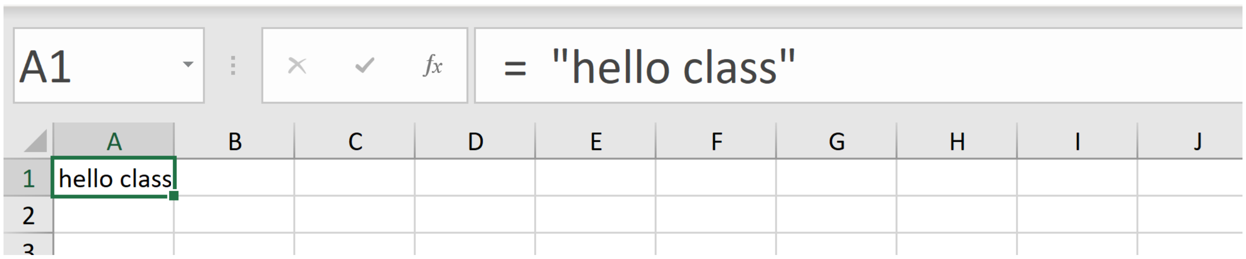
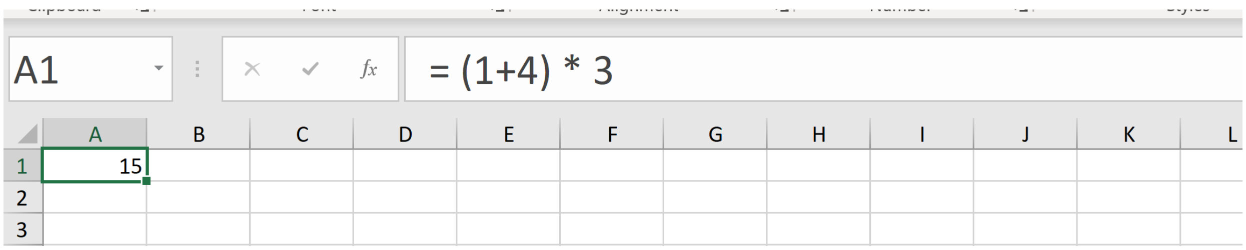
Welcome to Python (in Jupyter Notebook)!

Entering Code

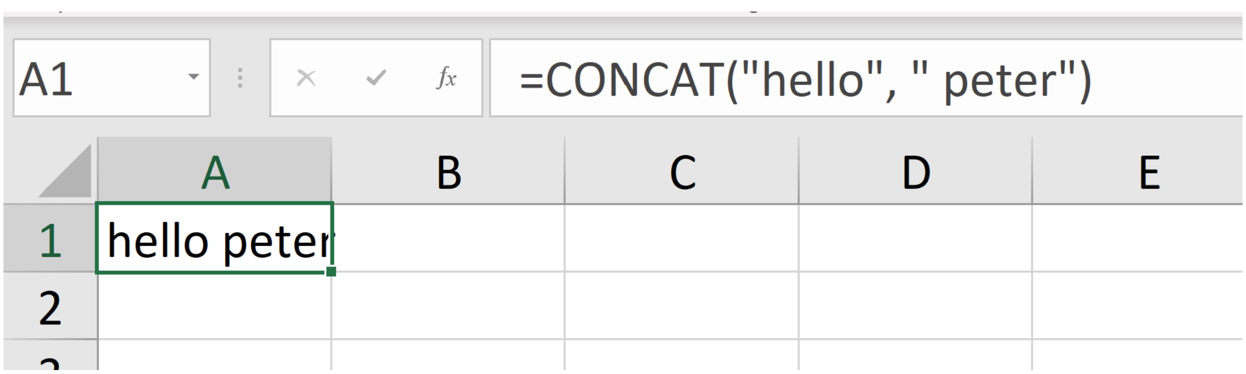
Before we load the data, let's talk a little more about how Python is and isn't like the Excel formula editor (the place you've been doing your coding in excel).
Just like in Excel, you can enter code or formula into each cell of this jupyter notebook. **press "Run" or shift+enter to see the result**



In [1]:
```
"hello class"
```
Out[1]:
```
'hello class'
```



In [2]:
```
(1+4)*3
```
Out[2]:
```
15
```

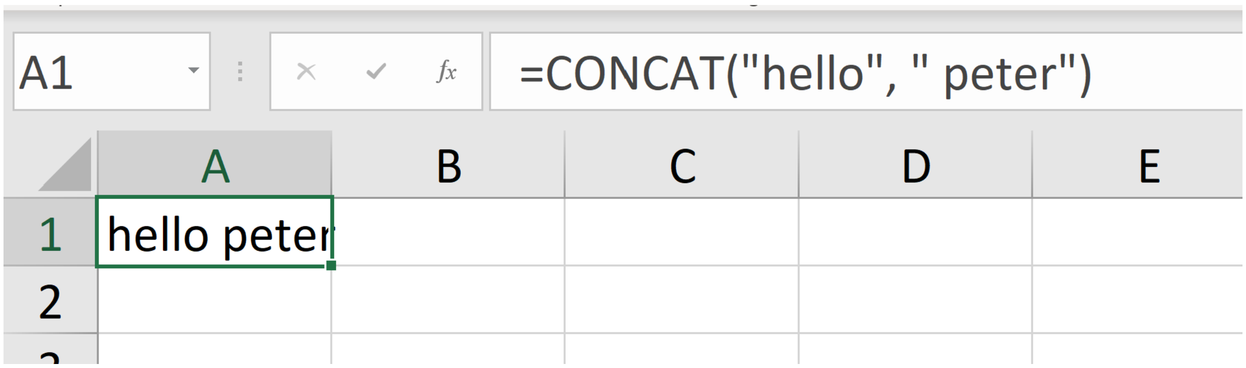


In [3]:
```
"hello" + " peter"
```
Out[3]:
```
'hello peter'
```

Assigning Variables

In Excel, when you type a formula, its result is being assigned to a cell.
For example, here I am putting the formula **CONCAT("hello", " peter")** into cell A1.
You can think of this as a line of code:

```
A1 = CONCAT("hello", " peter")
```



In Python, the same code would look like this (in Python you can use + instead of CONCAT):

In [4]:
```python
A1 = "hello" + " peter"
```

You can view the value assigned to a variable by typing the variable into a cell then running it (shift+enter)

In [5]:
```python
A1
```

Out[5]:
```
'hello peter'
```

In Python you can use any word to name a variable (as long as it doesn't have spaces or any special character besides the underscore _)

In [6]:
```python
result = (45*710)**.5
```

In [7]:
```python
result
```

Out[7]:
```
178.74562931719478
```

Try to make your variable names as intuitive and clear as possible. I constantly forget what the variable names I come up with mean

In [8]:
```python
one_hundred_dogs = "dog " * 100
```

In [9]:
```python
one_hundred_dogs
```

Out[9]:
```
'dog dog dog dog dog dog dog dog dog dog dog dog dog dog dog dog dog dog dog dog dog dog
dog dog dog dog dog dog dog dog dog dog dog dog dog dog dog dog dog dog dog dog dog dog
dog dog dog dog dog dog dog dog dog dog dog dog dog dog dog dog dog dog dog dog dog dog
```

```
dog dog dog dog dog dog dog dog dog dog dog dog dog dog dog dog dog dog dog dog dog dog
dog dog dog dog dog dog dog dog dog dog dog dog '
```

The biggest difference here is that we can't see what all of our assigned variables are in a clean spreadsheet layout like in Excel. We assign values to variables rather than positioning them in a spreadsheet, and we have to consult our earlier code to remember that we've already assigned the following variables:

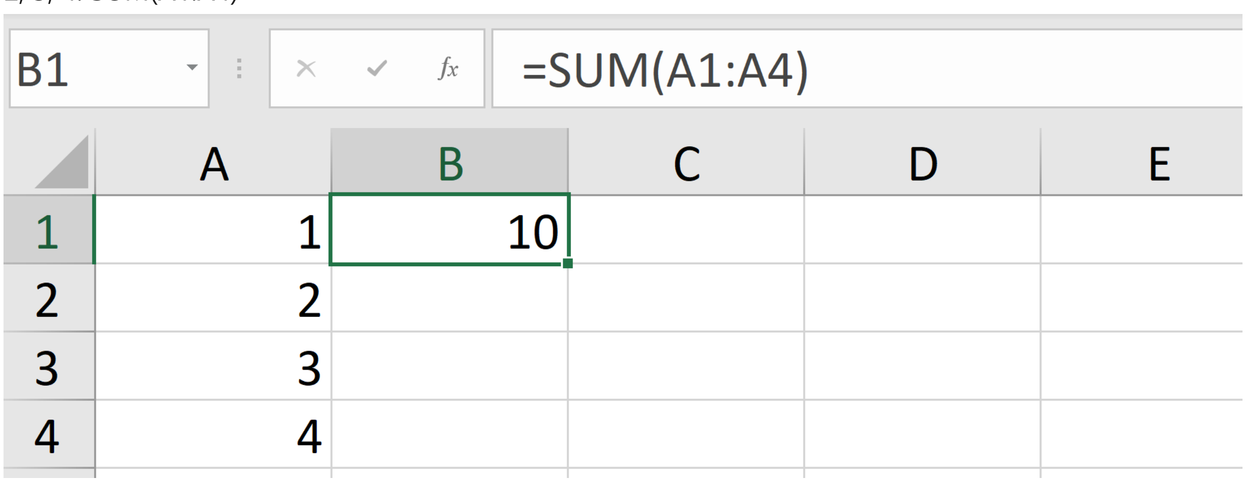
```
A1
result
one_hundred_dogs
```

Functions

Functions actually work pretty similarly in Excel and in Python. You type the name of the function then parenthesis, then whatever inputs the function requires:

```
Excel -> SUM()
Python -> sum()
```

One of the biggest differences is that it's a lot easier to write custom functions in Python, but we won't be getting into that. Consider the code in Excel for adding up a list of numbers, in this case 1, 2, 3, 4. SUM(A1:A4)



The process in Python is very similar

In [10]:
```python
A = [1, 2, 3, 4]
```

In [11]:
```python
sum(A)
```

Out[11]: 10

Let's try combining functions

In [12]:
```python
len(A)
```

Out[12]: 4

In [13]:
```
average = sum(A) / len(A)
```

In [14]:
```
average
```

Out[14]: 2.5

Data Analysis

OK, congrats you now get the basics of what Python is. There are a lot of important topics we are going to not cover today -

- iteration (for, while)
- logical operators (and, or, if)
- defining functions
- other data types (dictionaries, tuples)
- etc.

I want us to get into using Python for actual tasks as quickly as possible, so we are going to leverage what's built into the Pandas library now, because that's what you would actually be doing most of the time if you were using Python for work.

The following code will load the Pandas library and make it so that we can reference Pandas by just typing in pd (it's like a nickname)
I also include a tiny bit of code that makes it so that Pandas won't use scientific notation and will instead round numbers to the sixth decimal place. This is personal preference (I never know what scientific notation means)

In [15]:
```
import pandas as pd
pd.options.display.float_format = '{:.6f}'.format
```

In [16]:
```
pd
```

Out[16]:
```
<module 'pandas' from 'C:\\Users\\peteramerkhanian\\Anaconda3\\lib\\site-packages\\panda
s\\__init__.py'>
```

Loading Data

In Excel, loading data is as simple as double clicking a .xlsx file or a .csv file. In Python and Pandas, opening a file is a little different. It needs to be in the same folder as your Jupyter notebook file, and you open it with a Pandas function, like one of the following (depending on whether your file is .csv or .xlsx):

```
pd.read_csv()
pd.read_excel()
```

by typing pd. you get access to all of the functions that Pandas comes with

In [17]:

```
pd.read_csv("Week_10_Workbook_FINAL_2021.csv")
```

Out[17]:

	Country Code	Country Name	Region ID	Region Name	Income Group Acronym	Income Group Name	Transaction Type ID	Transaction Type Name	Fiscal Year
0	ABW	Aruba	6	Western Hemisphere	HIC	High Income Country	2	Obligations	1999
1	ABW	Aruba	6	Western Hemisphere	HIC	High Income Country	2	Obligations	2000
2	ABW	Aruba	6	Western Hemisphere	HIC	High Income Country	2	Obligations	2004
3	ABW	Aruba	6	Western Hemisphere	HIC	High Income Country	2	Obligations	2005
4	ABW	Aruba	6	Western Hemisphere	HIC	High Income Country	2	Obligations	2006
...	...	...	...	...	...	...	...	...	...
21591	ZWE	Zimbabwe	5	Sub-Saharan Africa	LMIC	Lower Middle Income Country	3	Disbursements	2017
21592	ZWE	Zimbabwe	5	Sub-Saharan Africa	LMIC	Lower Middle Income Country	3	Disbursements	2018
21593	ZWE	Zimbabwe	5	Sub-Saharan Africa	LMIC	Lower Middle Income Country	3	Disbursements	2019
21594	ZWE	Zimbabwe	5	Sub-Saharan Africa	LMIC	Lower Middle Income Country	3	Disbursements	2020
21595	ZWE	Zimbabwe	5	Sub-Saharan Africa	LMIC	Lower Middle Income Country	3	Disbursements	2021

21596 rows × 11 columns

That code read the data in, but we need to save into our python environment if we want to keep

working with it.
Let's assign the result of pd.read_csv() to a variable called df (short for data frame), which we'll use to refer to our data in the future

In [18]:
```
df = pd.read_csv("Week_10_Workbook_FINAL_2021.csv")
```

When we type in df, we can see our table

In [19]:
```
df
```

Out[19]:

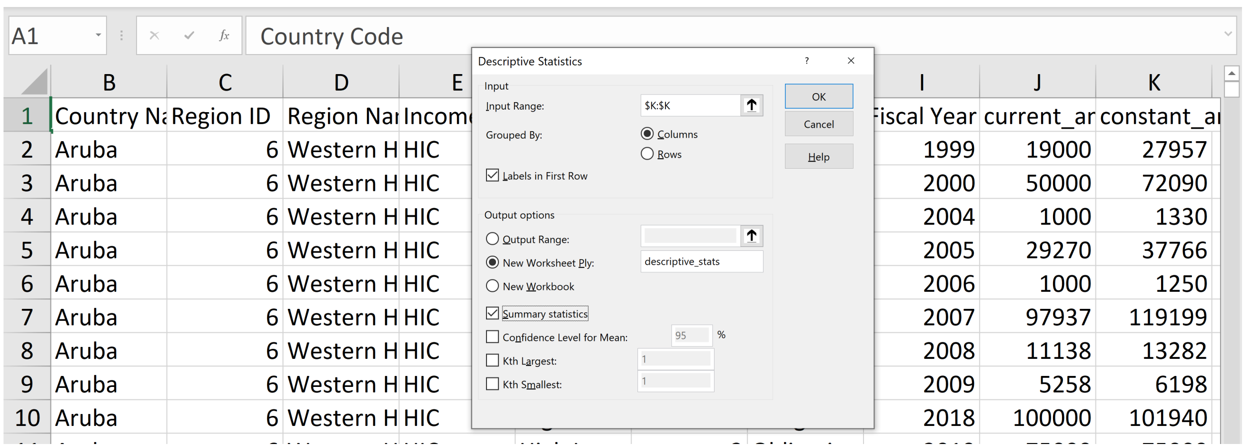
	Country Code	Country Name	Region ID	Region Name	Income Group Acronym	Income Group Name	Transaction Type ID	Transaction Type Name	Fiscal Year
0	ABW	Aruba	6	Western Hemisphere	HIC	High Income Country	2	Obligations	1999
1	ABW	Aruba	6	Western Hemisphere	HIC	High Income Country	2	Obligations	2000
2	ABW	Aruba	6	Western Hemisphere	HIC	High Income Country	2	Obligations	2004
3	ABW	Aruba	6	Western Hemisphere	HIC	High Income Country	2	Obligations	2005
4	ABW	Aruba	6	Western Hemisphere	HIC	High Income Country	2	Obligations	2006
...	...	...	...	...	...	...	...	...	...
21591	ZWE	Zimbabwe	5	Sub-Saharan Africa	LMIC	Lower Middle Income Country	3	Disbursements	2017
21592	ZWE	Zimbabwe	5	Sub-Saharan Africa	LMIC	Lower Middle Income Country	3	Disbursements	2018
21593	ZWE	Zimbabwe	5	Sub-Saharan Africa	LMIC	Lower Middle Income Country	3	Disbursements	2019
21594	ZWE	Zimbabwe	5	Sub-Saharan Africa	LMIC	Lower Middle Income Country	3	Disbursements	2020

	Country Code	Country Name	Region ID	Region Name	Income Group Acronym	Income Group Name	Transaction Type ID	Transaction Type Name	Fiscal Year
21595	ZWE	Zimbabwe	5	Sub-Saharan Africa	LMIC	Lower Middle Income Country	3	Disbursements	2021

21596 rows × 11 columns

Getting Descriptive Statistics

In Excel, we could use analysis toolpak and be able to get descriptive statistics for a given column like this (here I'm getting the descriptive statistics for the constant aid amounts):



In Pandas, we reference specific columns like this:

In [20]:

```python
df["current_amount"]
```

Out[20]:

```
0            19000
1            50000
2             1000
3            29270
4             1000
           ...
21591    226474886
21592    227007896
21593    259153975
21594    235788780
21595    186140320
Name: current_amount, Length: 21596, dtype: int64
```

While using Pandas, we can access general Pandas functions by typing pd then any given function

```
pd.function()
```

But we can also access functions in other more specialized ways. Pandas has a number of specific functions that are for anlyzing and processing columns of data. to access those, you select a column

```
df["current_amount"]
```

then you type . and can access a number of functions for columns (the technical term is method instead of function, but the distinction isn't important right now)

```
df["current_amount"].describe()
```

In [21]:
```
df["current_amount"].describe()
```

Out[21]:
```
count          21596.000000
mean       172682905.133497
std        971650169.397417
min        -67313490.000000
25%          1744487.750000
50%         16981664.500000
75%         76056486.500000
max      28851188510.000000
Name: current_amount, dtype: float64
```

Note that describe() is the Pandas function for getting descriptive statistics from a column. Here are a few other column functions

In [22]:
```
df["constant_amount"].sum()
```

Out[22]:
```
5467954240727
```

In [23]:
```
df["constant_amount"].median()
```

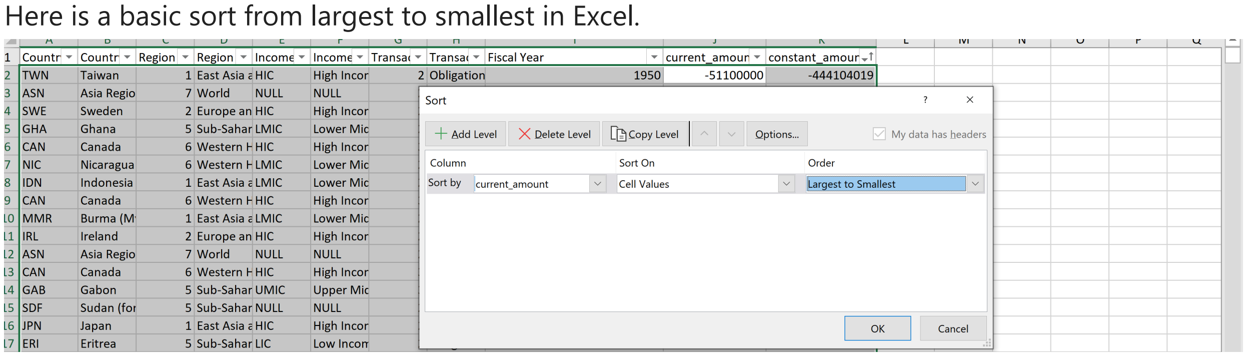
Out[23]:
```
26205987.5
```

In [24]:
```
df["constant_amount"].mean()
```

Out[24]:
```
253192917.2405538
```

Sorting

Here is a basic sort from largest to smallest in Excel.



In Pandas, sorting is one of the special functions for use on full datasets (not single columns), so you can get access to the sorting function by typing . after a variable that contains a dataset. We have the variable df containing all our data, so the notation will be

```
df.sort_values()
```

Many functions in Pandas and in Python generally can accept a lot of different arguments. For example, this is the full number of arguments you can add to df.sort_values()

```
df.sort_values(
    by,
    axis=0,
    ascending=True,
    inplace=False,
    kind='quicksort',
    na_position='last',
    ignore_index=False,
    key: 'ValueKeyFunc' = None,
)
```

Many of these are optional or else have default values, so you don't need to explicitly add those arguments in.
In our case, we will just be supplying two arguments, "by" which is the column we want to sort everything on, and "ascending" which we will set to False because we want the data sorted descending (largest to smallest)

In [25]:

```
df.sort_values(by="current_amount", ascending=False)
```

Out[25]:

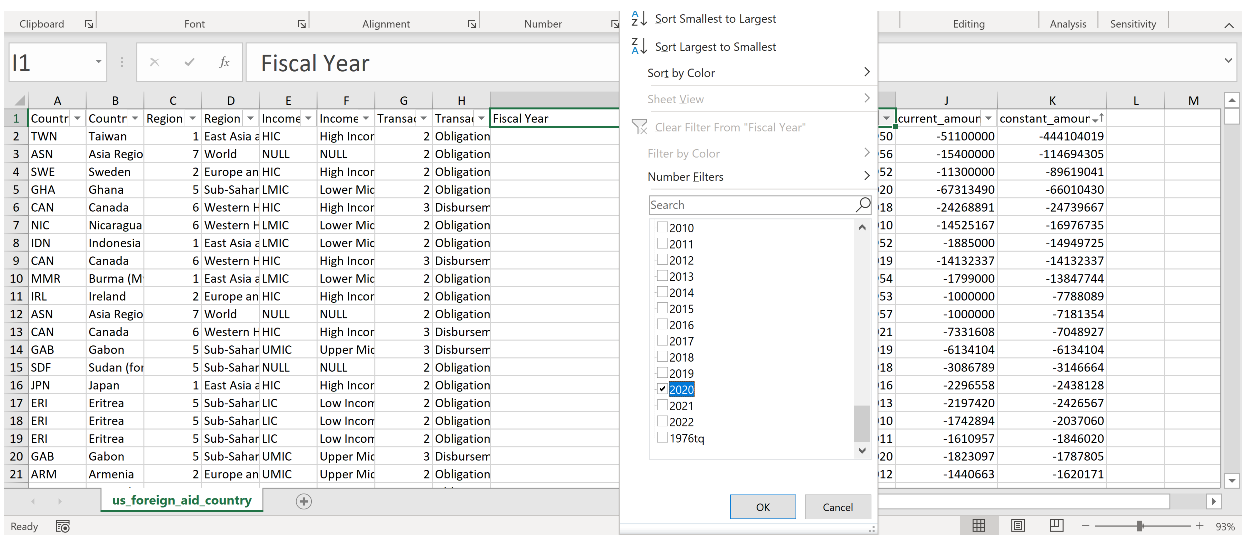
	Country Code	Country Name	Region ID	Region Name	Income Group Acronym	Income Group Name	Transaction Type ID	Transaction Type Name	Fiscal Year
20975	WLD	World	7	World	NaN	NaN	18	President's Budget Requests	2017
20974	WLD	World	7	World	NaN	NaN	18	President's Budget Requests	2016
20973	WLD	World	7	World	NaN	NaN	18	President's Budget Requests	2015
20863	WLD	World	7	World	NaN	NaN	1	Appropriated and Planned	2020
20859	WLD	World	7	World	NaN	NaN	1	Appropriated and Planned	2016
...	...	...	...	...	...	...	...	...	...
14051	NIC	Nicaragua	6	Western Hemisphere	LMIC	Lower Middle Income Country	2	Obligations	2010
691	ASN	Asia Region	7	World	NaN	NaN	2	Obligations	1956

	Country Code	Country Name	Region ID	Region Name	Income Group Acronym	Income Group Name	Transaction Type ID	Transaction Type Name	Fiscal Year
2905	CAN	Canada	6	Western Hemisphere	HIC	High Income Country	3	Disbursements	2018
19722	TWN	Taiwan	1	East Asia and Oceania	HIC	High Income Country	2	Obligations	1950
6998	GHA	Ghana	5	Sub-Saharan Africa	LMIC	Lower Middle Income Country	2	Obligations	2020

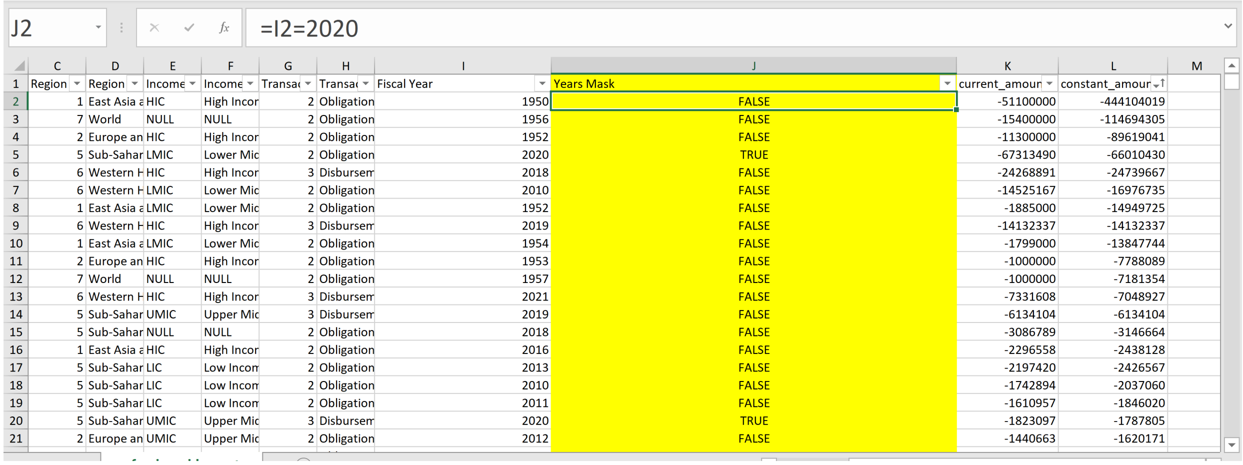
21596 rows × 11 columns

Filtering

In Excel, you filter data by applying a filter to the top row then clicking the drop down for the column you want to filter on, then selecting the values you want to isolate with your filter, in this case 2020:

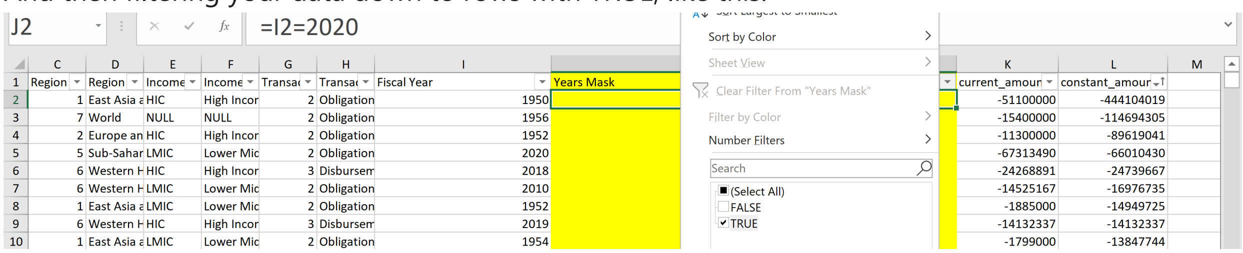


In Python the process is a little different. We first have to create a "mask column" of True and False values - True if a given row is from Fiscal Year 2020, False otherwise. In Excel, that would look more like this:



	C	D	E	F	G	H	I	J	K	L	M
1	Region	Region	Income	Income	Transac	Transac	Fiscal Year	Years Mask	current_amoun	constant_amour	
2	1	East Asia a	HIC	High Incor	2	Obligation	1950	FALSE	-51100000	-444104019	
3	7	World	NULL	NULL	2	Obligation	1956	FALSE	-15400000	-114694305	
4	2	Europe an	HIC	High Incor	2	Obligation	1952	FALSE	-11300000	-89619041	
5	5	Sub-Sahar	LMIC	Lower Mic	2	Obligation	2020	TRUE	-67313490	-66010430	
6	6	Western H	HIC	High Incor	3	Disburser	2018	FALSE	-24268891	-24739667	
7	6	Western H	LMIC	Lower Mic	2	Obligation	2010	FALSE	-14525167	-16976735	
8	1	East Asia a	LMIC	Lower Mic	2	Obligation	1952	FALSE	-1885000	-14949725	
9	6	Western H	HIC	High Incor	3	Disburser	2019	FALSE	-14132337	-14132337	
10	1	East Asia a	LMIC	Lower Mic	2	Obligation	1954	FALSE	-1799000	-13847744	
11	2	Europe an	HIC	High Incor	2	Obligation	1953	FALSE	-1000000	-7788089	
12	7	World	NULL	NULL	2	Obligation	1957	FALSE	-1000000	-7181354	
13	6	Western H	HIC	High Incor	3	Disburser	2021	FALSE	-7331608	-7048927	
14	5	Sub-Sahar	UMIC	Upper Mic	3	Disburser	2019	FALSE	-6134104	-6134104	
15	5	Sub-Sahar	NULL	NULL	2	Obligation	2018	FALSE	-3086789	-3146664	
16	1	East Asia a	HIC	High Incor	2	Obligation	2016	FALSE	-2296558	-2438128	
17	5	Sub-Sahar	LIC	Low Incon	2	Obligation	2013	FALSE	-2197420	-2426567	
18	5	Sub-Sahar	LIC	Low Incon	2	Obligation	2010	FALSE	-1742894	-2037060	
19	5	Sub-Sahar	LIC	Low Incon	2	Obligation	2011	FALSE	-1610957	-1846020	
20	5	Sub-Sahar	UMIC	Upper Mic	3	Disburser	2020	TRUE	-1823097	-1787805	
21	2	Europe an	UMIC	Upper Mic	2	Obligation	2012	FALSE	-1440663	-1620171	

And then filtering your data down to rows with TRUE, like this:



	C	D	E	F	G	H	I	J	K	L	M
1	Region	Region	Income	Income	Transac	Transac	Fiscal Year	Years Mask	current_amoun	constant_amour	
2	1	East Asia a	HIC	High Incor	2	Obligation	1950		-51100000	-444104019	
3	7	World	NULL	NULL	2	Obligation	1956		-15400000	-114694305	
4	2	Europe an	HIC	High Incor	2	Obligation	1952		-11300000	-89619041	
5	5	Sub-Sahar	LMIC	Lower Mic	2	Obligation	2020		-67313490	-66010430	
6	6	Western H	HIC	High Incor	3	Disburser	2018		-24268891	-24739667	
7	6	Western H	LMIC	Lower Mic	2	Obligation	2010		-14525167	-16976735	
8	1	East Asia a	LMIC	Lower Mic	2	Obligation	1952		-1885000	-14949725	
9	6	Western H	HIC	High Incor	3	Disburser	2019		-14132337	-14132337	
10	1	East Asia a	LMIC	Lower Mic	2	Obligation	1954		-1799000	-13847744	

Here is how we make that "mask column" in Pandas, to see if each row has a Fiscal Year of 2020:

In [26]:
```
df["Fiscal Year"] == "2020"
```

Out[26]:
```
0        False
1        False
2        False
3        False
4        False
         ...
21591    False
21592    False
21593    False
21594    False
21595    False
Name: Fiscal Year, Length: 21596, dtype: bool
```

We can then apply our mask to the data using the following notation. This will filter the data down to only the rows where our "mask column" above is True.

In [27]:
```
df[df["Fiscal Year"] == 2020]
```

Out[27]:

	Country Code	Country Name	Region ID	Region Name	Income Group Acronym	Income Group Name	Transaction Type ID	Transaction Type Name	Fiscal Year
10	ABW	Aruba	6	Western Hemisphere	HIC	High Income Country	2	Obligations	2020
139	AFG	Afghanistan	4	South and Central Asia	LIC	Low Income Country	1	Appropriated and Planned	2020

	Country Code	Country Name	Region ID	Region Name	Income Group Acronym	Income Group Name	Transaction Type ID	Transaction Type Name	Fiscal Year
140	AFG	Afghanistan	4	South and Central Asia	LIC	Low Income Country	2	Obligations	2020
141	AFG	Afghanistan	4	South and Central Asia	LIC	Low Income Country	3	Disbursements	2020
142	AFG	Afghanistan	4	South and Central Asia	LIC	Low Income Country	18	President's Budget Requests	2020
...	...	...	...	...	...	...	...	...	...
21465	ZMB	Zambia	5	Sub-Saharan Africa	LMIC	Lower Middle Income Country	18	President's Budget Requests	2020
21494	ZWE	Zimbabwe	5	Sub-Saharan Africa	LIC	Low Income Country	1	Appropriated and Planned	2020
21507	ZWE	Zimbabwe	5	Sub-Saharan Africa	LIC	Low Income Country	18	President's Budget Requests	2020
21573	ZWE	Zimbabwe	5	Sub-Saharan Africa	LMIC	Lower Middle Income Country	2	Obligations	2020
21594	ZWE	Zimbabwe	5	Sub-Saharan Africa	LMIC	Lower Middle Income Country	3	Disbursements	2020

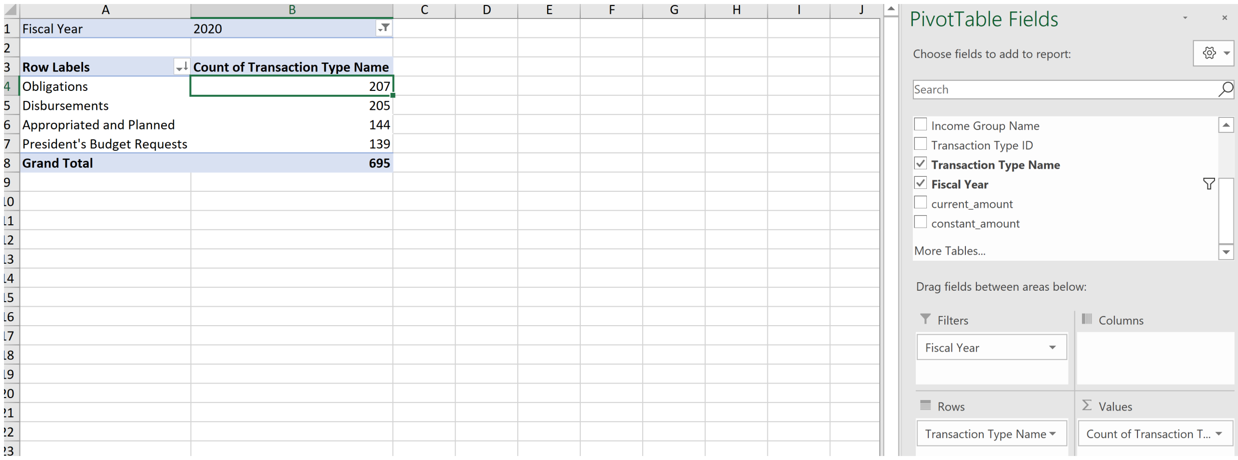
695 rows × 11 columns

Let's save that to a variable, df_2020, so that we can use this filtered data later

In [28]:
```
df_2020 = df[df["Fiscal Year"] == 2020]
```

Pivot Tables

Here's an example of an Excel pivot table that counts up how many aid disbersements there were of each transaction type in FY2020



	A	B
1	Fiscal Year	2020
2		
3	Row Labels	Count of Transaction Type Name
4	Obligations	207
5	Disbursements	205
6	Appropriated and Planned	144
7	President's Budget Requests	139
8	Grand Total	695

In Pandas, creating simple pivot tables that only deal with one column is done using the value_counts() function. This is a function that operates directly from a column, so you access it with the following notation:

```
df["column_name"].value_counts()
```

Let's keep working with the df_2020 variable we made, which is the original data filtered down to just FY2020

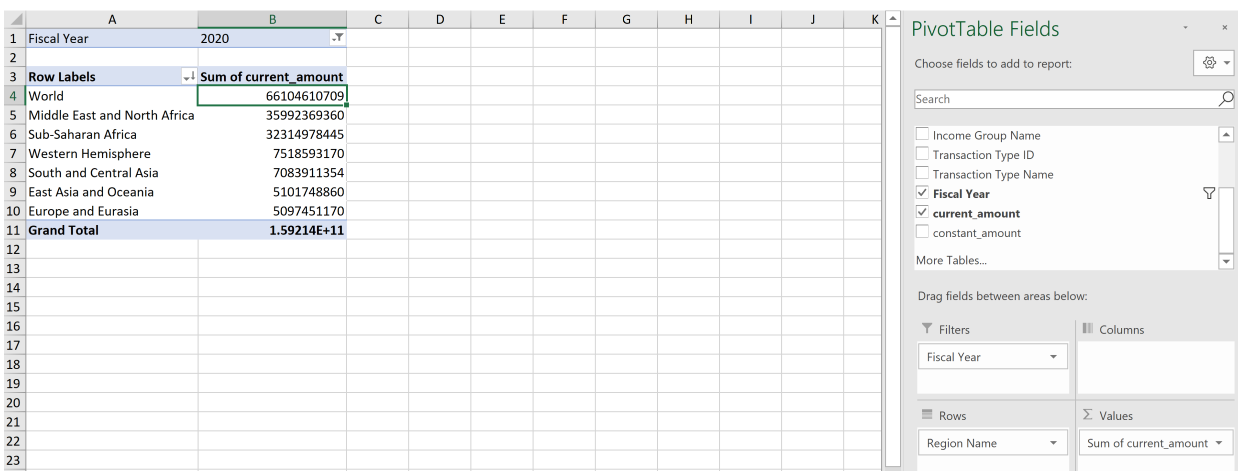
In [29]:

```
df_2020["Transaction Type Name"].value_counts()
```

Out[29]:

```
Obligations                    207
Disbursements                  205
Appropriated and Planned       144
President's Budget Requests    139
Name: Transaction Type Name, dtype: int64
```

Here's a slightly more complex pivot table, and in this case we can't just use value_counts(), we will have to use the more powerful pivot_table() function



	A	B
1	Fiscal Year	2020
2		
3	Row Labels	Sum of current_amount
4	World	66104610709
5	Middle East and North Africa	35992369360
6	Sub-Saharan Africa	32314978445
7	Western Hemisphere	7518593170
8	South and Central Asia	7083911354
9	East Asia and Oceania	5101748860
10	Europe and Eurasia	5097451170
11	Grand Total	1.59214E+11

pivot_table() is a function that you access from a full dataset, so the general syntax is:

```
df.pivot_table()
```

Let's use this pivot_table() function on our filtered down df_2020 dataset

In [30]:

```
df_2020.pivot_table(index='Region Name', values='current_amount', aggfunc='sum')
```

Out[30]:

	current_amount
Region Name	
East Asia and Oceania	5101748860
Europe and Eurasia	5097451170
Middle East and North Africa	35992369360
South and Central Asia	7083911354
Sub-Saharan Africa	32314978445
Western Hemisphere	7518593170
World	66104610709

Let's save that into a variable, pivot_table
At that point, it's a standalone dataset, and we can apply full dataset functions, like sort_values()

In [31]:

```
pivot_table = df_2020.pivot_table(index='Region Name', values='current_amount', aggfunc
```

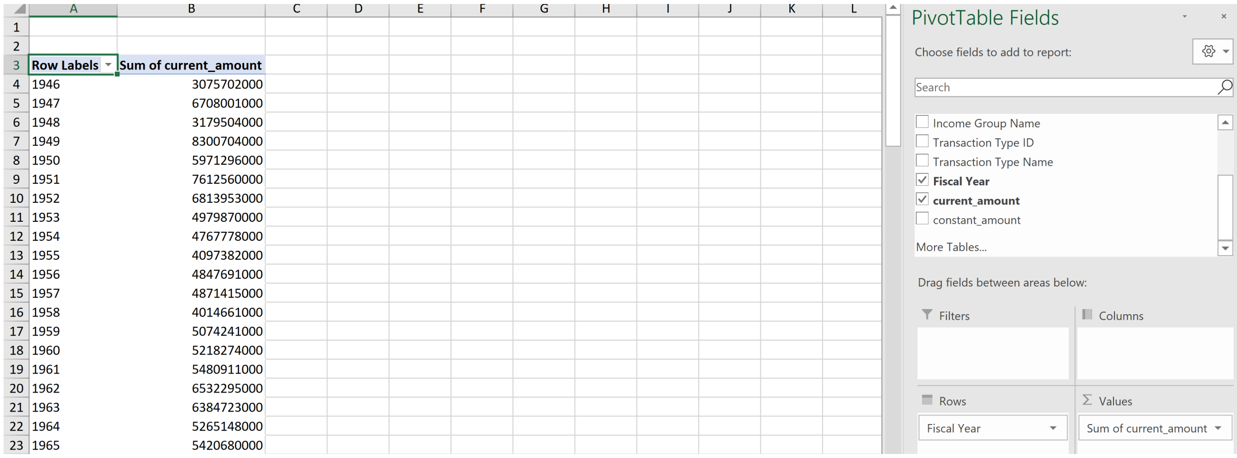
In [32]:

```
pivot_table.sort_values(by="current_amount", ascending=False)
```

Out[32]:

	current_amount
Region Name	
World	66104610709
Middle East and North Africa	35992369360
Sub-Saharan Africa	32314978445
Western Hemisphere	7518593170
South and Central Asia	7083911354
East Asia and Oceania	5101748860
Europe and Eurasia	5097451170

Here's another slightly complex pivot table that sums up aid amounts for each fiscal year



Row Labels	Sum of current_amount
1946	3075702000
1947	6708001000
1948	3179504000
1949	8300704000
1950	5971296000
1951	7612560000
1952	6813953000
1953	4979870000
1954	4767778000
1955	4097382000
1956	4847691000
1957	4871415000
1958	4014661000
1959	5074241000
1960	5218274000
1961	5480911000
1962	6532295000
1963	6384723000
1964	5265148000
1965	5420680000

In [33]:
```
df.pivot_table(index='Fiscal Year', values='current_amount', aggfunc='sum')
```

Out[33]:

Fiscal Year	current_amount
1946	3075702000
1947	6708001000
1948	3179504000
1949	8300704000
1950	5971296000
...	...
2018	208716100386
2019	208473462710
2020	159213663068
2021	73243656789
2022	42131052000

77 rows × 1 columns

Let's save that into a variable, pivot_table_year, so we can keep using this dataset

In [34]:
```
pivot_table_year = df.pivot_table(index='Fiscal Year', values='current_amount', aggfunc
```

VLOOKUP()

in Pandas, instead of VLOOKUP(), we use pd.merge()

I think that as a function, pd.merge() is actually a bit simpler that VLOOKUP().

Before we use the function though, let's load another dataset that we will be VLOOKUP()ing into.

In this case, we have U.S. GDP data for each year since 1947

In [35]:
```
gdp = pd.read_csv("GDP.csv")
```

In [36]:

```
gdp
```

Out[36]:

	Fiscal Year	GDP
0	1947	260000000000.000000
1	1948	280000000000.000000
2	1949	271000000000.000000
3	1950	320000000000.000000
4	1951	356000000000.000000
...	...	...
69	2016	19000000000000.000000
70	2017	19900000000000.000000
71	2018	20800000000000.000000
72	2019	21700000000000.000000
73	2020	21500000000000.000000

74 rows × 2 columns

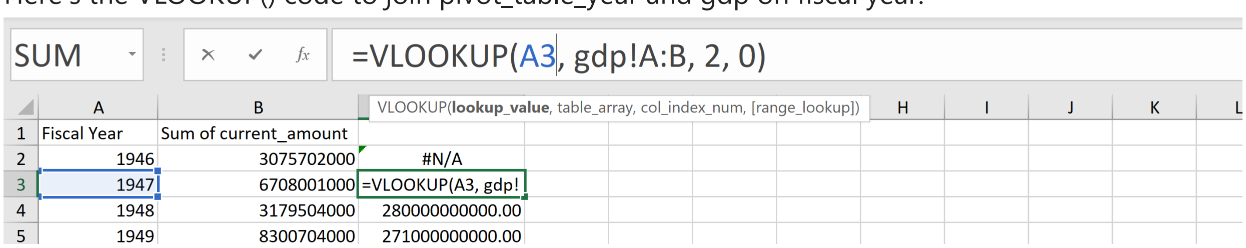
The structure of a VLOOKUP() is:

```
VLOOKUP(value_to_lookup, dataset_2, number_of_column_to_match_on,
exact_or_inexact_match)
```

The structure of pd.merge() will be:

```
pd.merge(dataset_1, dataset_2, column_to_match_on)
```

Note that there are many more optional arguments you can add to pd.merge(), but we are just using those three.
Here's the VLOOKUP() code to join pivot_table_year and gdp on fiscal year:



In [37]:

```
pd.merge(pivot_table_year, gdp, on="Fiscal Year")
```

Out[37]:

	Fiscal Year	current_amount	GDP
0	1947	6708001000	260000000000.000000

	Fiscal Year	current_amount	GDP
1	1948	3179504000	280000000000.000000
2	1949	8300704000	271000000000.000000
3	1950	5971296000	320000000000.000000
4	1951	7612560000	356000000000.000000
...	...	...	...
69	2016	229410557689	19000000000000.000000
70	2017	231653412868	19900000000000.000000
71	2018	208716100386	20800000000000.000000
72	2019	208473462710	21700000000000.000000
73	2020	159213663068	21500000000000.000000

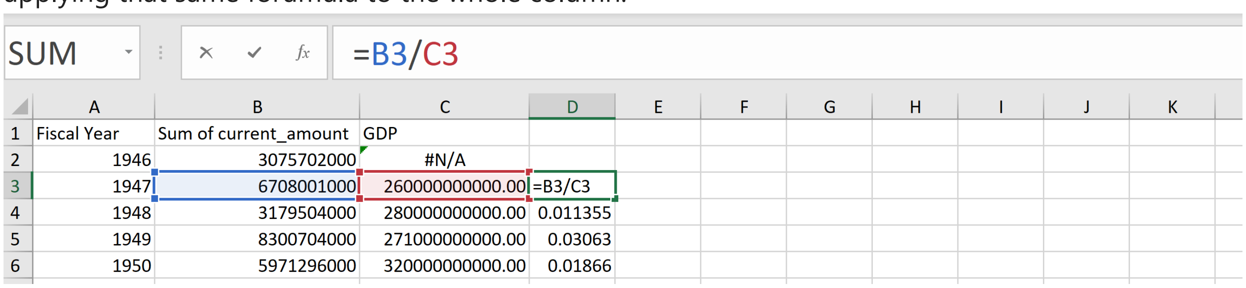
74 rows × 3 columns

Now let's assign that new merged data set to a variable so that we can keep using it

In [38]:
```
df_merged = pd.merge(pivot_table_year, gdp, on="Fiscal Year")
```

Calculating new columns

In Excel, calculating a new column means applying a basic arithmetic formula to one cell, then applying that same forumula to the whole column.


SUM =B3/C3

	A	B	C	D
1	Fiscal Year	Sum of current_amount	GDP	
2	1946	3075702000	#N/A	
3	1947	6708001000	260000000000.00	=B3/C3
4	1948	3179504000	280000000000.00	0.011355
5	1949	8300704000	271000000000.00	0.03063
6	1950	5971296000	320000000000.00	0.01866

In Pandas, we can just apply arithmetic to entire columns, and it will automatically apply it row by row

In [39]:
```
df_merged["current_amount"] / df_merged["GDP"]
```

Out[39]:
```
0     0.025800
1     0.011355
2     0.030630
3     0.018660
4     0.021384
        ...
69    0.012074
70    0.011641
71    0.010034
72    0.009607
```

```
73    0.007405
Length: 74, dtype: float64
```

Let's save that calculated column as a new column in our dataset. We can do that by just referring to the new column name like so:

```
df_merged["amount_as_%_GDP"]
```

Then setting that column equal to the calculated column above

In [40]:
```
df_merged["amount_as_%_GDP"] = df_merged["current_amount"] / df_merged["GDP"]
```

In [41]:
```
df_merged
```

Out[41]:

	Fiscal Year	current_amount	GDP	amount_as_%_GDP
0	1947	6708001000	260000000000.000000	0.025800
1	1948	3179504000	280000000000.000000	0.011355
2	1949	8300704000	271000000000.000000	0.030630
3	1950	5971296000	320000000000.000000	0.018660
4	1951	7612560000	356000000000.000000	0.021384
...	...	...	...	...
69	2016	229410557689	19000000000000.000000	0.012074
70	2017	231653412868	19900000000000.000000	0.011641
71	2018	208716100386	20800000000000.000000	0.010034
72	2019	208473462710	21700000000000.000000	0.009607
73	2020	159213663068	21500000000000.000000	0.007405

74 rows × 4 columns

Plotting

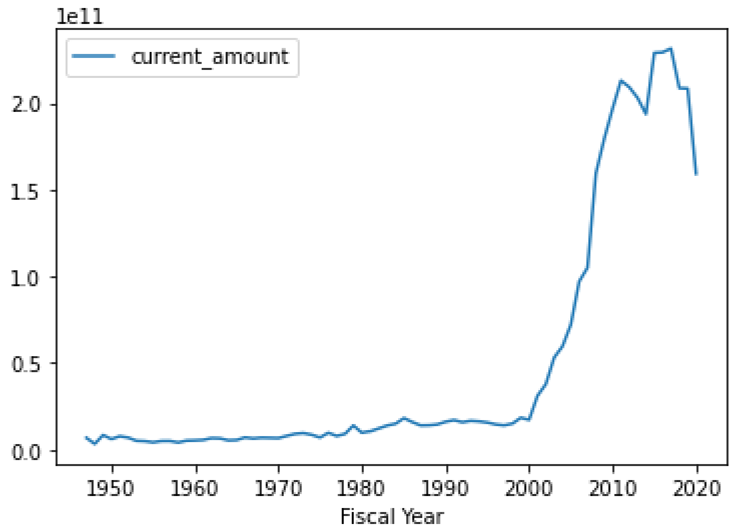
Plotting in Python is generally not ideal. The best data visualization libraries (D3, Leaflet) are generally written in JavaScript rather than Python. Pandas uses another Python library called matplotlib to make plots, and while it is extremely customizable, the visual style is fairly bare bones. The nice thing is that the code is fairly straightfoward.
plot() is a function that can apply to entire datasets, so we access it from a dataset variable

```
df.plot(x, y)
```

We will be using the df_merged dataset we made earlier

In [42]:
```
df_merged.plot(x="Fiscal Year", y="current_amount")
```

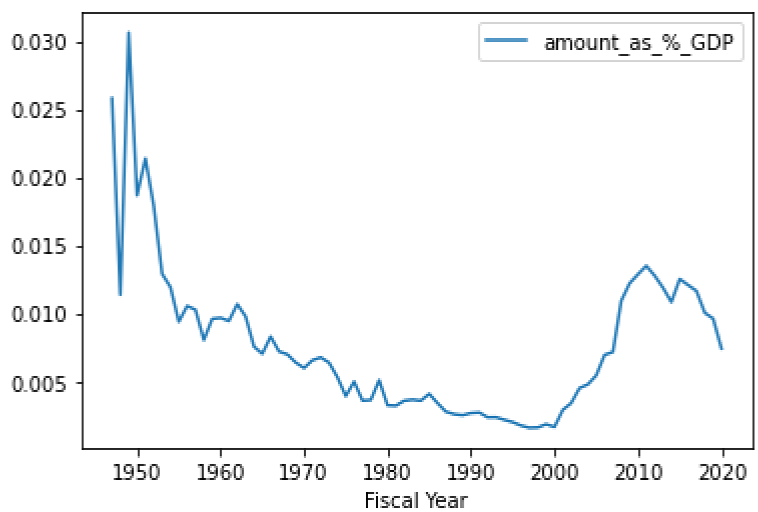
Out[42]: <AxesSubplot:xlabel='Fiscal Year'>



In [43]:

```
df_merged.plot(x="Fiscal Year", y="amount_as_%_GDP")
```

Out[43]: <AxesSubplot:xlabel='Fiscal Year'>



We can change the style of the graph a little bit using matplotlib's built in styles. One of them is called "fivethirtyeight" as it attempts to mimic the style of graphs from the FiveThirtyEight website.

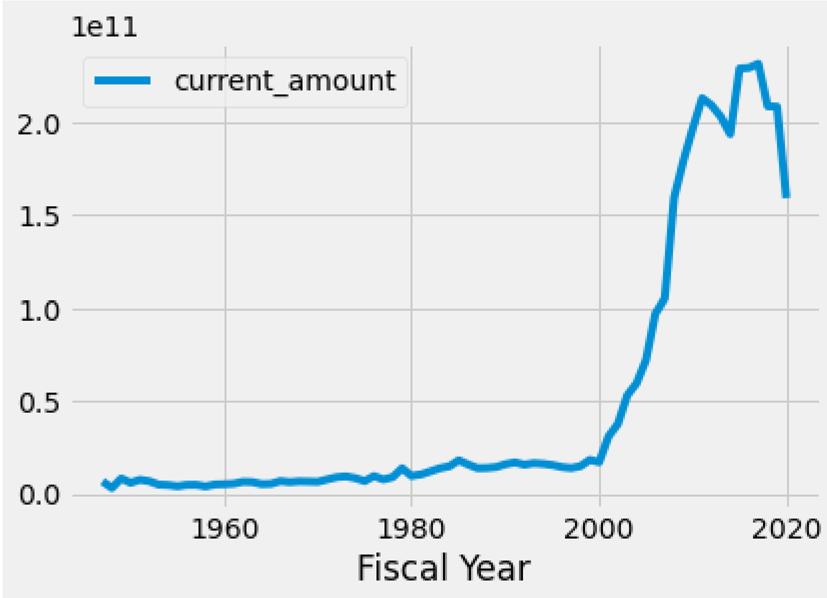
In [44]:

```
import matplotlib
matplotlib.style.use('fivethirtyeight')
```

In [45]:

```
df_merged.plot(x="Fiscal Year", y="current_amount")
```

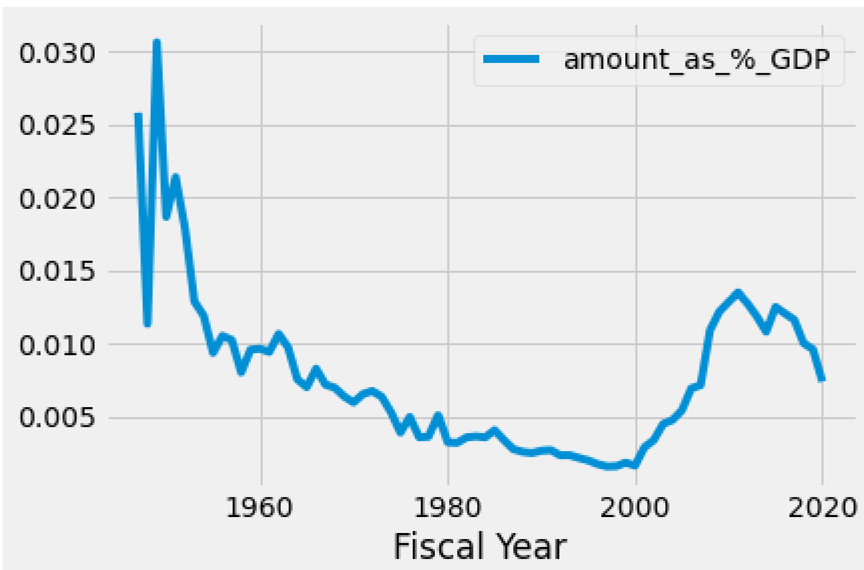
Out[45]: <AxesSubplot:xlabel='Fiscal Year'>



In [46]:

```
df_merged.plot(x="Fiscal Year", y="amount_as_%_GDP")
```

Out[46]: <AxesSubplot:xlabel='Fiscal Year'>



In [ ]: