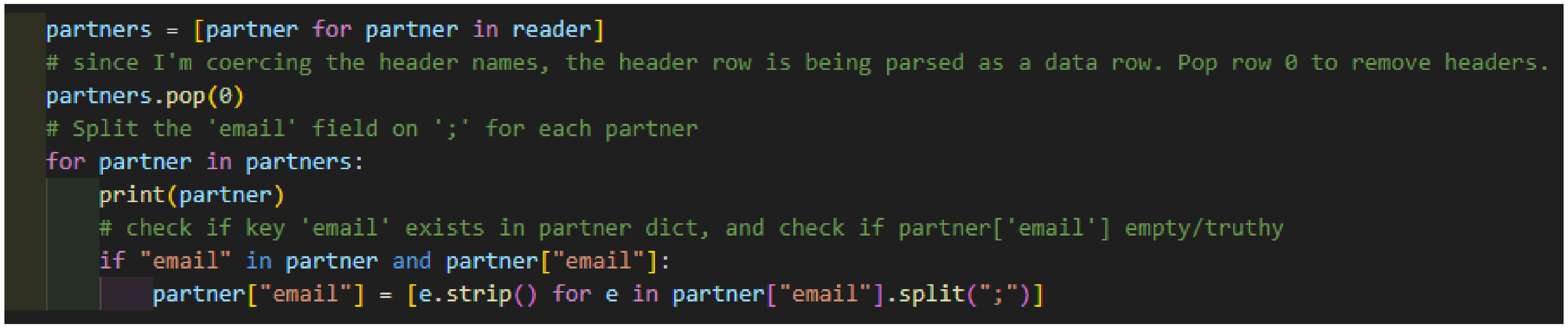
```python
response = ses.send_email(
    Source="karl.johnson@lambweston.com", # TODO: change this to the IA DL email
    Destination={
        "ToAddresses": recipients,
```

   ii. A reminder for a cryptic line of code:



```
segments = [i for i in segments if i] # this is magic that removes empty strings. I don't know how it works.
```

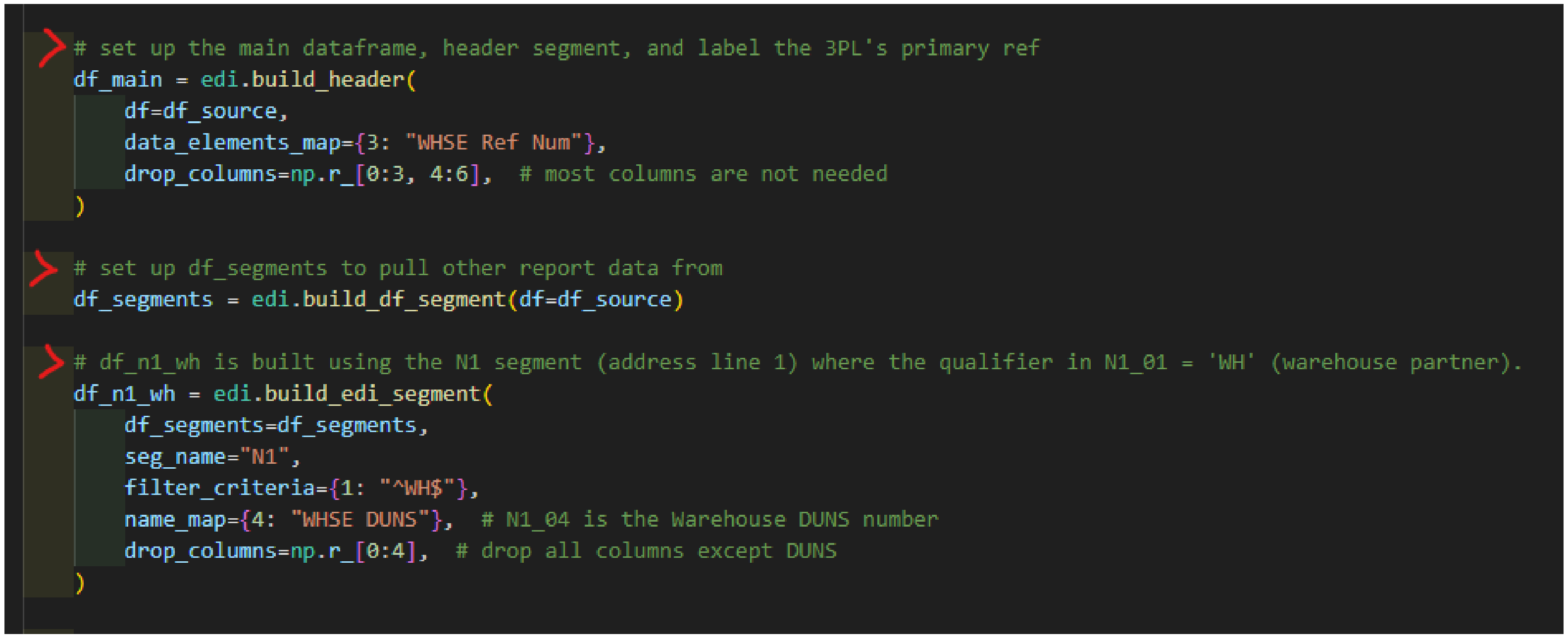
2. Block comments:
   a. Insert comments within large blocks of code describing the actions being taken at each major step.
      i. Describing the problem and the solution:



```
partners = [partner for partner in reader]
# since I'm coercing the header names, the header row is being parsed as a data row. Pop row 0 to remove headers.
partners.pop(0)
# Split the 'email' field on ';' for each partner
for partner in partners:
    print(partner)
    # check if key 'email' exists in partner dict, and check if partner['email'] empty/truthy
    if "email" in partner and partner["email"]:
        partner["email"] = [e.strip() for e in partner["email"].split(";")]
```

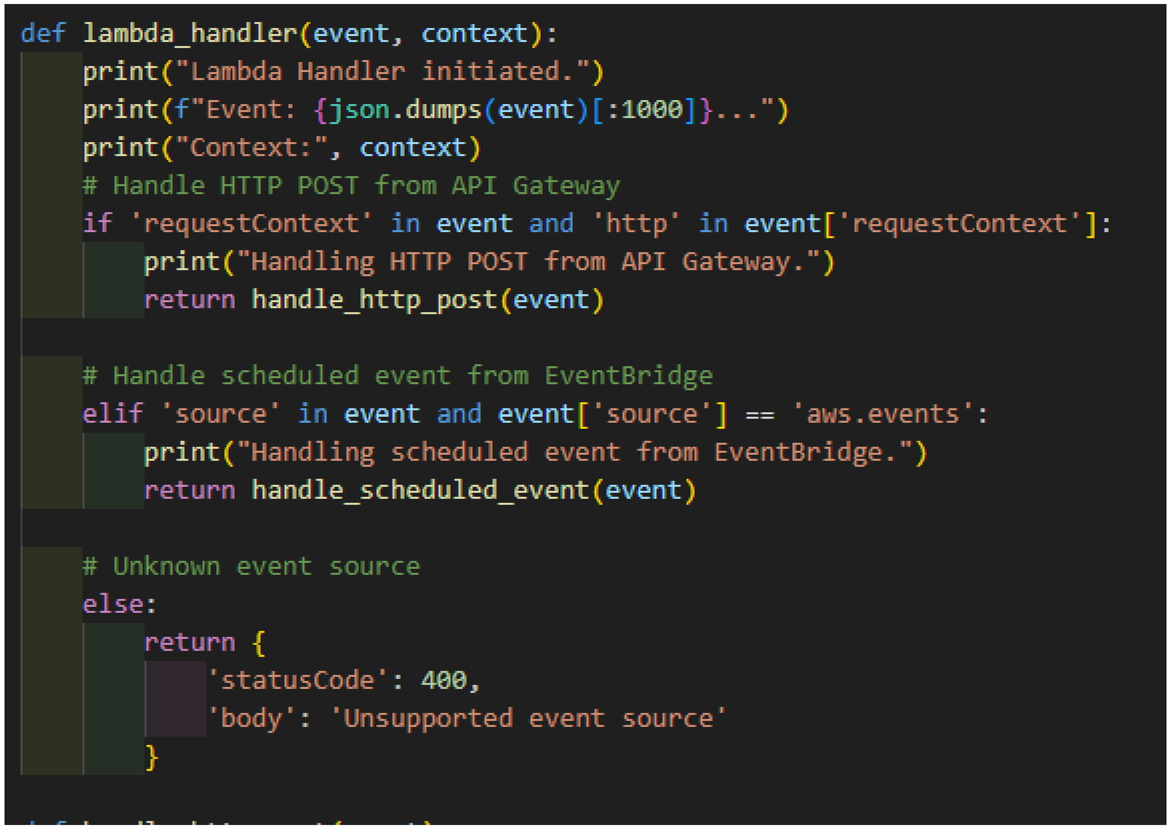
   b. Break up sections into digestible chunks, write a comment explaining what each section is responsible for:

i. Sections explaining various data processing steps:



```
# set up the main dataframe, header segment, and label the 3PL's primary ref
df_main = edi.build_header(
    df=df_source,
    data_elements_map={3: "WHSE Ref Num"},
    drop_columns=np.r_[0:3, 4:6],  # most columns are not needed
)

# set up df_segments to pull other report data from
df_segments = edi.build_df_segment(df=df_source)

# df_n1_wh is built using the N1 segment (address line 1) where the qualifier in N1_01 = 'WH' (warehouse partner).
df_n1_wh = edi.build_edi_segment(
    df_segments=df_segments,
    seg_name="N1",
    filter_criteria={1: "^WH$"},
    name_map={4: "WHSE DUNS"},  # N1_04 is the Warehouse DUNS number
    drop_columns=np.r_[0:4],  # drop all columns except DUNS
)
```

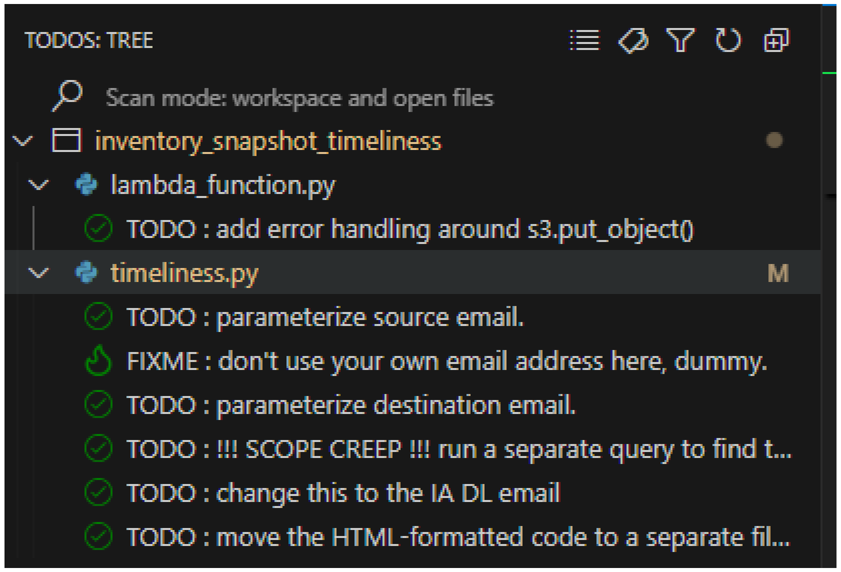
ii. If-elif-else block with comments for each path:



```python
def lambda_handler(event, context):
    print("Lambda Handler initiated.")
    print(f"Event: {json.dumps(event)[:1000]}...")
    print("Context:", context)
    # Handle HTTP POST from API Gateway
    if 'requestContext' in event and 'http' in event['requestContext']:
        print("Handling HTTP POST from API Gateway.")
        return handle_http_post(event)

    # Handle scheduled event from EventBridge
    elif 'source' in event and event['source'] == 'aws.events':
        print("Handling scheduled event from EventBridge.")
        return handle_scheduled_event(event)

    # Unknown event source
    else:
        return {
            'statusCode': 400,
            'body': 'Unsupported event source'
        }
```

Todo Tree Extension:

I use an extension in VS Code called “Todo Tree” (there are others like it, but I like this one for its simplicity). During early development phases on a project, I will write “# TODO:” comments all over my code, this allows me to make a note of a thought that I might want to explore later, or functionality I don’t have time to expand on at the moment. Then, when I have time to review, I open the ToDo Tree and can see all my ToDo’s organized by folder/file/type:



TODOS: TREE

Scan mode: workspace and open files

inventory_snapshot_timeliness

lambda_function.py

TODO : add error handling around s3.put_object()

timeliness.py M

TODO : parameterize source email.

FIXME : don't use your own email address here, dummy.

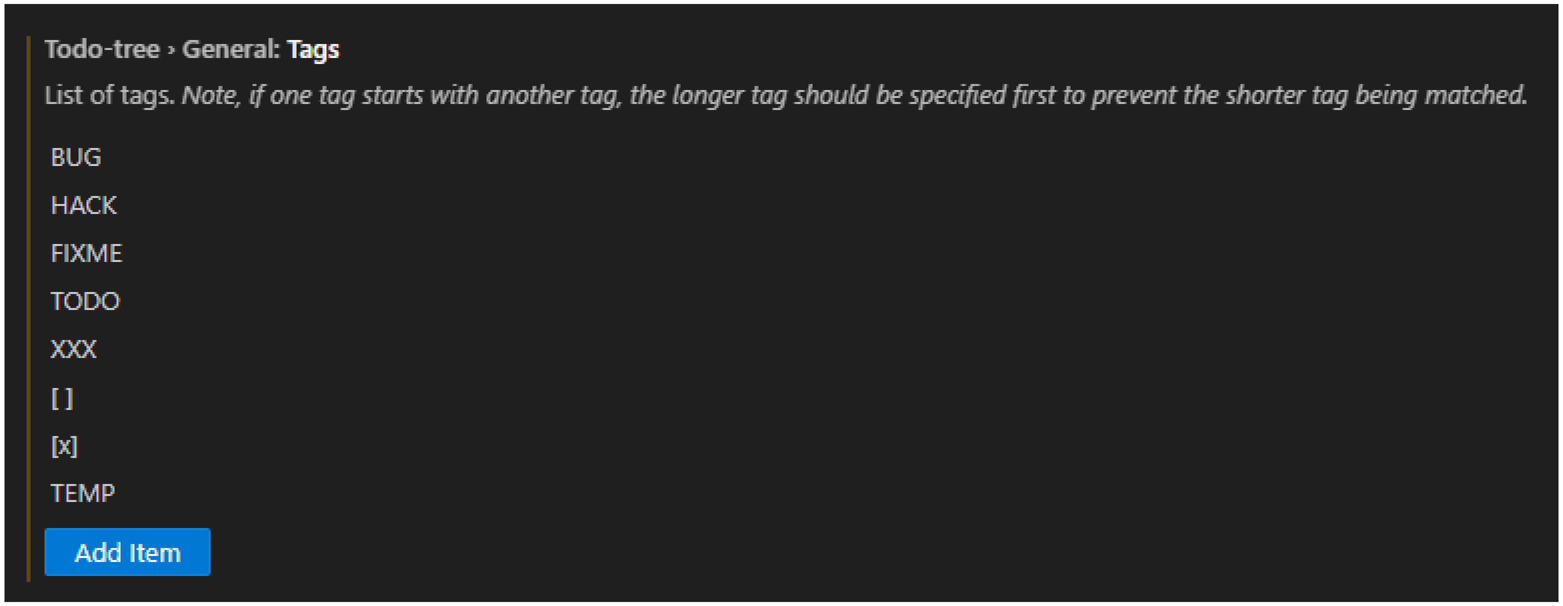
TODO : parameterize destination email.

TODO : !!! SCOPE CREEP !!! run a separate query to find t...

TODO : change this to the IA DL email

TODO : move the HTML-formatted code to a separate fil...

It also detects other types of tags (some extensions call these “code anchors”), the list of tags is configurable if you want to add more:


Todo-tree › General: Tags

List of tags. *Note, if one tag starts with another tag, the longer tag should be specified first to prevent the shorter tag being matched.*

BUG
HACK
FIXME
TODO
XXX
[ ]
[x]
TEMP

DocStrings

DocStrings are essential for keeping track of your many functions and classes in larger projects, but are of course also useful in smaller projects for debugging. They give you a chance to write your documentation in a consistent, formatted way. There are several flavors of DocString formatting that IDE's like VS Code will recognize. My personal favorite is Sphynx, which uses the ReStructuredText Documentation format (also called "RTD" or "ReST"). Sphynx is used for generating external documentation using your docstrings.

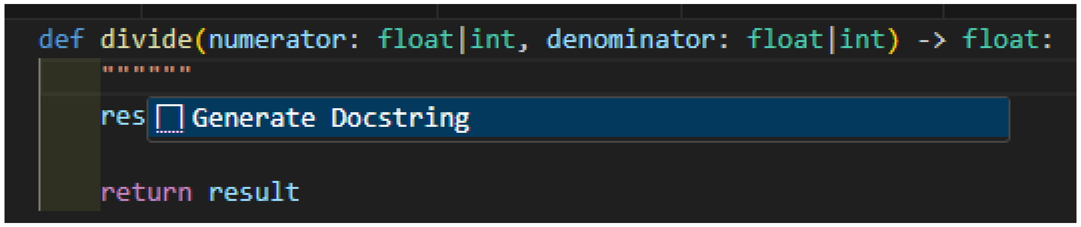
Online documentation for the ReST markdown format can be found here:

- reStructuredText – Primary ReST Syntax documentation site.
    - Quick reStructuredText – a cheatsheet with visual examples

- Sphynx - reStructuredText Primer – a primer written by the Sphynx team
- Python.org - reStructuredText markup – another primer written for the Python Developer's guide.
    - ReST was proposed as the de-facto standard for Python docstrings in PEP 287, and is widely used in the Python community
    - While not yet officially adopted, PEP 287 remains "active" and has not been replaced by subsequent PEP publications

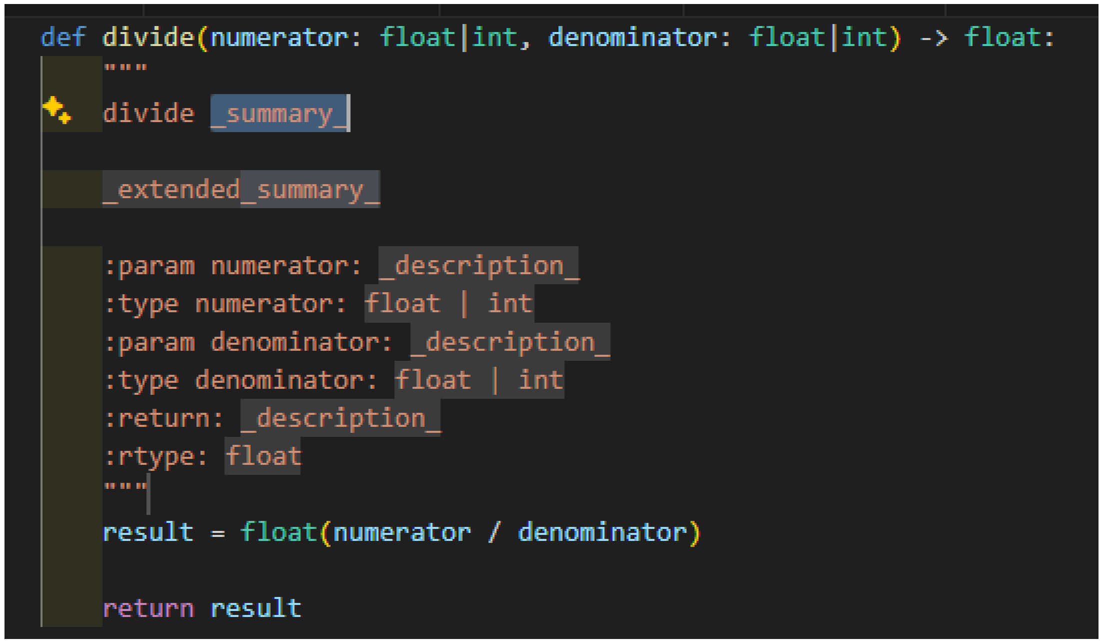
AutoDocstring

There is an extension in VSCode for generating Python DocStrings called "autoDocstring" that I highly recommend. It can be configured to use one of several different flavors of docstring markdown. After the beginning of a new function or class, typing triple quotes will begin a docstring. AutoDocstring detects this and will prompt you to automatically insert a generated block of text:



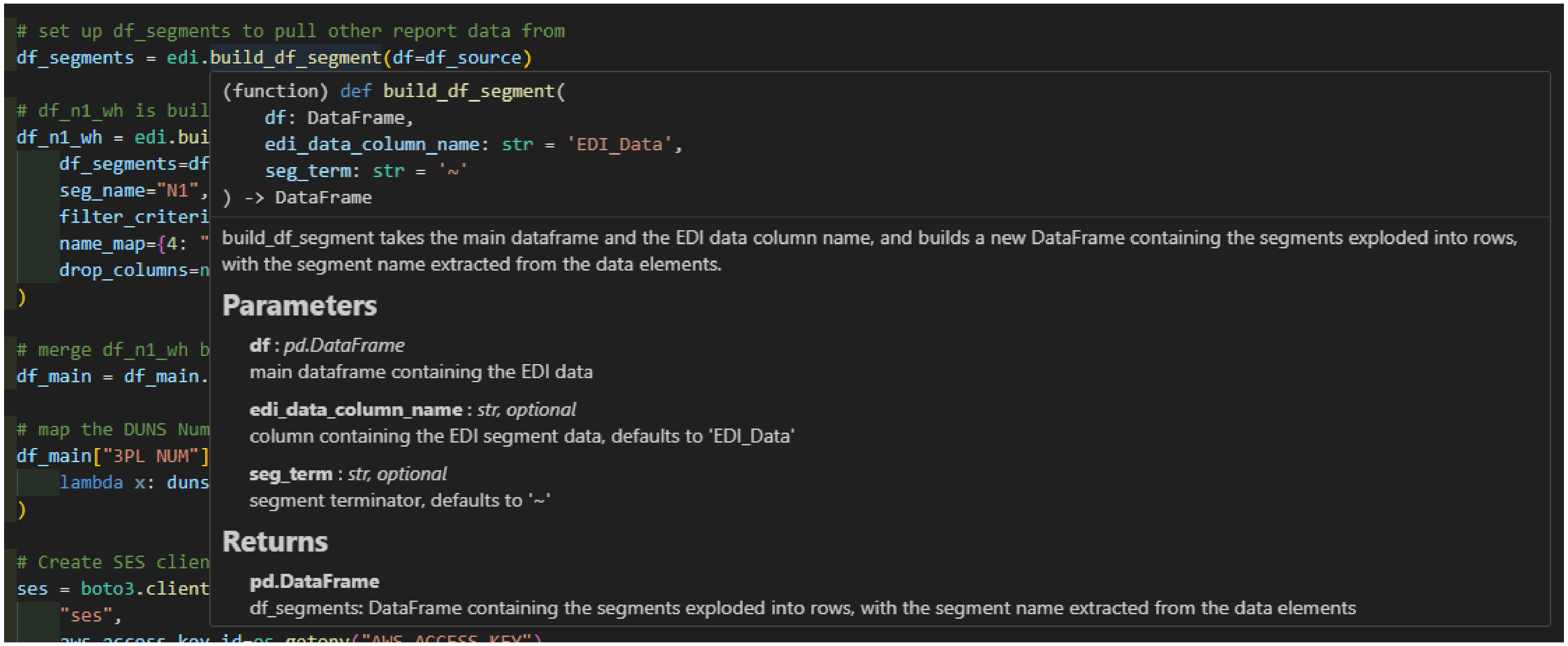
```
def divide(numerator: float|int, denominator: float|int) -> float:
    """"""
    res

    return result
```

Generates:



```
def divide(numerator: float|int, denominator: float|int) -> float:
    """
    divide _summary_

    _extended_summary_

    :param numerator: _description_
    :type numerator: float | int
    :param denominator: _description_
    :type denominator: float | int
    :return: _description_
    :rtype: float
    """
    result = float(numerator / denominator)

    return result
```

Then fill out the underscored fields. Writing formatted docstrings makes complex, interconnected projects much easier when you can mouse over a function call and see exactly what you had intended when you wrote the function:



```
# set up df_segments to pull other report data from
df_segments = edi.build_df_segment(df=df_source)

# df_n1_wh is buil
df_n1_wh = edi.bui
    df_segments=df
    seg_name="N1",
    filter_criteri
    name_map={4: "
    drop_columns=n
)

# merge df_n1_wh b
df_main = df_main.

# map the DUNS Num
df_main["3PL NUM"]
    lambda x: duns
)

# Create SES clien
ses = boto3.client
    "ses",
```

```
(function) def build_df_segment(
    df: DataFrame,
    edi_data_column_name: str = 'EDI_Data',
    seg_term: str = '~'
) -> DataFrame
```

build_df_segment takes the main dataframe and the EDI data column name, and builds a new DataFrame containing the segments exploded into rows, with the segment name extracted from the data elements.

Parameters

df : *pd.DataFrame*
main dataframe containing the EDI data

edi_data_column_name : *str, optional*
column containing the EDI segment data, defaults to 'EDI_Data'

seg_term : *str, optional*
segment terminator, defaults to '~'

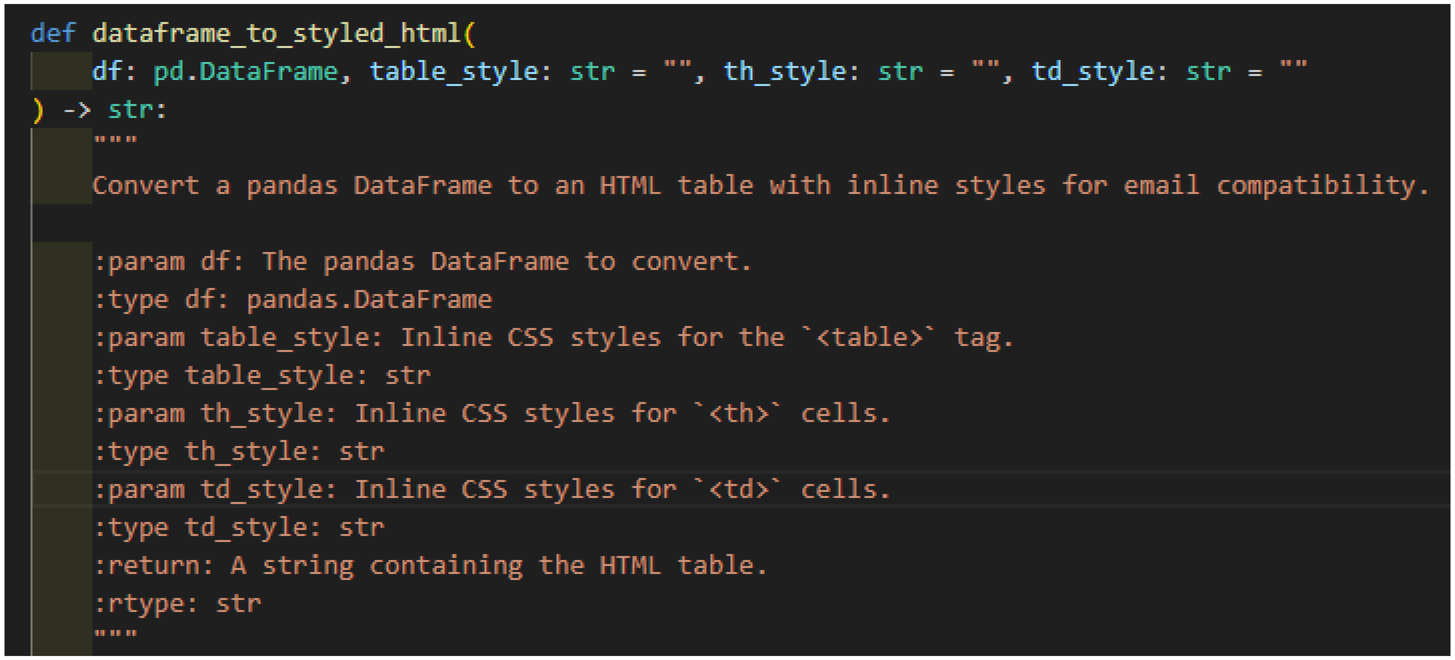
Returns

pd.DataFrame
df_segments: DataFrame containing the segments exploded into rows, with the segment name extracted from the data elements

AI Generated DocStrings

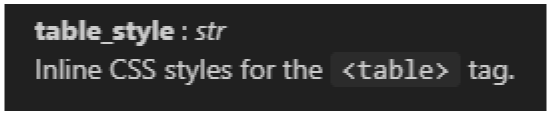
Copilot also does a good job of writing docstrings and can be prompted to write them in your preferred format. As with all things AI-generated, *check the generated docstring for accuracy.*

Here's one I asked Copilot to write (for a function that it also wrote (mostly) on its own):



```python
def dataframe_to_styled_html(
    df: pd.DataFrame, table_style: str = "", th_style: str = "", td_style: str = ""
) -> str:
    """
    Convert a pandas DataFrame to an HTML table with inline styles for email compatibility.

    :param df: The pandas DataFrame to convert.
    :type df: pandas.DataFrame
    :param table_style: Inline CSS styles for the `<table>` tag.
    :type table_style: str
    :param th_style: Inline CSS styles for `<th>` cells.
    :type th_style: str
    :param td_style: Inline CSS styles for `<td>` cells.
    :type td_style: str
    :return: A string containing the HTML table.
    :rtype: str
    """
```

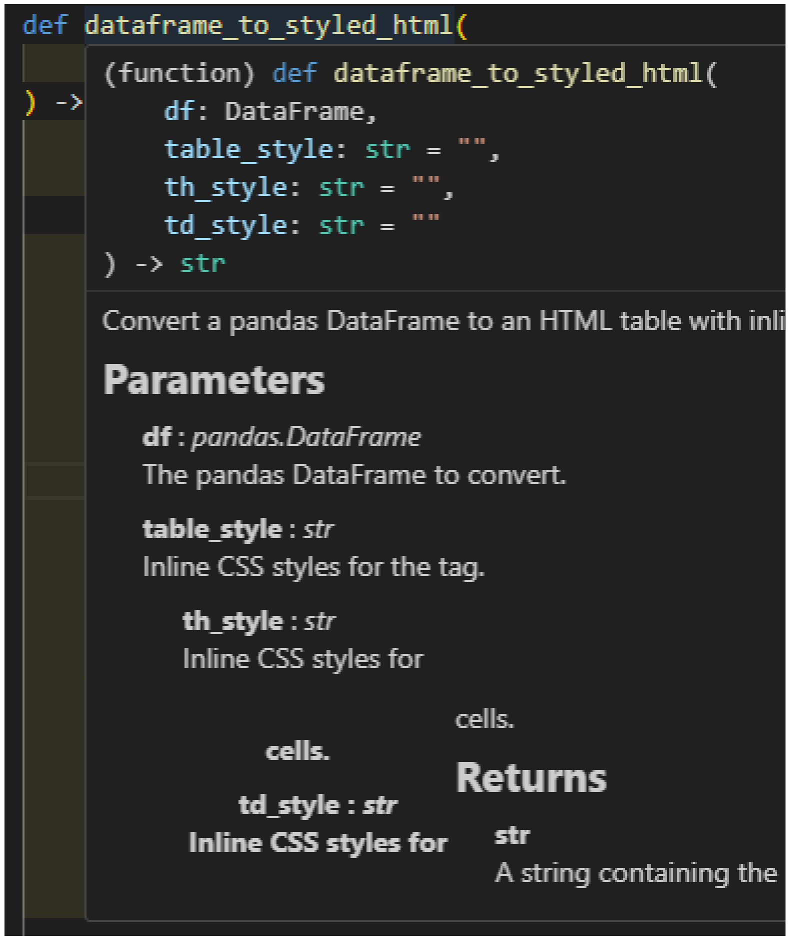
It took some corrections, however. For example: the html tags in the parameters have angle brackets around them, which indicate internal hyperlinks in the ReST format. I had to add backticks ("` ") around the tags to indicate they should be treated as inline code:



table_style : *str*
Inline CSS styles for the `<table>` tag.

Otherwise, it wreaks havoc on the docstring tooltip:

BAD (without backticks): GOOD (with backticks):



```
def dataframe_to_styled_html(
) ->
(function) def dataframe_to_styled_html(
    df: DataFrame,
    table_style: str = "",
    th_style: str = "",
    td_style: str = ""
) -> str
```

Convert a pandas DataFrame to an HTML table with inli

Parameters

df : *pandas.DataFrame*
The pandas DataFrame to convert.

table_style : *str*
Inline CSS styles for the tag.

th_style : *str*
Inline CSS styles for

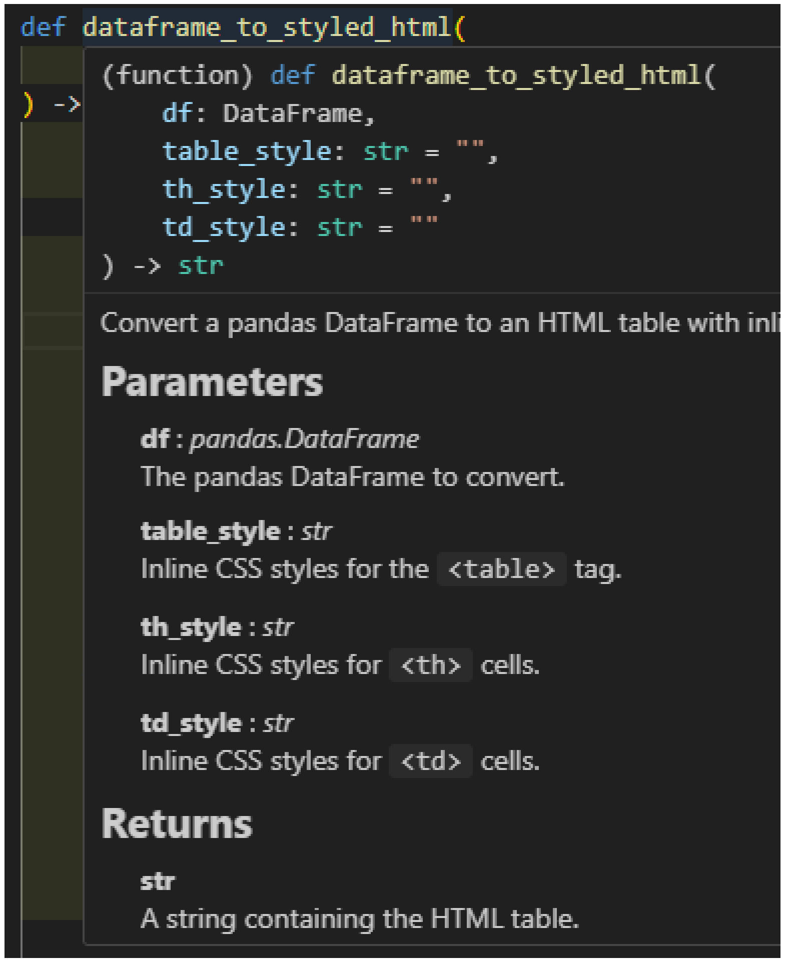
cells.

cells.

td_style : *str*
Inline CSS styles for

Returns

str
A string containing the



```
def dataframe_to_styled_html(
) ->
(function) def dataframe_to_styled_html(
    df: DataFrame,
    table_style: str = "",
    th_style: str = "",
    td_style: str = ""
) -> str
```

Convert a pandas DataFrame to an HTML table with inli

Parameters

df : *pandas.DataFrame*
The pandas DataFrame to convert.

table_style : *str*
Inline CSS styles for the `<table>` tag.

th_style : *str*
Inline CSS styles for `<th>` cells.

td_style : *str*
Inline CSS styles for `<td>` cells.

Returns

str
A string containing the HTML table.

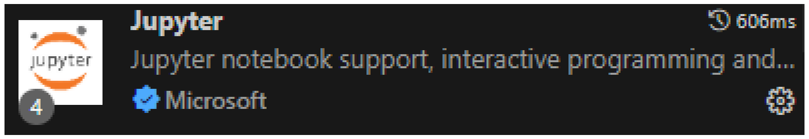
It’s important to proofread any AI-generated docstring, but also to understand how your preferred flavor of markup functions so you can diagnose when something goes wrong.

Other Extensions

While not specifically used with commenting or documentation in mind, there are several other useful extensions that I would classify as either "Quality of Life" or "Workflow Optimization" type extensions. Anything that will help make myself more efficient or make coding less tedious/frustrating.

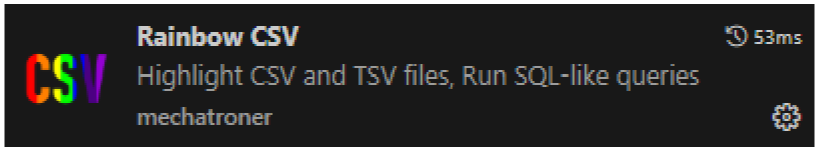
Sorted by number of installs according to the VS Code Extension Marketplace at time of writing.

- Jupyter

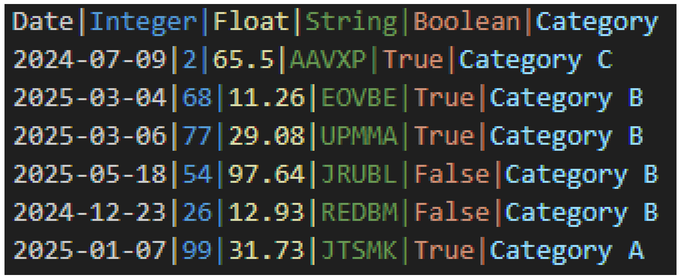

Jupyter 606ms
Jupyter notebook support, interactive programming and...
Microsoft

  - Adds notebook-style coding functionality for interactive programming.
  - Breaks code into digestible code "cells" that can be run individually or sequentially.
  - Helps with debugging, or when writing "train of thought" style code. In other words, when you know where you want to get but you're not sure how you're going to get there.

- Rainbow CSV

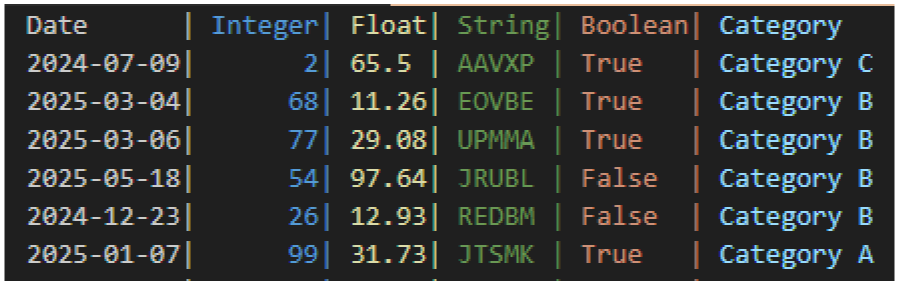

Rainbow CSV 53ms
Highlight CSV and TSV files, Run SQL-like queries
mechatroner

- Color-codes columns in a CSV file



```
Date|Integer|Float|String|Boolean|Category
2024-07-09|2|65.5|AAVXP|True|Category C
2025-03-04|68|11.26|EOVBE|True|Category B
2025-03-06|77|29.08|UPMMA|True|Category B
2025-05-18|54|97.64|JRUBL|False|Category B
2024-12-23|26|12.93|REDBM|False|Category B
2025-01-07|99|31.73|JTSMK|True|Category A
```

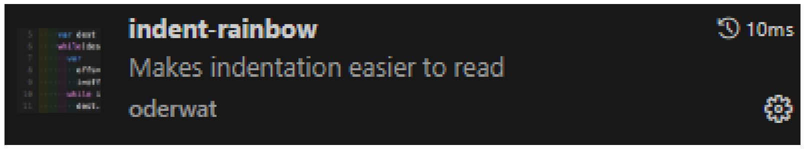
- Can also align columns with whitespace for easier visualization:



```
Date      | Integer| Float| String| Boolean| Category
2024-07-09|       2| 65.5 | AAVXP | True   | Category C
2025-03-04|      68| 11.26| EOVBE | True   | Category B
2025-03-06|      77| 29.08| UPMMA | True   | Category B
2025-05-18|      54| 97.64| JRUBL | False  | Category B
2024-12-23|      26| 12.93| REDBM | False  | Category B
2025-01-07|      99| 31.73| JTSMK | True   | Category A
```

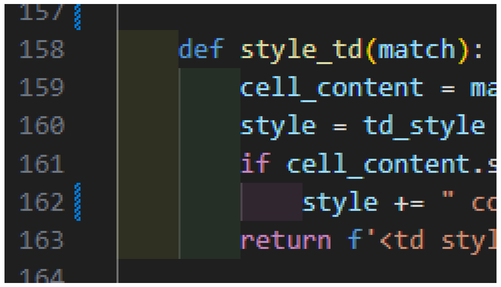
- Works with just about any separator (tab, pipe, semicolon, etc.)
- Has lots of other CSV manipulation tools/techniques built-in (too many to list here)

- indent-rainbow



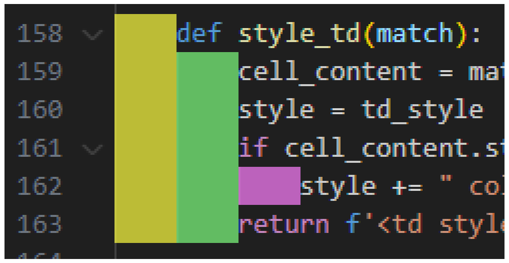
indent-rainbow
Makes indentation easier to read
oderwat

- Another color coding tool, makes indentation levels visually distinct:



```
def style_td(match):
    cell_content = ma
    style = td_style
    if cell_content.s
        style += " co
    return f'<td styl
```

- Colors are configurable in settings in RGBA format
  - Change color scheme entirely as an accessibility option for color blind folks
  - Change alpha channel to make color “pop” more against different backgrounds depending on theme
    - Same code block as above, but with 70% alpha (instead of the 7% shown before)



```
def style_td(match):
    cell_content = ma
    style = td_style
    if cell_content.s
        style += " co
    return f'<td styl
```

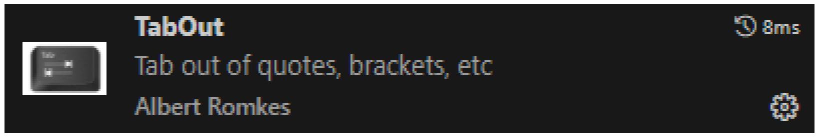
- Black Formatter / autopep8

Black Formatter 63ms
Formatting support for Python files using the Black form...
Microsoft

autopep8 50ms
Formatting support for Python files using the autopep8 f...
Microsoft

    - I’m including both of these here as a reminder to frequently auto-format your code. I prefer Black Formatter for its configurability, and it’s the more popular of the two in the marketplace (by about a 25% margin). The ethos I’ve read around this practice is to make your code more readable to yourself by having a consistent format. Over time you will become accustomed to the format you’ve chosen, allowing you to parse your code easier. It’s easiest if you just pick a formatter and stick with the style it generates.

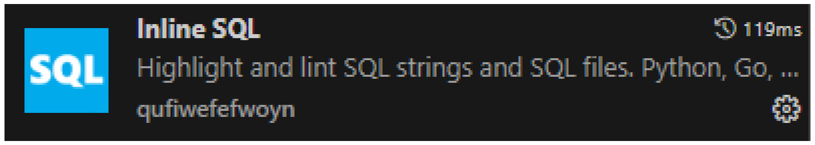
- TabOut



TabOut
8ms
Tab out of quotes, brackets, etc
Albert Romkes

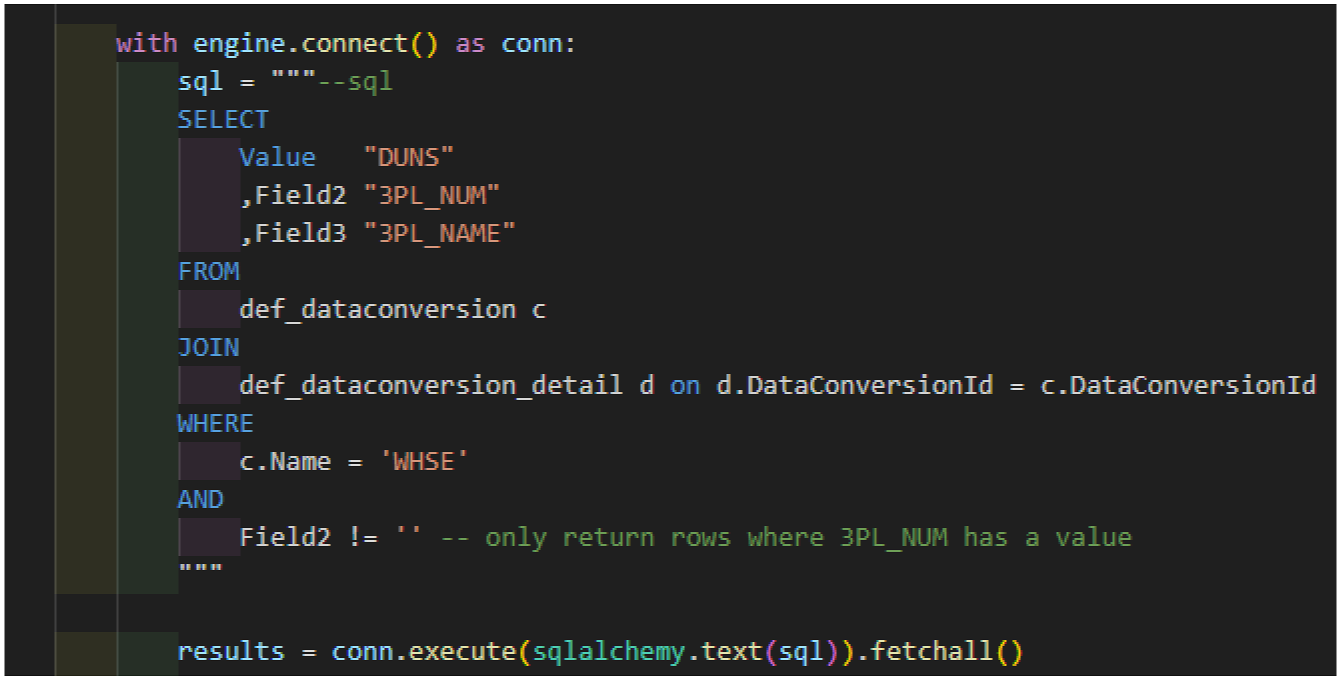
    - A straightforward extension: you can easily jump out of quotes, parentheses, brackets, etc. by pressing tab. Can cause some conflicts when also using tab to auto-complete lines of code, but I find that trade-off worthwhile. YMMV.

- Inline SQL



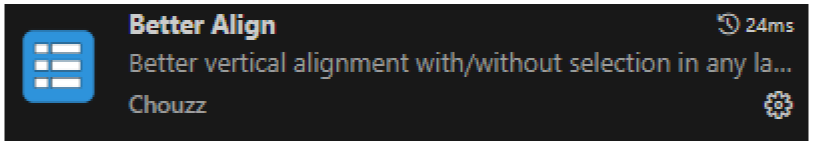
Inline SQL
119ms
Highlight and lint SQL strings and SQL files. Python, Go, ...
qufiwefefwoyn

    - Enables lint highlighting of SQL strings inside of other languages. Supports Python, Go, Rust, JS, TS, Ruby, Java, C#, PHP, and Lua.
    - Example: SQL code seen within python block string. Inserting --sql after the triple quotes will signal Inline SQL to force syntax highlighting for this block:



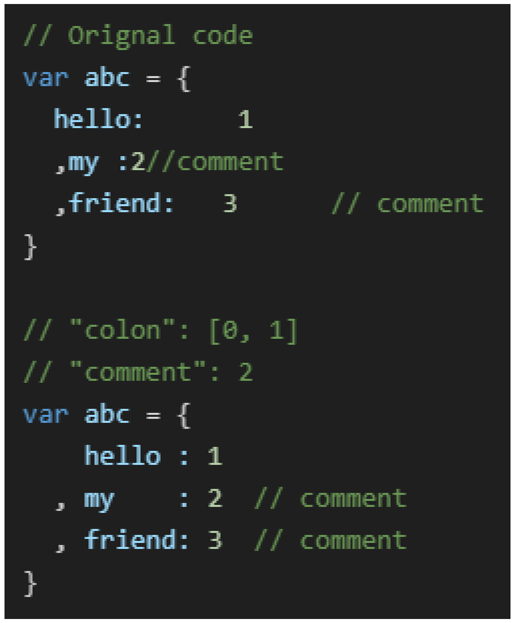
```
with engine.connect() as conn:
    sql = """--sql
    SELECT
        Value   "DUNS"
        ,Field2 "3PL_NUM"
        ,Field3 "3PL_NAME"
    FROM
        def_dataconversion c
    JOIN
        def_dataconversion_detail d on d.DataConversionId = c.DataConversionId
    WHERE
        c.Name = 'WHSE'
    AND
        Field2 != '' -- only return rows where 3PL_NUM has a value
    """

    results = conn.execute(sqlalchemy.text(sql)).fetchall()
```

- Better Align



Better Align 24ms
Better vertical alignment with/without selection in any la...
Chouzz

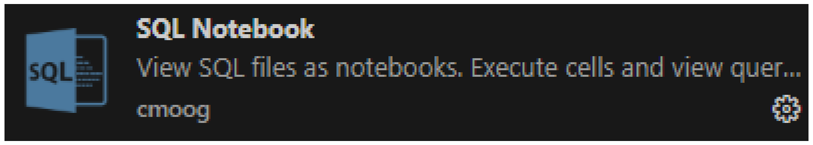
- Aligns blocks of args or collections, can be helpful for visualizing/debugging:



```
// Orignal code
var abc = {
  hello:      1
  ,my :2//comment
  ,friend:   3      // comment
}

// "colon": [0, 1]
// "comment": 2
var abc = {
    hello : 1
  , my    : 2  // comment
  , friend: 3  // comment
}
```

- Spacing is fully configurable

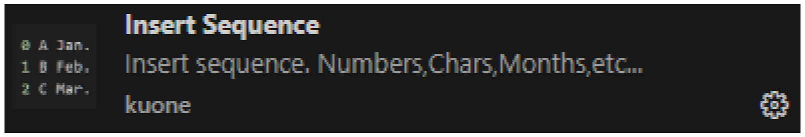
- SQL Notebook



SQL Notebook
View SQL files as notebooks. Execute cells and view quer...
cmoog

  - Same concept as Jupyter notebooks, but applied to SQL. Allows for visually breaking down your SQL code into sections for rapid development and debugging.

- Insert Sequence

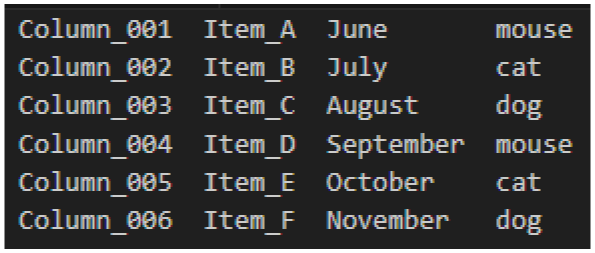


Insert Sequence

Insert sequence. Numbers,Chars,Months,etc...

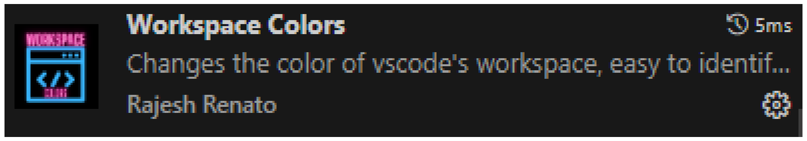
kuone

    - Does what it says on the tin. Sometimes you need a sequence of numbers, letters, dates, or just some repeated strings. Works with column selection for quickly adding prefixes/suffixes to existing text.



```
Column_001  Item_A  June       mouse
Column_002  Item_B  July       cat
Column_003  Item_C  August     dog
Column_004  Item_D  September  mouse
Column_005  Item_E  October    cat
Column_006  Item_F  November   dog
```

- Workspace Colors



Workspace Colors

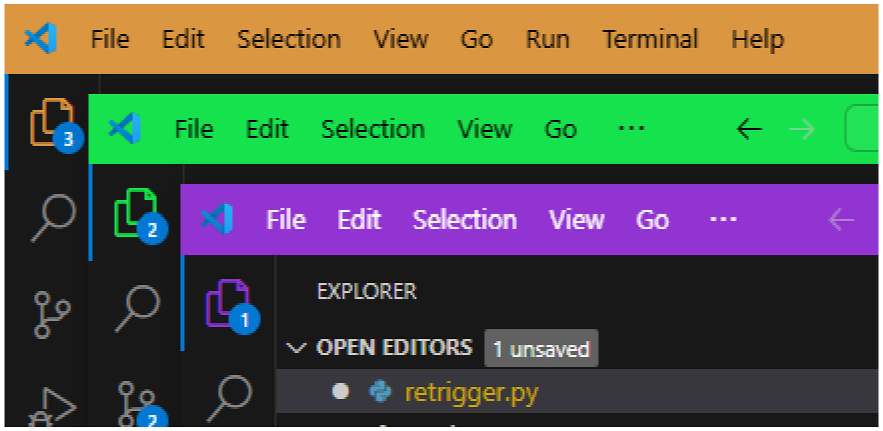
5ms

Changes the color of vscode's workspace, easy to identif...

Rajesh Renato

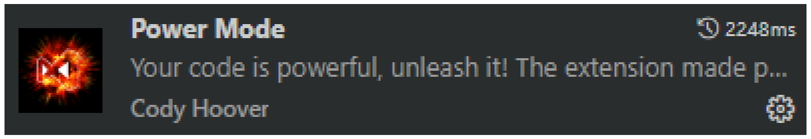
    - Assigns a random color (configurable) to your workspace, provides a quick visual indication of which codebase you're working in if you are working on multiple projects simultaneously.

- Color setting is saved to your workspace settings, so can be backed up to repo or shared across multiple machines to provide consistency.



- Can you tell I really like color-coding?

- *Honorable Mention:* Power Mode



Power Mode 2248ms
Your code is powerful, unleash it! The extension made p...
Cody Hoover

  - Neither a QoL nor a Workflow Efficiency extension, this is just a silly extension that I turn on sometimes to give me a laugh. It adds particle effects, explosions, screen shaking, c-c-c-combo multipliers, or any of several other silly little effects to your UI while you type. Lots of options to pick from, and many community-added effects on their github repo.