Finance with Python
Jan Schneider

For best viewing experience open this document in a "two page view" (and use a big monitor . . .).

1 Our Work Environment: JupyterLab

1.1 Installing JupyterLab

In this book we use Python to ananlyze financial data. We are going to run our ananlysis in an open-source software environment called Jupyterlab. The Anaconda Distribution provides a simple way to install Python and JupyterLab. Download Ananconda from here:

https://www.anaconda.com/products/individual

If you see multiple download buttons select the grapical installer for the most recent Python version (Figure 1). Open the installer you just downloaded (.pkg or .exe file) and follow the guided installation.

Note for Windows users: unfortunately some of you might run into technical issues when you try to install packages from JupyterLab (such as “Environment not found” or other permission problems). Many of these problems result from Windows putting a white space in your username ("C:\Users\Johnny Ramone\..."). To prevent these problems, try installing Anaconda in a directory that does not contain any spaces (when you run the Anaconda installer and it prompts you to choose a destination folder, choose a folder that has no spaces; if you don’t see a destination folder you might have to abort the installation and then restart the process by right-clicking on the installer and select “run as administrator").

You can move the installer to the trash once you are done.
That’s it!

Now open the Anaconda application and click on the *Launch* button for JupyterLab (the left button of three Launch buttons in Figure 2).
JupyterLab will open in your browser. The currently supported browsers are Firefox, Chrome and Safari.

When you open JupyterLab you will initially see a "Launcher" window (Figure 3). Here you can launch various types of documents. To start a new notebook select "Python 3" under "Notebook".
JupyterLab will create the new notebook in your current working directory (in Figure 3 this is "Desktop/..."). You can use the panel on the left side of JupyterLab to navigate to a different directory.

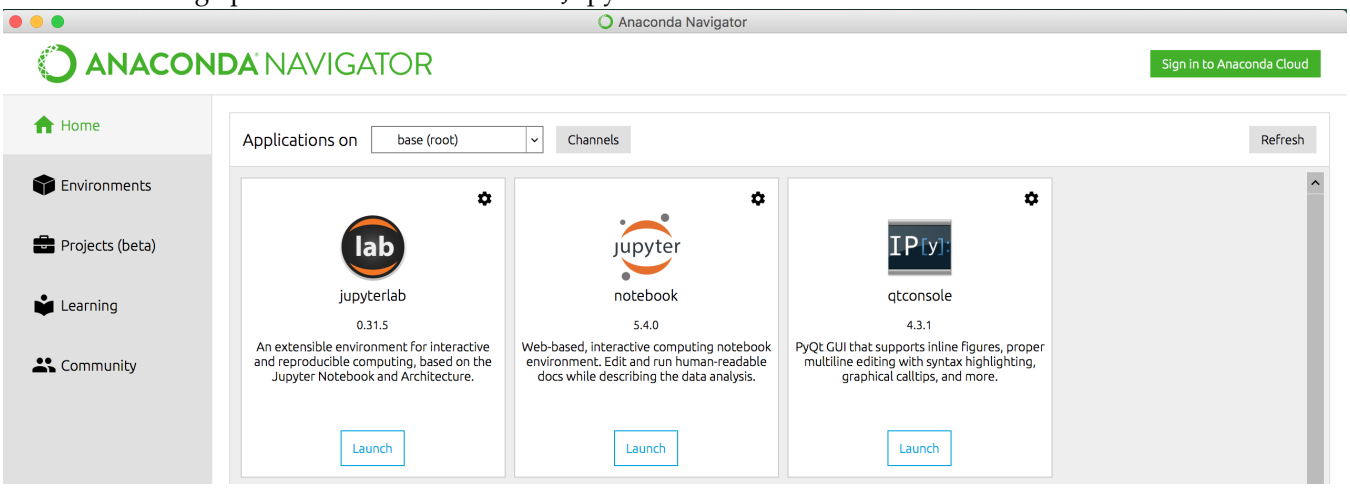
Now you have an empty notebook called "Untitled.ipynb" (Figure 4). You can rename this notebook by right-clicking on its title. You can use the cell inside the notebook to run Python code. For example type

```
1+1
```

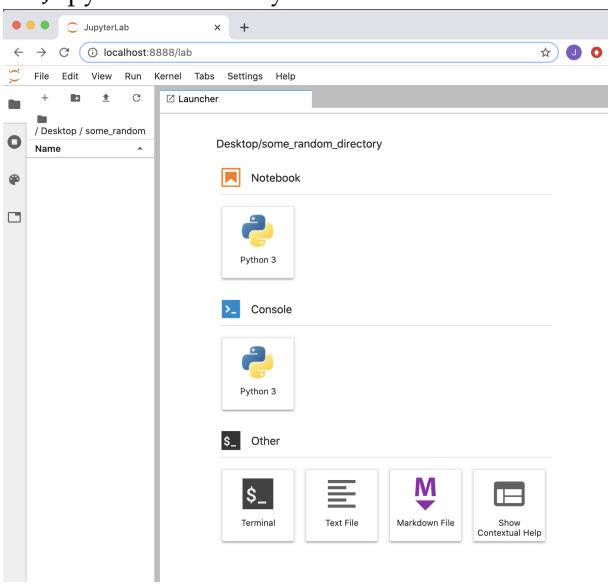
and press the "run" button (▶ in the menu of your notebook) to see the result.

1 Download Ananconda with latest Python version from here:

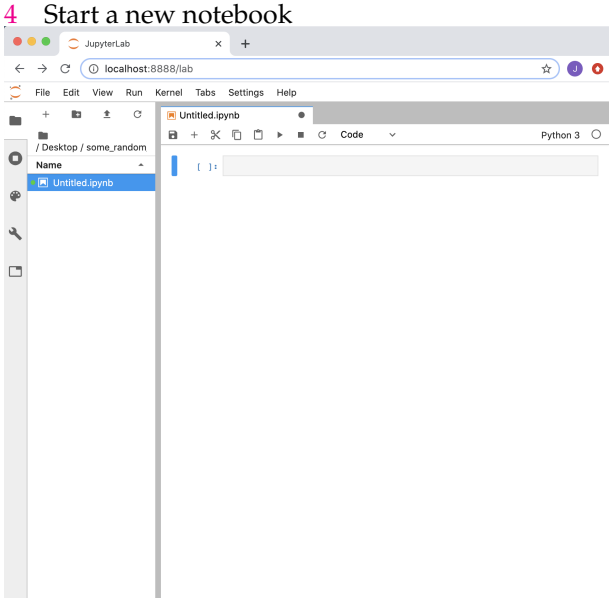
2 After intalling open Ananconda and launch JupyterLab:



3 JupyterLab runs in your browser



4 Start a new notebook



1.2 Working with JupyterLab

Start JupyterLab and generate a new notebook and right-click on "Untitled.ipynb" to rename it (see page 1).
Now you have an empty notebook with a single cell (Figure 1).
JupyterLab has three types of cells that we can select in the top menu: *Code*, *Markdown* and *Raw*.
To write some text select *Markdown* from the top menu and write:

```
## First Notebook
This is our ***first notebook***. Let's calculate 1+1:
```

(Figure 2). When you press the "Run" (▶) button in the top menu, the cell frame disappears and you just see your rendered text and JupyterLab also automatically creates a new "Code" cell (Figure 3).
Now type

```
1+1
```

in the code cell and run the cell to see the result (Figure 4).
Code cells can contain multiple lines of code. Jupyter generally shows us the output of the last line in the cell. For example, run the following cell:

```
1+1                                     # 2 (we don't see this result).
2*3                                     # 6 (we don't see this result).
2**3                                    # Cell output: 8 (2 to the power of 3).
```

(The text after the # symbol is a comment. Comments are for us (the readers of this code), Python ignores all comments.)
Alternatively we can also tell Jupiter explicitly to print various results:

```
print(1+1)                              # Prints '2'.
print('hello')                          # Prints 'hello'.
print('2 to the power of 3 =', 2**3)    # Prints '2 to the power of 3 = 8'.
```

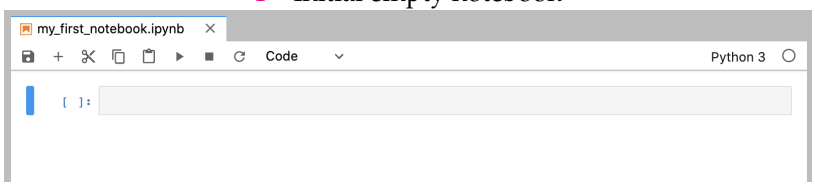
(Figure 5).

Every time we run the bottom cell of a notebook, Jupyter automatically creates a new cell below. Alternatively we can also manually create new cells with the [+] menu button. This new cell will appear below the current active cell (the cell highlighted in blue). We can change the order of the cells by pointing the cursor to the left of a cell and dragging it up or down.
Here are some useful keyboard shortcuts for common operations:

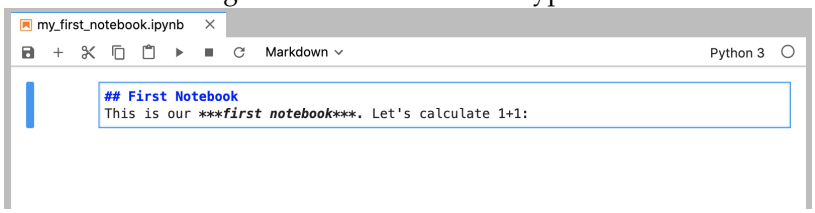
- press [enter] to enter a cell
- press [esc] to exit cell
- if you are inside a cell press: [shift] + [enter] to run the cell
- outside cell: press [m] to switch to markdown and [y] to switch to code
- outside cell: press [x] to delete cell
- outside cell: press [a] / [b] to insert new a cell above / below

Take a few minutes to practice these shortcuts, they will increase your productivity significantly!

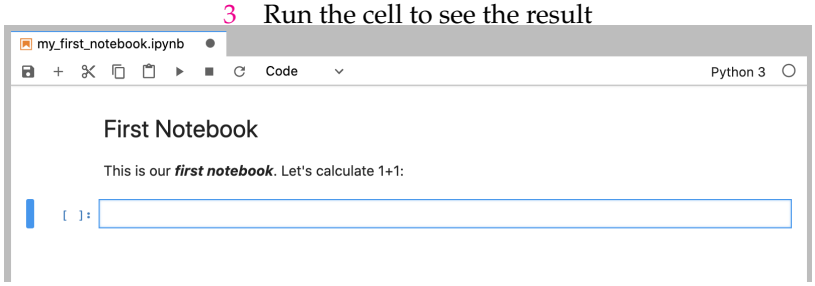
1 Initial empty notebook



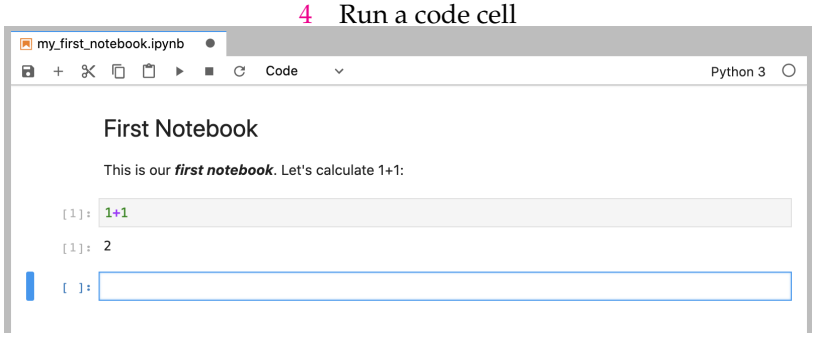
2 Change cell to Markdown and type some text



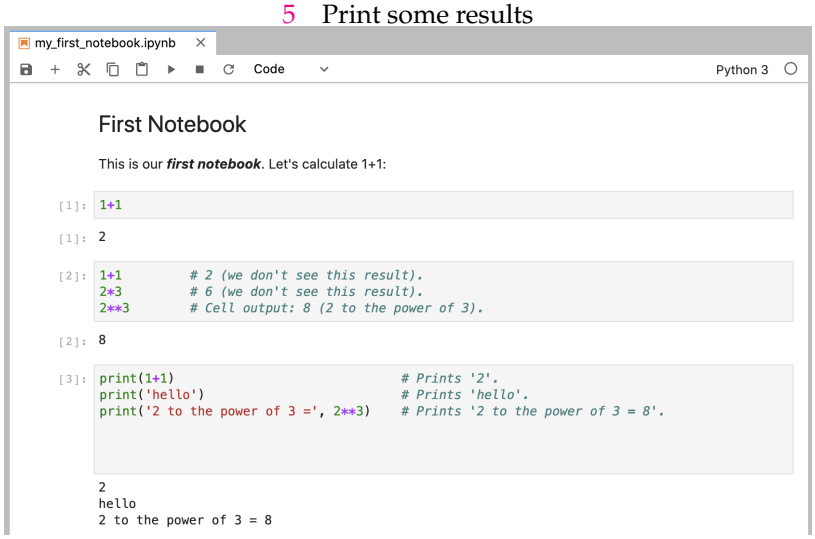
3 Run the cell to see the result



4 Run a code cell



5 Print some results



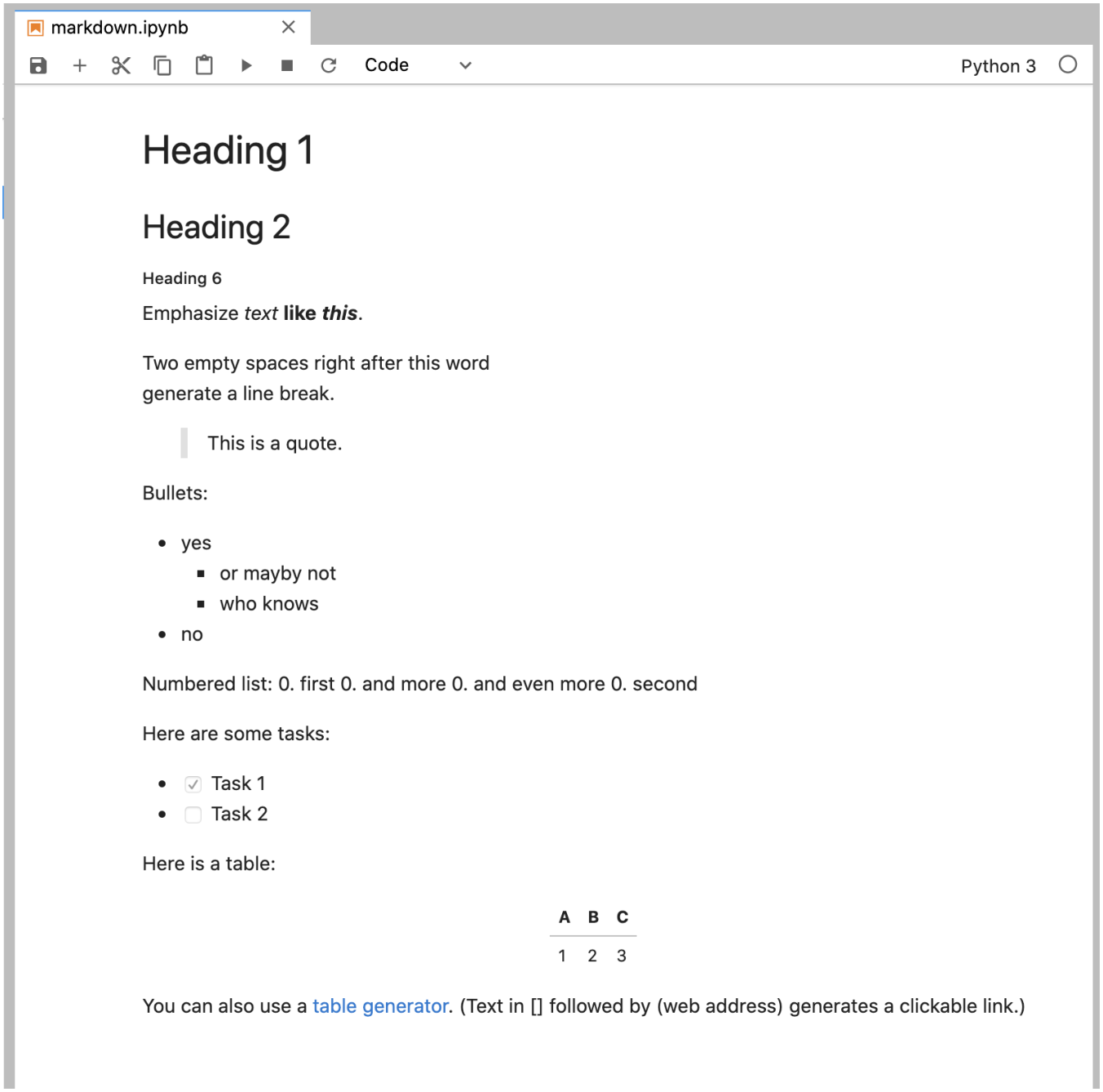
1.3 Formatting Text with Markdown

In JupyterLab we can use cells to write text or to write code.
To write text click on a cell and select Markdown from the top menu.
Markdown provides easy-to-use rules to format text. For example you can generate the output on the right page with the following text in a markdown cell:

```
# Heading 1
## Heading 2
###### Heading 6

Emphasize *text* **like** ***this***.

Two empty spaces right after this word
generate a line break.

> This is a quote.

Bullets:
- yes
    - or mayby not
    - who knows
- no

Numbered list:
0. first
    0. and more
    0. and even more
0. second

Here are some tasks:
- [x] Task 1
- [ ] Task 2

Here is a table:

| A | B | C |
|---|---|---|
| 1 | 2 | 3 |

You can also use a [table generator](https://www.tablesgenerator.com/markdown_tables).
(Text in [] followed by (web address) generates a clickable link.)
```

You can also write markdown outside of Jupyterlab using other online editiors (for example https://stackedit.io/) or look for one of the many free editors to download.

Type the text on page 5 into a markdown cell and then press shift + enter and cell will look like this:



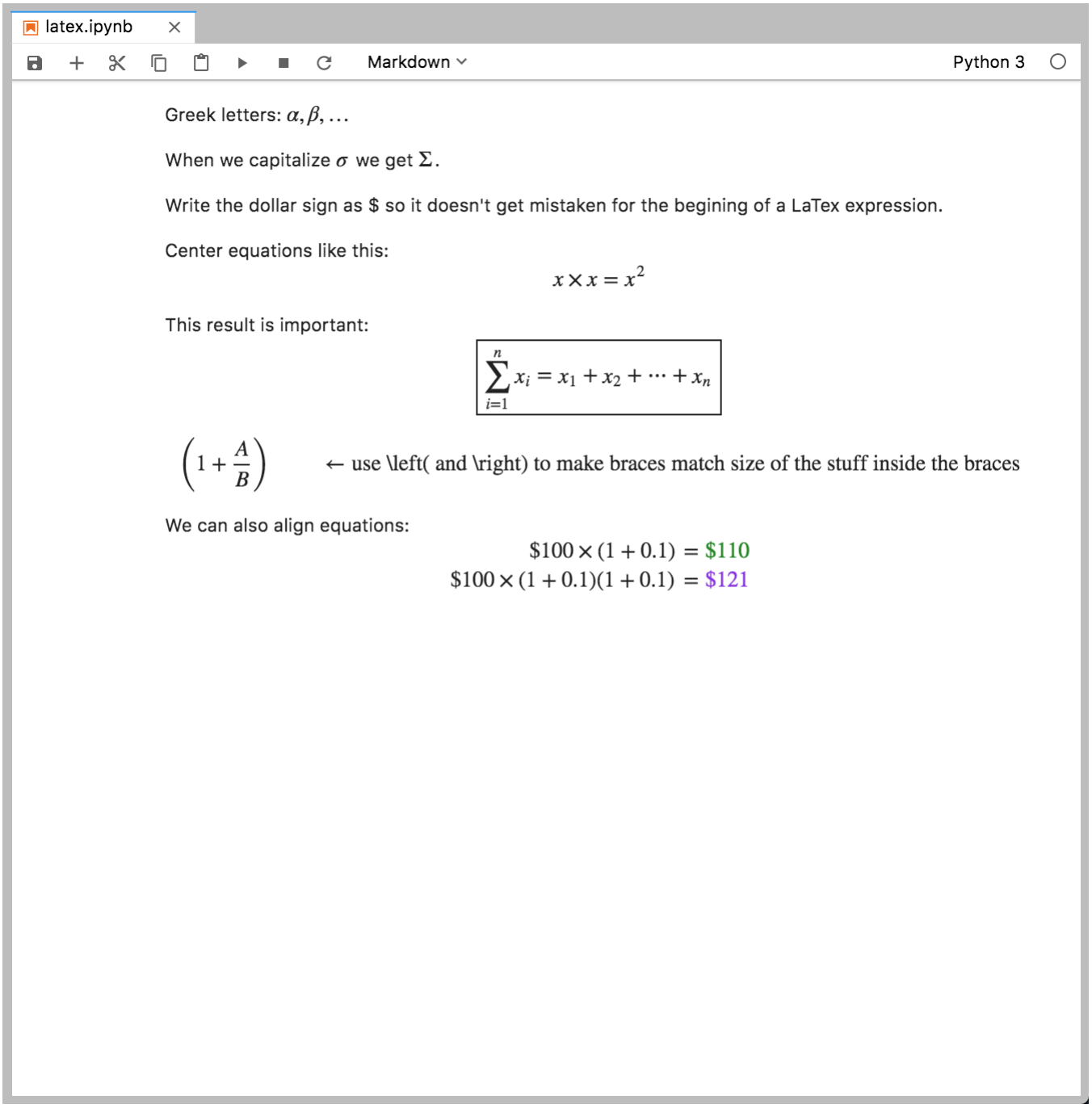
1.4 Beautiful Math with LaTex

Finance is all about numbers and we often express financial concepts with the help of mathematical equations. We can use LaTex in our markdown cells to write professional looking mathematical formulas:

```
Greek letters: $\alpha, \beta, \ldots$

When we capitalize $\sigma$ we get $\Sigma$.

Write the dollar sign as \$ so it doesn't get mistaken for the begining of a LaTex expression.

Center equations like this:
$$
x \times x = x^2
$$

This result is important:
$$
\boxed{ \sum_{i=1}^n x_i = x_1 + x_2 + \dots + x_n }
$$

$$
\left( 1+\frac{A}{B} \right) \hspace{1cm} \leftarrow
\text{use \left( and  \right) to make braces match size of the stuff inside the braces}
$$

We can also align equations:
\begin{align*}
\$100\times(1+0.1) &= \color{ForestGreen}{\$110}\\
\$110\times(1+0.1) &= \color{BlueViolet}{\$121}
\end{align*}
```

You can do a lot more with LaTex than just writing equations. For example the entire document you are reading right now is written in LateX.

Type the text on page 7 into a markdown cell and then press shift + enter and cell will look like this:



2 Time Value of Money

2.1 Python Variable Assignments

We can use Python for standard mathematical operations:

```
1 + 1           #  2
2 * 3           #  6
2 ** 3          #  8 (2 to the power of 3)
```

(Python ignores the comments following the "#").
In spreadsheet programs like Microsoft Excel we reference values by location (for example cells A1 + B2).
In Python we reference values by giving them names:

```
price    = 40
n_shares = 5
Price    = 60   # Names are case-sensitive (this assignment does not change the variable 'price').
```

(You need to run this cell (shift + enter) so that the program becomes aware of it, just typing the code is not enough).
If you run a cell with multiple lines, JupyterLab automatically outputs the result of the last line unless the last line is an assignment. If you want to see the content of a variable just type it in the last line like this:

```
value = n_shares * price     #  Multiply 5 by 40 and assign the result to the variable "value".
value                        #  Show the content of value.
```

(check Figure 1 to see how cell [1] outputs only the last result (the number 8), cell [2] produces no output since the last line is an assignment, and cell [3] outputs the content of "value" (the number 200)).
We can also explicitly print results:

```
print(price)
print(n_shares)
```

Or combine text and variables when we print:

```
print("We own", n_shares, "shares at", price, "dollar per share.")
```

(check the output of cell [5] in Figure 1).

1 Jupyter Notebook

variables.ipynb

Code Python 3

Assigning Variables

Calculate something:

[1]:
```
1 + 1          #  2
2 * 3          #  6
2 ** 3         #  8 (2 to the power of 3)
```

[1]:
```
8
```

Assign some variables:

[2]:
```
price    = 40
n_shares = 5
Price    = 60   # Names are case-sensitive (this assignment does not change the variable 'price').
```

Assign a variable and show the output:

[3]:
```
value = n_shares * price    #  Multiply 5 by 40 and assign the result to the variable "value".
value                       #  Show the content of value.
```

[3]:
```
200
```

Print something:

[4]:
```
print(price)
print(n_shares)
```

```
40
5
```

[5]:
```
print("We own", n_shares, "shares at", price, "dollar per share.")
```

```
We own 5 shares at 40 dollar per share.
```

2.2 Present and Future Values

Suppose today we purchase a piece of land for \$100 (or multiply by 1000 to make the numbers more realistic). Suppose next year we sell the land for \$120. Then our investment earnes 20% return:

$$\$100 \times (1 + 0.2) = \$120 \qquad \Longleftrightarrow \qquad r = \frac{\$120}{\$100} - 1 = 0.2 = 20\%$$

Suppose we invest the proceeds of \$120 into another property and then, one year later, sell this property for \$132. Now the total value of our investment is given by:

$$\begin{aligned} \text{year 1:} \quad & \$100 \times (1 + 0.2) = \$120 \\ \text{year 2:} \quad & \$120 \times (1 + 0.1) = \$132 \end{aligned}$$

We can combine these two equations and compound the initial investment by two years like this:

$$\$100 \times (1 + 0.2) \times (1 + 0.1) = \$132$$

Suppose we have an investment that earns a constant annual return r. If V_t is our account value at the end of year t then we have in general:

$$\boxed{V_t = V_0 \times (1 + r)^t}$$

We can also reverse this equation:

$$V_0 = \frac{V_t}{(1 + r)^t}$$

Example: with 10% annual interest how much do we need to invest today to receive \$121 in two years from now? Answer:

$$\$121/1.1^2 = \$100$$

For this type of calculation we sometimes call the \$121 the future value (FV), the \$100 the present value (PV) and the 10% the discount rate.
Note how we call the 10% a *return* when we look forward (calculate the future value) and we call the same number a *discount rate* when we look backwards (calculate the present value).
Example: if we invest \$100 at 5% per year, how long do we need to double our investment?
We can apply the log to solve for the exponent:

$$V_t = V_0 \times (1 + r)^t \qquad \Longleftrightarrow \qquad \log(V_t) = \log(V_0) + t \times \log(1 + r)$$

We can implement the log function in Python with the *numpy* library.
Start by importing the library:

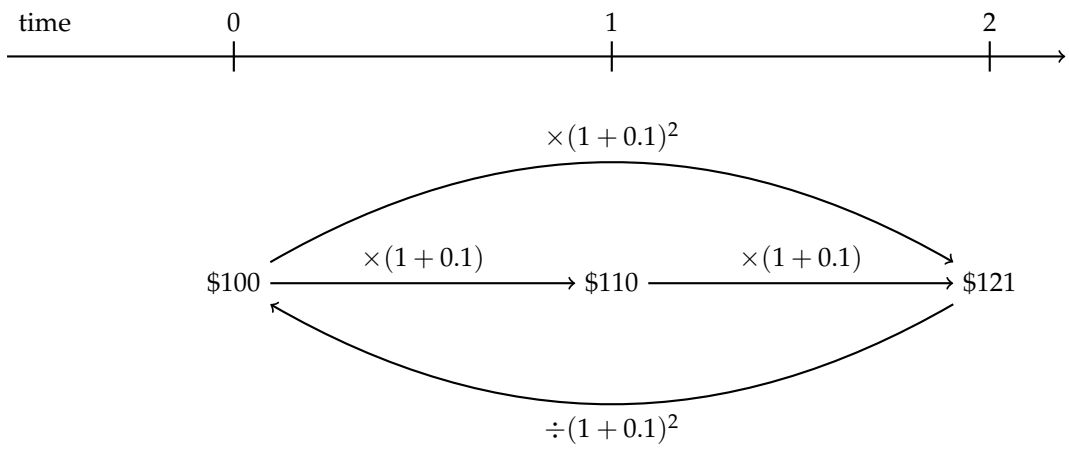
```
import numpy as np    # Import numpy library and abbreviate as "np".
```

(you need to import the library every time to run a new notebook or restart an existing notebook).
And now find the number of years:

```
r  = 0.05
v0 = 100
vt = 2 * v0

t = np.log(vt/v0) / np.log(1+r)
t                                    # Show the result
```

Figure 2 shows that it will take about 14 years.

1 Returns versus discount rates



2 Numpy library

present_future_value.ipyr

Code

Python 3

Present Value - Future Value

Import the numpy library:

[1]:
```
import numpy as np    # For scientific computing.
```

How many years to double our investment?

[2]:
```
r  = 0.05
v0 = 100
vt = 2 * v0

t = np.log(vt/v0) / np.log(1+r)
t                                    # Show the result
```

[2]:
```
14.206699082890461
```

[4]:
```
np.irr([-100,10,10,110])
```

2.3 Sequences of Cash Flows

Suppose you would like to withdraw \$10 at the end of each year for the next 3 years from your account. With 10% interest, how much money do you need to invest right now to cover your future withdrawals?
We call such a sequence of fixed cash flows an annuity. The initial investment (or present value) for an annuity is the sum of the present values of the individual cash flows:

$$PV = \frac{10}{(1+r)^1} + \frac{10}{(1+r)^2} + \frac{10}{(1+r)^3}$$

In general we have for the present value of an annuity with future cash flows CF:

$$\boxed{PV = \frac{CF}{r}\left(1 - \frac{1}{(1+r)^t}\right)}$$

(see the derivation in Table 1.) So for our example:

$$\frac{\$10}{0.1}\left(1 - \frac{1}{1.1^3}\right) = \$24.87$$

Table 2 shows how an initial investment of \$24.87 will generate exactly three annual \$10 cash flows.
Alternatively, suppose you plan to live forever and would like to withdraw \$10 from your account each year until the end of the universe. With 10% interest rate, how much do you need to invest initially?
We might be able to guess this right away - the answer has to be \$100 since then each year we withdraw exactly what we earn and so our wealth never changes.
We call such an infinite sequence of fixed cash flows a perpetuity.
Since for a perpetuity the cash flows have to equal the interest payments, we have:

$$PV \times r = CF \qquad \Longrightarrow \qquad \boxed{PV = \frac{CF}{r}}$$

(we also get this result by taking t to infinity in the annuity formula above.) In our case:

$$PV = \frac{\$10}{0.1} = \$100$$

Now suppose we initially invest \$100, and at the end of each year we withdraw \$10 from our account and after three years we we are left with \$100 after withdrawing the third \$10. What is the interest rate that makes this strategy possible? To find the interest rate we need set the initial investment equal to the discounted future cash flows like this:

$$\$100 = \frac{\$10}{(1+r)^1} + \frac{\$10}{(1+r)^2} + \frac{\$110}{(1+r)^3}$$

The rate r that solves this equation is called internal rate of return (IRR).
For this example we already know the answer has to be 10%, but in general we cannot solve for r analytically. We can find the IRR numerically in Python using the irr function in the *numpy financial* library:

```
import numpy_financial as npf          # Import numpy financial library and abbreviate as "npf".

r = npf.irr([-100,10,10,110])          # Internal rate of return (initial investment is a cash outflow).
```

If we run this code we see that the IRR is 10% consistent with our previous calculations.

1 Present value formula for annuity

$$PV = \frac{10}{(1+r)^1} + \frac{10}{(1+r)^2} + \frac{10}{(1+r)^3}$$

multiply by $1+r$:

$$PV \times (1+r) = 10 + \frac{10}{(1+r)^1} + \frac{10}{(1+r)^2}$$

equation 2 minus equation 1:

$$PV \times r = 10 - \frac{10}{(1+r)^3}$$

Divide by r :

$$PV = \frac{10}{r}\left(1 - \frac{1}{(1+r)^3}\right)$$

2 Year-end account values for annuity (intial value = \$24.8685)

year	account value before withdrawal	withdrawal	account value after withdrawal
1	$24.8685 \times 1.1 = 27.3554$	10	17.3554
2	$17.3554 \times 1.1 = 19.0909$	10	9.0909
3	$9.0909 \times 1.1 = 10.0000$	10	0

3 Year-end account values for perpetuity (intial value = \$100)

year	account value before withdrawal	withdrawal	account value after withdrawal
1	$100 \times 1.1 = 110$	10	100
2	$100 \times 1.1 = 110$	10	100
3	$100 \times 1.1 = 110$	10	100
$\vdots$			

2.4 Nominal vs Real Returns

Suppose a cafe latte currently costs \$5 at your local coffee shop and the inflation rate is 4% per year.
If the price of the latte increases at the rate of inflation, it will cost \$5.20 in one year from now:

```
i  = 0.04            # Annual inflation rate.
p0 = 5               # Price today.
p1 = p0 * (1+i)      # Price in 1 year (5 * 1.04 = 5.20)
```

(in the U.S., the Bueau of Labor Statistics estimates inflation as the 12-month change of Consume Price Index, see Figure 1).
Suppose you invest \$100 for one year and you earn a return of 30%:

```
r  = 0.3             # Annual return.
v0 = 100             # Account value today.
v1 = v0 * (1+r)      # Account value in 1 year (100 * 1.3 = 130).
```

We also call the 30% a nominal return because this return only measures the growth of the dollar value of our account without considering the purchasing power (how much we can buy).
How many lattes can we buy?

```
n0 = v0 / p0         # Number of lattes we can buy today     (100/5.0 = 20).
n1 = v1 / p1         # Number of lattes we can buy next year (130/5.2 = 25).
```

So the number of lattes we can purchase next year versus today increases by $\frac{130/5.2}{100/5} - 1 = 25/20 - 1 = 0.25 = 25\%$.
We call the 25% the real return of our investment. So we have in general:

$$1 + \text{ real return} = \frac{\text{purchasing power at } t_1}{\text{purchasing power at } t_0} = \frac{v_1/p_1}{v_0/p_0} = \frac{v_1/v_0}{p_1/p_0} = \frac{1+r}{1+i} = \frac{1+\text{nominal return}}{1+\text{inflation rate}}$$

For small rates we can approximate the real returns as:

$$\text{real return} = \frac{1+\text{nominal return}}{1+\text{inflation rate}} - 1 \approx \text{nominal return} - \text{inflation rate}$$

For example, if the nominal return is 10% and the inflation rate is 2%, we have:

```
real_return               = (1+0.1) / (1+0.02) - 1        # 0.07843
real_return_approximation = 0.1 - 0.02                    # 0.08
```

1 Bureau of Labor Statistics

An official website of the United States government Here is how you know

United States Department of Labor

U.S. BUREAU OF LABOR STATISTICS

Follow Us | Release Calendar | Blog

Search BLS.gov

HOME | SUBJECTS | DATA TOOLS | PUBLICATIONS | ECONOMIC RELEASES | CLASSROOM | BETA

Consumer Price Index

Search Consumer Price Go

CPI Home | CPI Publications | CPI Data | CPI Methods | About CPI | Contact CPI

Bureau of Labor Statistics > Consumer Price Index > Home

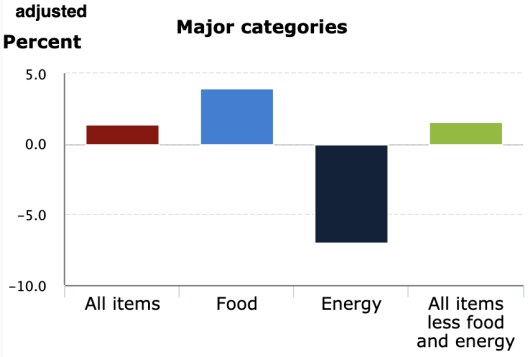
The **Consumer Price Index (CPI)** is a measure of the average change over time in the prices paid by urban consumers for a market basket of consumer goods and services. Indexes are available for the U.S. and various geographic areas. Average price data for select utility, automotive fuel, and food items are also available.

NOTICES

- Correction of seasonally adjusted values for special relative series Read More »
- Updated seasonal factors to be introduced February 8, 2021 Read More »
- The impact of the coronavirus (COVID-19) pandemic on CPI data Read More »

CHARTS

12-month percentage change, Consumer Price Index, selected categories, December 2020, not seasonally adjusted



Source: U.S. Bureau of Labor Statistics.

NEWS RELEASES

CPI for all items rises 0.4% in December as gasoline index increases

01/13/2021

In December, the Consumer Price Index for All Urban Consumers rose 0.4 percent on a seasonally adjusted basis; rising 1.4 percent over the last 12 months, not seasonally adjusted. The index for all items less food and energy rose 0.1 percent in December (SA); up 1.6 percent over the year (...

HTML | PDF | RSS | Charts | Local and Regional CPI

NEXT RELEASE

January 2021 CPI data are scheduled to be released on February 10, 2021, at 8:30 A.M. Eastern Time.

read more »

2.5 Arithmetic vs Geometric Average

Suppose we invest \$100 in a stock. Now consider the following two scenarios:

1. the stock price first goes up 30% and then drops 10%
2. the stock price first drops 10% and then goes up 30%

Table 1 shows our account values at time 0, 1 and 2, and Figure 2 shows a plot of these values. In both scenarios we arrive at a final account value of \$117 (the order of multiplication does not matter).
So the compound return is given by:

$$(1+0.3)\times(1-0.1) = (1-0.1)\times(1+0.3) = 1.17 \quad \Longrightarrow \quad 1.17-1 = 0.17 = 17\%$$

Calculate average return:

$$am = \frac{0.3+(-0.1)}{2} = 0.1 = 10\%$$

So we earn 10% on average. We also call this number the arithmetic mean. If we compound the arithmetic mean we get

$$1.1\times 1.1 - 1 = 1.21 - 1 = 21\%$$

which is greater than the 17% compound value of the original returns.
If we compound the geometric mean we get the same return as the compounded original returns:

$$(1+gm)^2 = 1.3\times 0.9 \quad \Longrightarrow \quad gm = \left(1.3\times 0.9\right)^{\frac{1}{2}} - 1 = 0.081665 = 8.1665\%$$

Implement these calculations in Python:

```
up   = 1.3
down = 0.9

am = (up + down) / 2                          #  Arithmetic mean.
gm = (up * down) ** (1/2) - 1                 #  Geometic mean.
```

Table 3 and Figure 4 show how the compound arithmetic average overstates the actual investment performance. In genenral, the arithmetic mean is only equal to the geometric mean if returns are constant. If returns are not constant, the arithmetic mean is always greater than the geometric mean and the difference between these to values increases with the volatility of the returns.

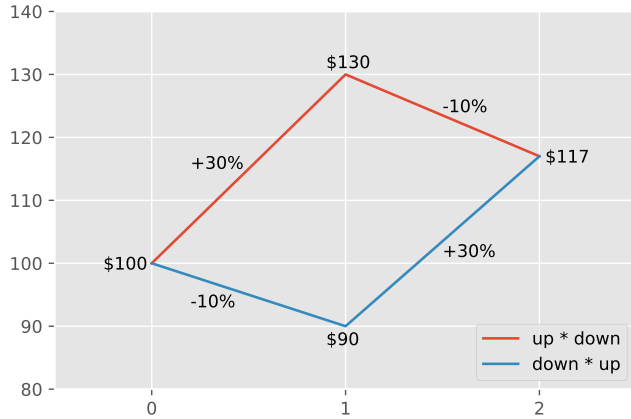
1

time	up * down	down * up
0	100.0	100.0
1	130.0	90.0
2	117.0	117.0

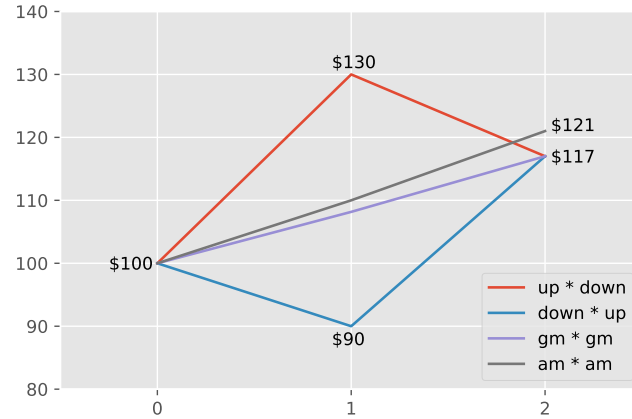
3

time	up * down	down * up	gm * gm	am * am
0	1.00	1.00	1.000000	1.00
1	1.30	0.90	1.081665	1.10
2	1.17	1.17	1.170000	1.21

2



4



2.6 Annual Percentage Rate (APR)

Compare the following two investments:

A you pay \$80 now and get \$84 in 6 months from now

B you pay \$80 now and get \$86 in 8 months from now

Returns of these investments:

```
rA = 84 / 80 - 1          # 0.05  = 5%
rB = 86 / 80 - 1          # 0.075 = 7.5%
```

Which investment is better? This is difficult to say because we earn the returns over different time horizons.
In finance we typically compare returns (or any other types of rates) by annualizing them.
For example, suppose we could reinvest our money in scenario A for six additional months at the same return.
Then our 12-month return in scenario A is given by:

```
(1 + rA) ** 2             # 0.1025 = 12.25%
```

Accordingly, the annual return of investment B is given by:

```
(1 + rB) ** (12/8)        # 0.1146 = 11.46%
```

We also call these returns the effect rate annual rate (EAR) (somtimes also called APY).
Alternatively, we sometimes also ignore compounding when we annualize:

```
(1 + rA) * 2              # 0.10   = 10%
(1 + rb) * (12/8)         # 0.1125 = 11.25%
```

(we would earn these returns if we do not reinvest the interest payments).
We call these returns the annual percentage rate (APR).
Banks typically quote their rates as APRs. For example in Figure 1 we see a credit card that charges a variable APR between 14.24% and 24.27% (variable means that the rate changes based on some index like the Prime rate). For example, suppose the APR is currently 24%. Then you pay a monthly interest rate of

$$\frac{0.24}{12} = 0.02 = 2\%$$

That means a \$1 loans will turn after one year into

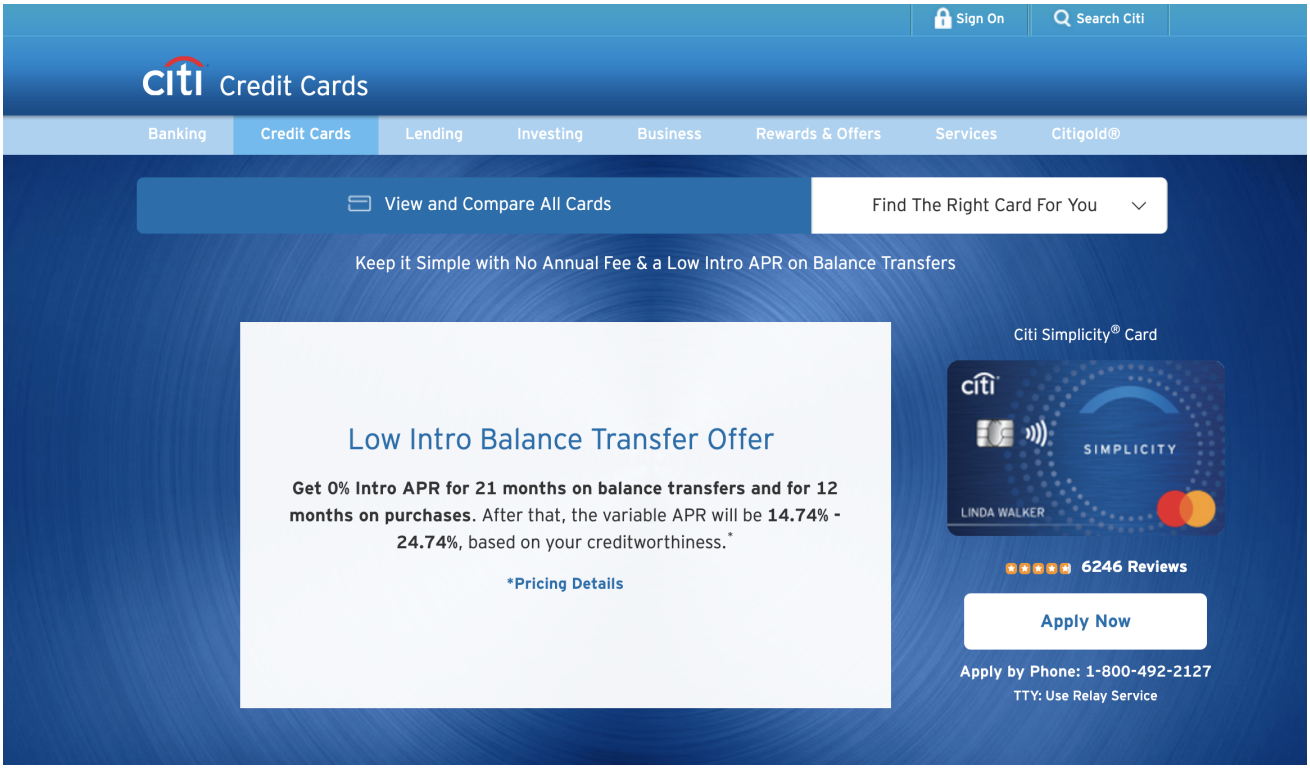
$$(1 + 0.02)^{12} = \$1.2682$$

So our effect rate annual rate (EAR) is given by

$$1.2668 - 1 = 0.2682 = 26.28\%$$

The 24% APR ignores the effect of compounding and the 2.82% difference between EAR and APR is the interest on interest resulting from the monthly compounding.

1 Credit card APR

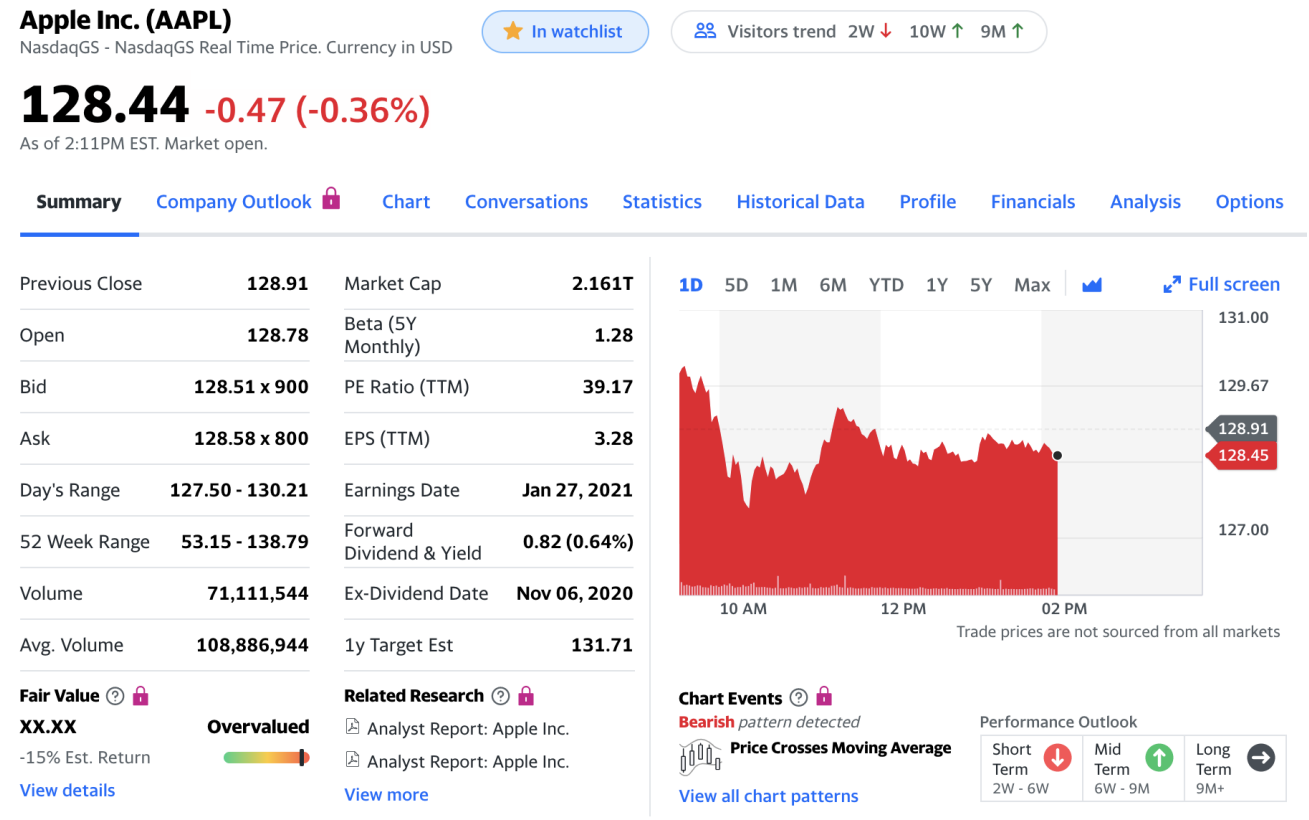


3 Basic Python

3.1 Publicly Traded Companies

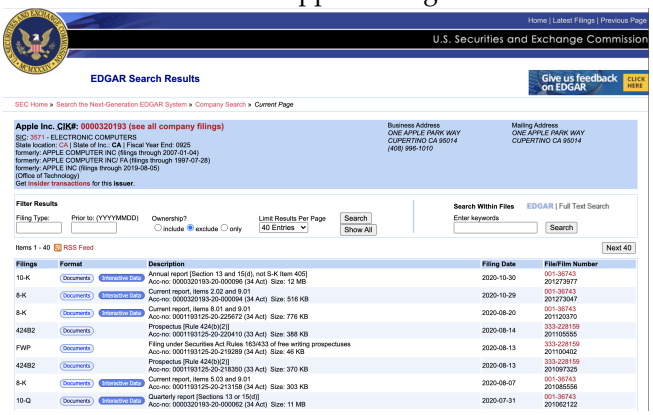
Figure 1 shows information about Apple stock on Yahoo Finance.
Apple's shares traded on the Nasdaq stock exchange under the ticker symbol AAPL.
We can see that Apple's earnings per share (EPS) was $3.28 (ttm stands for "trailing twelve months", so this is the income Apple generated over the last 12 months).
How does Yahoo (or any data provider) know the earnings of Apple?
All publicly traded companies are required to file quarterly financial statement with the Security and Exchange Commission (SEC).
We can find these filings on the SEC website at https://www.sec.gov/edgar/searchedgar/companysearch.html (Figure 2).
For example, Figure 3. shows the most recent filings of Apple. The SEC identifies firms with a "central index key" (CIK). Figure 2 and 3 show that Apple has the CIK 320193.

1 Yahoo Finance website: Apple



2 SEC: search for Apple

3 SEC: Apple's filings



3.2 Python Objects

Suppose a company has 200 million shares outstanding and generates $500 million quarterly earnings:

```
earnings = 500                  # 500 million dollar
shares   = 200                  # 200 million shares outstanding
```

Here we create two objects (the numbers 500 and 200) and refer to them as "earnings" and "shares".
These objects are now stored somewhere in the memory of our computer.
We can find the memory address of "earnings" like this:

```
id(earnings)                    # Get the "identity" (memory address) of earnings.
```

(your result will differ from the number in Figure 1).
We can find the type of "earnings" like this:

```
type(earnings)                  # int
```

So "earnings" is an integer. Let's calculate the earnings per share:

```
eps = earnings / shares         # 2.5

type(eps)                       # float
```

This object is a float (it has a decimal point).
Suppose next quarter the earnings increase by $100 million:

```
earnings + 100                  # 500 + 100
earnings                        # Check the value of earnings => still 500.
```

So the value of "earnings" has not changed. Why not? When we write "earnings + 100", we create a new object (the integer 600). But the name "earnings" still refers to the old object (the integer 500).
To change the value of earnings, we need to reassign it:

```
earnings = earnings + 100       # Assign the name "earnings" to the sum of 500 and 100.
print(earnings, id(earnings))
```

Note how "earnings" has a new id now, indicating that we created a new object (rather than "changing" the old object).
What about "eps"? The variable "eps" refers the number 2.5, and this number did not change. If we want to update "eps", we have to rerun the calculation and reassign the variable name:

```
eps = earnings / shares         # 3.0
print(eps, id(eps))
```

(note how "eps" has a new id now).
We can also reassign variables like this:

```
x  = 1
x += 1        # Now x is 2 (same as: x = x + 1).
```

And accordingly:

```
x *= 4        # Now x is 8 (same as: x = x * 4).
x /= 2        # Now x is 4 (same as: x = x / 2).
```

1 Object references

objects.ipynb

Code Python 3

Python Object References

[1]:
```
earnings = 500                  # 500 million $.
shares   = 200                  # 200 million shares outstanding.
```

[2]:
```
id(earnings)                    # This number changes every time you rerun the cell above.
```

[2]:
```
140310681901200
```

[3]:
```
type(earnings)                  # This is an integer.
```

[3]:
```
int
```

[4]:
```
eps = earnings / shares         # Earnings per share.
eps
```

[4]:
```
2.5
```

[5]:
```
type(eps)                       # This is a floating point number.
```

[5]:
```
float
```

[6]:
```
id(eps)
```

[6]:
```
140310681901584
```

[7]:
```
earnings + 100                  # 500 + 100.
```

[7]:
```
600
```

[8]:
```
earnings                        # Still 500!
```

[8]:
```
500
```

[9]:
```
earnings = earnings + 100       # Now earnings is 600.

print(earnings, id(earnings))
```

```
600 140310681902800
```

[10]:
```
eps                             # eps did not change!
```

[10]:
```
2.5
```

[11]:
```
eps = earnings / shares         # Rerun the calculation and reassign the variable name.

print(eps, id(eps))
```

```
3.0 140310681902384
```

[12]:
```
x  = 1
x += 1                          # Now x is 2 (same as: x = x + 1).
x *= 4                          # Now x is 8 (same as: x = x * 4).
x /= 2                          # Now x is 4 (same as: x = x / 2).
x
```

[12]:
```
4.0
```

3.3 Built-In Functions

Here are some Phython functions:

```
len('abcd')          #  4     (length of object)
round(1.23456, 2)    #  1.23  (round to 2 decimal places)
```

We also refer to these functions as built-in functions since they are built into Python (see Figure 1).
We will see how to define our own custom functions on page 31.
We have already seen the *type()* function on page 23:

```
firm = 'Micro'        # We can also use " " instead of ' '.
type(firm)            # str
```

So 'Micro' is a string. We can combine ("concatenate") strings like this:

```
firm = firm + 'soft'      # 'Microsoft'
```

But we cannot add an integer or float to a string:

```
x = 1                     #  This is an integer.
y = firm + x              #  Not possible. This produces a 'TypeError'.
```

Note how the error message in Figure 2 tells us what error we made (trying to concatenate string and integer) and where the error ocurred (line 2). If we want to add "firm" and "x" we need to convert "x" into a string first like this:

```
y = firm + str(x)   #  Convert the variable x to a string and add to firm ( = 'Micro1').
```

Sometimes functions are attached to specific objects with *object.function()*. We often call these functions methods.
For example, the *.find()* methods finds the index of the first occurrance of a substring:

```
firm.find('o')      #  4 (position of the first 'o')
firm.rfind('o')     #  6 (position of the last 'o' ("reverse find"))
```

In Python counting generally starts at zero ('M' is character zero of 'Microsoft').
Methods are attached to specific objects and we cannot use them on their own like regular functions:

```
max(1,8,5)                          #  8   (this is the max function applied to intergers)
max('azb')                          #  'z' (this is the max function applied to a string)
'azb'.max()                         #  Produces an error message since .max() is not a string method

'abc'.capitalize()                  #  'Abc' (this is the capitalize method)
capitalize('abc')                   #  Produces an error message since Python has no capitalize function
```

We can also "chain" methods like this:

```
'MICROSOFT'.lower().capitalize()     #  'Micosoft'  (first transform to lower case and then capitalize).
```

Note that methods like ".lower()" and ".capitalize()" produce modified copies of strings, but they do not change the original string:

```
firm.lower()        #   Returns a copy of 'Microsoft' with all characters in lower case.
firm                #   still 'Microsoft'
```

If we want to change "firm" we have to reassign it:

```
firm = firm.lower()  #  Now firm is 'microsoft'
```

1 Python documentation, built-in functions

Python » English 3.9.1 Documentation » The Python Standard Library »

Previous topic
Introduction

Next topic
Built-in Constants

This Page
Report a Bug
Show Source

Built-in Functions

The Python interpreter has a number of functions and types built into it that are always available. They are listed here in alphabetical order.

		Built-in Functions		
abs()	delattr()	hash()	memoryview()	set()
all()	dict()	help()	min()	setattr()
any()	dir()	hex()	next()	slice()
ascii()	divmod()	id()	object()	sorted()
bin()	enumerate()	input()	oct()	staticmethod()
bool()	eval()	int()	open()	str()
breakpoint()	exec()	isinstance()	ord()	sum()
bytearray()	filter()	issubclass()	pow()	super()
bytes()	float()	iter()	print()	tuple()
callable()	format()	len()	property()	type()
chr()	frozenset()	list()	range()	vars()
classmethod()	getattr()	locals()	repr()	zip()
compile()	globals()	map()	reversed()	__import__()
complex()	hasattr()	max()	round()	

abs(*x*)

Return the absolute value of a number. The argument may be an integer, a floating point number, or an object implementing `__abs__()`. If the argument is a complex number, its magnitude is returned.

2 Converting Strings

builtin_functions.ipynb

Code Python 3

Builit-in Functions

[1]:
```
firm = 'Micro'
type(firm)
```

[1]: str

[2]:
```
firm = firm + 'soft'
firm
```

[2]: 'Microsoft'

[3]:
```
x = 1 # This is an integer.
y = firm + x # Not possible. This produces a 'TypeError'.
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-3-aef05a2bbf52> in <module>
      1 x = 1 # This is an integer.
----> 2 y = firm + x # Not possible. This produces a 'TypeError'.

TypeError: can only concatenate str (not "int") to str
```

[4]:
```
y = firm + str(x) # Convert the variable x to a string and add to firm ( = 'Micro1').
y
```

[4]: 'Microsoft1'

[5]:
```
firm.find('o')      #  4 (position of the first 'o')
```

[5]: 4

[6]:
```
firm.rfind('o')     #  6 (position of the last 'o' ("reverse find"))
```

[6]: 6

[7]:
```
'MICROSOFT'.lower().capitalize()     #  'Micosoft'  (first transform to lower case and then capitalize).
```

[7]: 'Microsoft'

3.4 Accessing Websites

The SEC maintains a file that contains names, ticker symbols and central index keys (CIKs) of all publicly traded firms. We can find a link to this file at https://www.sec.gov/file/company-tickers (Figure 1). If we click on the link *company_tickers.json*, we see the file in Figure 2.
We can access the document in Figure 2 directly from our notebook, using the Python *requests* library.
Start by importing the library:

```
import requests
```

And now get the file:

```
requests.get('https://www.sec.gov/files/company_tickers.json').text
```

Here *.get()* is a method (it "does something") and *.text* is an attribute (a feature or property). The method *get()* gets the website and the response object (produced by *requests.get()*) has various attributes.
For example, here is the status code attribute:

```
requests.get('https://www.sec.gov/files/company_tickers.json').status_code  # 200
```

When we request a website, the server issues a status code. Code "200" means that the request has been successfully completed.
Let's try to access the Wall Street Journal:

```
requests.get('https://www.wsj.com/').status_code  # 403
```

Code "403" means "forbidden" (the server understood our request but refuses the action).
If we want to work with the content of our request, we need to give it a name:

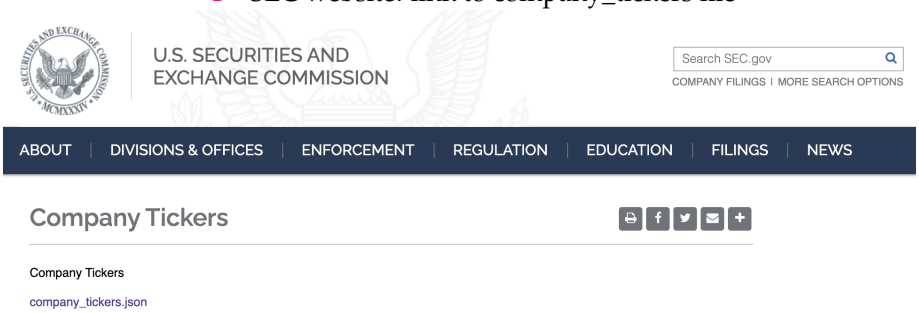
```
symbols = requests.get('https://www.sec.gov/files/company_tickers.json').text
```

What types of objects is this?

```
type(symbols)         # String
```

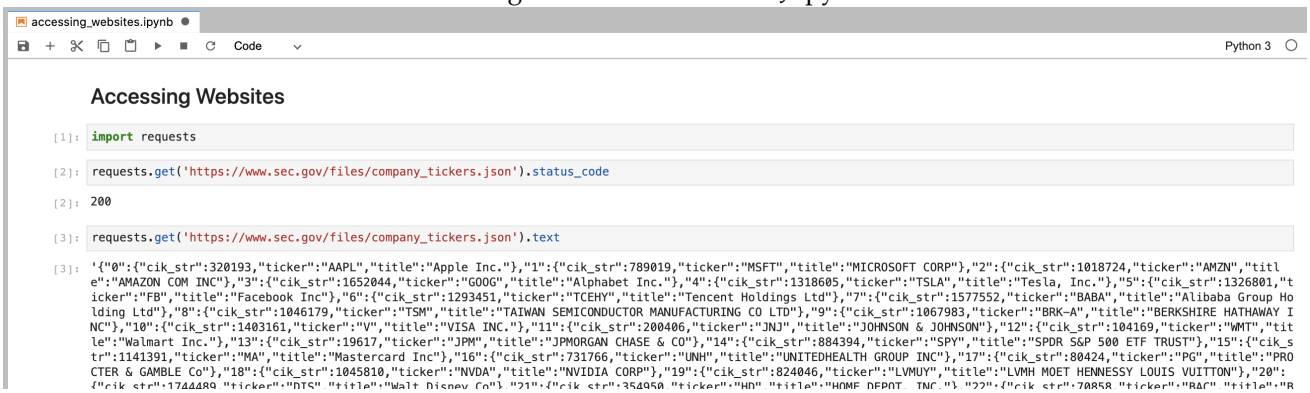
So now we can apply any of the Python string methods to obtain information from the object "symbols".

1 SEC website: link to company_tickers file



2 SEC Website: company_tickers file

3 Accessing the website from our Jupyter notebook



3.5 Strings

Get the company-ticker file from the SEC (see page 27):

```
import requests
symbols = requests.get('https://www.sec.gov/files/company_tickers.json').text
```

Now the variable "symbols" references the string

{"0":{"cik_str":320193,"ticker":"AAPL","title":"Apple Inc."},"1":{"cik_str":789019,"ticker":"MSFT", . . .

Our goal is to find specific firms within this string (for example, what are the ticker and CIK of Facebook?).
We can access parts of strings using square brackets. For example:

```
name     = 'Walmart'     # This is a string.
name[0]  = 'W'           # Position 0.
name[1]  = 'a'           # Position 1.
name[-1] = 't'           # Position -1 (counting backwards from the end).
```

Recall from page 25, counting generally starts at zero ('W' is character zero of 'Walmart').
We can also get a range of characters:

```
name[1:5]                #  'alma' (from 1 to 5, 5 is not included)
name[:4]                 #  'Walm' (from the beginning to 4, 4 is not included)
name[3:]                 #  'mart' (from 3 until the end)
name[-4:]                #  'mart' (from -4 until the end)
```

Let's find the entry for Facebook in our "symbols" string (highlighted in blue in Figure 1).
We can find the first occurance of a substring with the ".find()" method. First index of 'Facebook':

```
i = symbols.find('Facebook')  # 362 (if our string contains multiple 'Facebook', this will be the first).
```

(see Figure 2). Test:

```
symbols[i]                    # 'F' (so i = 362 points to the start of "Facebook"),
```

What is the position of the closing bracket for Facebook? Try:

```
symbols.find('}')             # 59 (position of first '}' in symbols).
```

But this is the first }, so this is the bracket that belongs to Apple (count until 59 in Figure 1).
To find the closing bracket for Facebook, let's restrict our search to the substring that starts with Facebook:

```
symbols[i:]                        # All characters from i = 362 (position of 'F') until the end.
symbols[i:].find('}')              # 13  (index of first } in the substring that starts at i = 362).
last = i + symbols[:i].find('}')   # 362 + 13 = 375 (index of first }, counting from position 0).
```

To find the corresponding opening bracket, we use a similar trick: restrict the search to the substring that ends with Facebook, and then use the "rfind()" method (index of the last occurrence).

```
symbols[:i]                        # All characters until (not including) i (position of 'F').
first = symbols[:i].rfind('{')     # 320 (index of last { in the substring that ends at i - 1 = 361).
```

And now we can get the entire entry for Facebook:

```
symbols[first:last+1]              # All characters from 'first' to 'last' (last+1 is not included).
```

1 SEC: company names and ticker symbols

sec.gov/files/company_tickers.json


```
{"0":{"cik_str":320193,"ticker":"AAPL","title":"Apple Inc."},"1":{"cik_str":789019,"ticker":"MSFT","title":"MICROSOFT CORP"},"2":{"cik_str":1018724,"ticker":"AMZN","title":"AMAZON
COM INC"},"3":{"cik_str":1652044,"ticker":"GOOG","title":"Alphabet Inc."},"4":{"cik_str":1318605,"ticker":"TSLA","title":"Tesla, Inc."},"5":
{"cik_str":1326801,"ticker":"FB","title":"Facebook Inc"},"6":{"cik_str":1293451,"ticker":"TCEHY","title":"Tencent Holdings Ltd"},"7":
{"cik_str":1577552,"ticker":"BABA","title":"Alibaba Group Holding Ltd"},"8":{"cik_str":1046179,"ticker":"TSM","title":"TAIWAN SEMICONDUCTOR MANUFACTURING CO LTD"},"9":
{"cik_str":1067983,"ticker":"BRK-A","title":"BERKSHIRE HATHAWAY INC"},"10":{"cik_str":1403161,"ticker":"V","title":"VISA INC."},"11":
{"cik_str":200406,"ticker":"JNJ","title":"JOHNSON & JOHNSON"},"12":{"cik_str":104169,"ticker":"WMT","title":"Walmart Inc."},"13":{"cik_str":19617,"ticker":"JPM","title":"JPMORGAN
CHASE & CO"},"14":{"cik_str":884394,"ticker":"SPY","title":"SPDR S&P 500 ETF TRUST"},"15":{"cik_str":1141391,"ticker":"MA","title":"Mastercard Inc"},"16":
{"cik_str":731766,"ticker":"UNH","title":"UNITEDHEALTH GROUP INC"},"17":{"cik_str":80424,"ticker":"PG","title":"PROCTER & GAMBLE Co"},"18":
{"cik_str":1045810,"ticker":"NVDA","title":"NVIDIA CORP"},"19":{"cik_str":824046,"ticker":"LVMUY","title":"LVMH MOET HENNESSY LOUIS VUITTON"},"20":
{"cik_str":1744489,"ticker":"DIS","title":"Walt Disney Co"},"21":{"cik_str":354950,"ticker":"HD","title":"HOME DEPOT, INC."},"22":{"cik_str":70858,"ticker":"BAC","title":"BANK OF
AMERICA CORP /DE/"},"23":{"cik_str":1633917,"ticker":"PYPL","title":"PayPal Holdings, Inc."},"24":{"cik_str":1445468,"ticker":"IDCBY","title":"Industrial & Commercial Bank of
China"},"25":{"cik_str":732712,"ticker":"VZ","title":"VERIZON COMMUNICATIONS INC"},"26":{"cik_str":1166691,"ticker":"CMCSA","title":"COMCAST CORP"},"27":
{"cik_str":320187,"ticker":"NKE","title":"NIKE, Inc."},"28":{"cik_str":796343,"ticker":"ADBE","title":"ADOBE INC."},"29":{"cik_str":1065280,"ticker":"NFLX","title":"NETFLIX
INC"},"30":{"cik_str":1737806,"ticker":"PDD","title":"Pinduoduo Inc."},"31":{"cik_str":21344,"ticker":"KO","title":"COCA COLA CO"},"32":
{"cik_str":1094517,"ticker":"TM","title":"TOYOTA MOTOR CORP/"},"33":{"cik_str":937966,"ticker":"ASML","title":"ASML HOLDING NV"},"34":
{"cik_str":1114448,"ticker":"NVS","title":"NOVARTIS AG"},"35":{"cik_str":50863,"ticker":"INTC","title":"INTEL CORP"},"36":{"cik_str":310158,"ticker":"MRK","title":"Merck & Co.,
Inc."},"37":{"cik_str":78003,"ticker":"PFE","title":"PFIZER INC"},"38":{"cik_str":732717,"ticker":"T","title":"AT&T INC."},"39":
{"cik_str":1108524,"ticker":"CRM","title":"SALESFORCE.COM, INC."},"40":{"cik_str":97745,"ticker":"TMO","title":"THERMO FISHER SCIENTIFIC INC."},"41":
{"cik_str":1341439,"ticker":"ORCL","title":"ORACLE CORP"},"42":{"cik_str":34088,"ticker":"XOM","title":"EXXON MOBIL CORP"},"43":{"cik_str":1800,"ticker":"ABT","title":"ABBOTT
LABORATORIES"},"44":{"cik_str":77476,"ticker":"PEP","title":"PEPSICO INC"},"45":{"cik_str":1445474,"ticker":"CICHY","title":"China Construction Bank Corp"},"46":
{"cik_str":1341439,"ticker":"ORCL","title":"ORACLE CORP"},"47":{"cik_str":858877,"ticker":"CSCO","title":"CISCO SYSTEMS, INC."},"48":
{"cik_str":1551152,"ticker":"ABBV","title":"AbbVie Inc."},"49":{"cik_str":1495688,"ticker":"CIHKY","title":"China Merchants Bank Co., Ltd. /ADR/"},"50":
```


2 Access Elements of Strings

strings.ipynb

Code | Python 3

Strings

[1]:
```
name = 'Walmart'
```

[2]:
```
print(name[0])          # First character (position 0).
print(name[1])
print(name[-1])         # Last character,
print(name[1:5])        # Positions 1,2,3,4 (5 is not included).
```

```
W
a
t
alma
```

[3]:
```
import requests

symbols = requests.get('https://www.sec.gov/files/company_tickers.json').text
```

[4]:
```
i = symbols.find('Facebook')
i
```

[4]: 362

[5]:
```
symbols[:i]
```

[5]:
```
'{"0":{"cik_str":320193,"ticker":"AAPL","title":"Apple Inc."},"1":{"cik_str":789019,"ticker":"MSFT","title":"MICROSOFT CORP"},"2":{"cik_str":1018724,"ticker":"AMZN","title":"AMAZON COM INC"},"3":{"c
ik_str":1652044,"ticker":"GOOG","title":"Alphabet Inc."},"4":{"cik_str":1318605,"ticker":"TSLA","title":"Tesla, Inc."},"5":{"cik_str":1326801,"ticker":"FB","title":"'
```

[6]:
```
first = symbols[:i].rfind('{')
first
```

[6]: 320

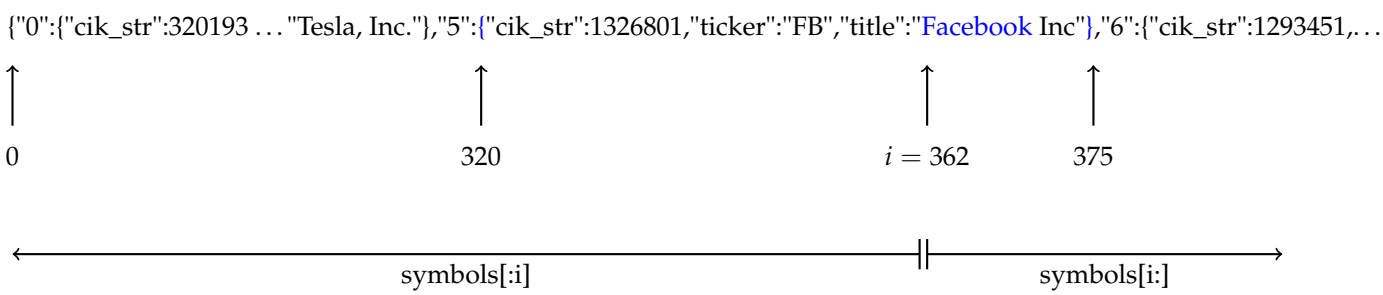
[7]:
```
last = i + symbols[i:].find('}')
last
```

[7]: 375

[8]:
```
symbols[first:last+1]
```

[8]:
```
'{"cik_str":1326801,"ticker":"FB","title":"Facebook Inc"}'
```

3 Finding Facebook in "symbols"



3.6 Defining Functions

This function is built into Python:

```
sum(2,3)              # 5
```

We can also define our own functions:

```
def add_this(x,y):    # This function takes 2 objects as input and refers to them internally as x and y.
    return x + y      # The code inside the function must be indented.
```

And now run the function like this:

```
add_this(2,3)         # 5     (when we run this, Python sets x to 2 and y to 3 in the function above).
add_this('a','b')     # 'ab'  (now Python sets x to 'a' and y to 'b' in the function above).
```

Let's construct a function that finds firms in the SEC ticker symbol file. Get the file from the SEC (page 27):

```
import requests
symbols = requests.get('https://www.sec.gov/files/company_tickers.json').text
```

For example, find the entry for Micosoft (see page 29):

```
i = symbols.find('MICROSOFT')             # Position of "MICROSOFT".
first = symbols[:i].rfind('{')            # Position of last  "{" before "MICROSOFT".
last  = i + symbols[i:].find('}')         # Position of first "}" after  "MICROSOFT".

symbols[first:last+1]                     # {"cik_str":789019,"ticker":"MSFT","title":"MICROSOFT CORP"}
```

Note how some firms in *symbols* only have the first letter capitalized and other firms are spelled in all caps. That means to find aa firm, we have to know its capitalization. To avoid this issue, let's convert all uppercase characters into lowercaser characters using the *.lower()* method:

```
symbols.lower()              # Returns a copy of "symbols" with all characters in lower case.
```

And now we can find Microsoft like this:

```
i = symbols.lower().find('microsoft')  # Transform characters to lower case, then search for 'microsoft'.
```

Let's put the entire search algorithm into a function:

```
def find_company(name):                # "name" should be some string.
    i = symbols.lower().find(name)     # find index (i) of first character in name
    last  = i + symbols[i:].find('}')  # Find the closing bracket.
    first = symbols[:i].rfind('{')     # Find the opening bracket.
    return symbols[first:last+1]       # Return substring that starts at { and ends at }.
```

Call the function:

```
find_company('microsoft')     # '{"cik_str":789019, "ticker":"MSFT", "title":"MICROSOFT CORP"}'
```

Note that this function can search by any stubstring inside the '{...}', so we can also find companies by CIK or ticker symbol:

```
find_company('wmt')           # '{"cik_str":104169,  "ticker":"WMT",  "title":"Walmart Inc."}'
find_company('1318605')       # '{"cik_str":1318605, "ticker":"TSLA", "title":"Tesla, Inc."}'
```

Nice!

1 SEC: company names and ticker symbols

sec.gov/files/company_tickers.json


```
{"0":{"cik_str":320193,"ticker":"AAPL","title":"Apple Inc."},"1":{"cik_str":789019,"ticker":"MSFT","title":"MICROSOFT CORP"},"2":
{"cik_str":1018724,"ticker":"AMZN","title":"AMAZON COM INC"},"3":{"cik_str":1652044,"ticker":"GOOG","title":"Alphabet Inc."},"4":
{"cik_str":1326801,"ticker":"FB","title":"Facebook Inc"},"5":{"cik_str":1293451,"ticker":"TCEHY","title":"Tencent Holdings Ltd"},"6":
{"cik_str":1318605,"ticker":"TSLA","title":"Tesla, Inc."},"7":{"cik_str":1577552,"ticker":"BABA","title":"Alibaba Group Holding Ltd"},"8":{"cik_str":1067983,"ticker":"BRK-
A","title":"BERKSHIRE HATHAWAY INC"},"9":{"cik_str":1046179,"ticker":"TSM","title":"TAIWAN SEMICONDUCTOR MANUFACTURING CO LTD"},"10":
{"cik_str":1403161,"ticker":"V","title":"VISA INC."},"11":{"cik_str":104169,"ticker":"WMT","title":"Walmart Inc."},"12":{"cik_str":200406,"ticker":"JNJ","title":"JOHNSON &
JOHNSON"},"13":{"cik_str":19617,"ticker":"JPM","title":"JPMORGAN CHASE & CO"},"14":{"cik_str":1141391,"ticker":"MA","title":"Mastercard Inc"},"15":
{"cik_str":80424,"ticker":"PG","title":"PROCTER & GAMBLE Co"},"16":{"cik_str":884394,"ticker":"SPY","title":"SPDR S&P 500 ETF TRUST"},"17":
{"cik_str":731766,"ticker":"UNH","title":"UNITEDHEALTH GROUP INC"},"18":{"cik_str":1045810,"ticker":"NVDA","title":"NVIDIA CORP"},"19":
{"cik_str":1744489,"ticker":"DIS","title":"Walt Disney Co"},"20":{"cik_str":824046,"ticker":"LVMUY","title":"LVMH MOET HENNESSY LOUIS VUITTON"},"21":
{"cik_str":354950,"ticker":"HD","title":"HOME DEPOT, INC."},"22":{"cik_str":1633917,"ticker":"PYPL","title":"PayPal Holdings, Inc."},"23":
{"cik_str":1445468,"ticker":"IDCBY","title":"Industrial & Commercial Bank of China"},"24":{"cik_str":70858,"ticker":"BAC","title":"BANK OF AMERICA CORP /DE/"},"25":
{"cik_str":732712,"ticker":"VZ","title":"VERIZON COMMUNICATIONS INC"},"26":{"cik_str":796343,"ticker":"ADBE","title":"ADOBE INC."},"27":
{"cik_str":1166691,"ticker":"CMCSA","title":"COMCAST CORP"},"28":{"cik_str":1065280,"ticker":"NFLX","title":"NETFLIX INC"},"29":
{"cik_str":21344,"ticker":"KO","title":"COCA COLA CO"},"30":{"cik_str":320187,"ticker":"NKE","title":"NIKE, Inc."},"31":{"cik_str":1114448,"ticker":"NVS","title":"NOVARTIS
AG"},"32":{"cik_str":1094517,"ticker":"TM","title":"TOYOTA MOTOR CORP/"},"33":{"cik_str":732717,"ticker":"T","title":"AT&T INC."},"34":
{"cik_str":937966,"ticker":"ASML","title":"ASML HOLDING NV"},"35":{"cik_str":310158,"ticker":"MRK","title":"Merck & Co., Inc."},"36":
{"cik_str":78003,"ticker":"PFE","title":"PFIZER INC"},"37":{"cik_str":1737806,"ticker":"PDD","title":"Pinduoduo Inc."},"38":{"cik_str":50863,"ticker":"INTC","title":"INTEL
CORP"},"39":{"cik_str":1108524,"ticker":"CRM","title":"SALESFORCE.COM, INC."},"40":{"cik_str":77476,"ticker":"PEP","title":"PEPSICO INC"},"41":
{"cik_str":1341439,"ticker":"ORCL","title":"ORACLE CORP"},"42":{"cik_str":1800,"ticker":"ABT","title":"ABBOTT LABORATORIES"},"43":
{"cik_str":1341439,"ticker":"ORCL","title":"ORACLE CORP"},"44":{"cik_str":1445474,"ticker":"CICHY","title":"China Construction Bank Corp"},"45":
```

3.7 Function Scope

Main advantages of using functions in our code:

1. functions allow us to perform then same calculation multiple times (with possibly different input),
2. we can "hide" code inside functions and do not need to worry that our variables inside the functions conflict with variables outside the function.

For example, suppose we have a very long piece of code that looks like this:

```
fixed_fee    = 5                                    # $5 fixed fee for trading.
variable_fee = 0.02                                 # 2% variable fee.

# Hundreds of lines of code here

def calculate_proceeds(shares,price):
    fixed_fee = 2
    return shares * price * (1-variable_fee) - fixed_fee

proceeds = calculate_proceeds(10,20)               # Returns 194 (=20*10*(1-0.02)-2)
print(fixed_fee)                                   # Output: 5
```

Here we define the variable "fixed_fee" twice: first outside of the function (we also call this a global variable), and then inside the function (we also call this a local variable). When our code gets very long, it is possible to accidentally overwrite variable names. Defining variables inside functions protects us from potential bugs since they do not conflict with variables outside the function: the local variable "fixed_fee" ($=$ 2) exists only inside *calculate_proceeds()* and this assignment does not affect the "fixed_fee" outside the function ($=$ 5).
Note how *calculate_proceeds()* uses the global"variable_fee" ($=$ 0.02). That means Python determines the value of variables inside functions in two steps:

1. Was the variable defined locally? If yes, use local variable.
2. If no, use most recent global variable definition.

We can use the *id()* function (page 23) to see which objects our variable refer to throughout our code (see Figure Figure 1):

- lines 1 - 9: define a function "do_this"
- line 13: create a string 'A' and assign it to the global variable x
- line 15: print the value and the id of x. Here we see that the computer that runs this program uses 140595754587632 as an internal reference for the object 'A'.
- line 17: pass the reference to 'A' to the function *do_this*
- line 1: assign the reference y to the object 'A'. Now y is a local variable that only lives inside *do_this*.
- line 3: print the id of y. Here we see that the local y and the global x have the same id, which means they refer to the same object 'A'
- line 5: generate a new object 'B' and assign it the reference x. Now x is a local variable that only lives inside *do_this* and this x is not the same as the global x we created in line 13.
- line 9: return a reference to 'A' and assign the object to z in line 17.
- line 19: print the id of z. Here we see that z refers to the original object 'A'
- line 20: print the id of the global x. Here we see that the local assignment in line 5 did not affect the global x.

1 Function scope

functions.ipynb

Code Python 3

Functions

[24]:

```python
def do_this(y):

    print('2: y =', y, id(y))

    x = 'B'

    print('3: x =', x, id(x))

    return y



x = 'A'

print('1: x =', x, id(x))

z = do_this(x)

print('4: z =', z, id(z))
print('5: x =', x, id(x))
```

```
1: x = A 140595754587632
2: y = A 140595754587632
3: x = B 140595486133040
4: z = A 140595754587632
5: x = A 140595754587632
```

3.8 Lists

We can collect multiple data points in a list:

```
tickers = ['AAPL','MSFT','AMZN','FB','NFLX']
```

We can access elements of this list just like we access characters of a string:

```
tickers[1]      # 'MSFT'                (list index starts with 0).
tickers[1:4]    # ['MSFT','AMZN','FB']  (elements 1 to 3; 4 not included).
```

Table 1 shows some more examples.
We can modify elements of the list:

```
tickers[0]   = 'NKE'          # Now tickers is ['NKE','MSFT','AMZN','FB','NFLX']
tickers[1:3] = ['WM','JNJ']   # Now tickers is ['NKE','WM','JNJ','FB','NFLX']
```

Combine (concaten) two lists (similar to string concatenations):

```
tickers = tickers + ['V']    #  create a new list ['NKE','WM','JNJ','FB','NFLX','V']Thi
tickers + 'C'                #  This creates a type error (can only concatenate lists).
```

There are various methods available for lists. For example we can find the index of 'JNJ':

```
tickers.index('JNJ')        #  2
```

Most list methods modify the list in place (that means the list changes when we apply the method):

```
tickers.append('JPM')      #  Now tickers is ['NKE','WM','JNJ','FB','NFLX','V','JPM']
tickers.sort()             #  Now tickers is ['FB','JNJ','JPM','NFLX','NKE','V','WM']
```

Here is another example of the difference between modifying a list "in place" and creating a new list:

```
a    = [1,2,3,4]
b    = a            #  Now b refers to the list a
a.append(5)         #  Modify the orignal list => now a and b are both [1,2,3,4,5]
a[0] = 9            #  Modify the orignal list => now a and b are both [9,2,3,4,5]
a    = a + [6]      #  Create a new list [9,2,3,4,5,6] and call this list "a" (no effect on b)
```

In the last line we reassign the name "a" to a new list object [9,2,3,4,5,6] ("a + [6]" creates a new list without modifying the old list). Since "b" refers to the old list, "b" does not change.
We can combine list function and the range function to create a sequence like this:

```
list(range(10))          #  [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]    (10 is not included)
list(range(10))[::2]     #  [0, 2, 4, 6, 8]                   ([::2] selects every 2nd element)
list(range(2,6))         #  [2, 3, 4, 5]
```

Convert a string into a list:

```
x = 'hey_how_are_you'
y = x.split('_')         #  ['hey', 'how', 'are', 'you']   (Split the string at each occurrene of '_')
```

Note that *.split* is a string method (we attache it to the string 'hey_how_are_you'). We can use another string method to convert the list back to a string:

```
' '.join(y)              #  'hey how are you'              (Use a single white space to join the list)
```

1 Select elements of a list

prices		101, 104, 108, 106, 104, 105, 110, 109								
positions		0	1	2	3	4	5	6	7	
		-8	-7	-6	-5	-4	-3	-2	-1	count from back
selection	0	101, 104, 108, 106, 104, 105, 110, 109								
	1	101, 104, 108, 106, 104, 105, 110, 109								
	-1	101, 104, 108, 106, 104, 105, 110, 109								count from back
	1:4	101, 104, 108, 106, 104, 105, 110, 109								4 not included
	:3	101, 104, 108, 106, 104, 105, 110, 109								start from 0
	3:	101, 104, 108, 106, 104, 105, 110, 109								
	-3:	101, 104, 108, 106, 104, 105, 110, 109								until the end
	1:-2	101, 104, 108, 106, 104, 105, 110, 109								
	::2	101, 104, 108, 106, 104, 105, 110, 109								every 2nd element
	1:6:2	101, 104, 108, 106, 104, 105, 110, 109								
	::-1	109, 110, 105, 104, 106, 108, 104, 101								reverse list

2 Modifying a list

builtin_functions.ipynb

Code Python 3

Modifying Lists In-Place

[1]:
```
a = [1,2,3,4]              #  Create a list [1,2,3,4] and call it "a".
b = a                      #  Now "b" refers to the same list as "a".
print('a:', a,' b:',b)
```

```
a: [1, 2, 3, 4]  b: [1, 2, 3, 4]
```

[2]:
```
a.append(5)                #  Now a and b are both [1,2,3,4,5].
print('a:', a,' b:',b)
```

```
a: [1, 2, 3, 4, 5]  b: [1, 2, 3, 4, 5]
```

[3]:
```
a[0] = 9                   #  Now a and b are both [9,2,3,4,5].
print('a:', a,' b:',b)
```

```
a: [9, 2, 3, 4, 5]  b: [9, 2, 3, 4, 5]
```

[4]:
```
a + [6,7]                  # Create a new list [9,2,3,4,5,6,7] (this does not change a or b).
```

[4]:
```
[9, 2, 3, 4, 5, 6, 7]
```

[5]:
```
a = a + [6,7]              # Create a new list and call this list "a" (no effect on b).
print('a:', a,' b:',b)
```

```
a: [9, 2, 3, 4, 5, 6, 7]  b: [9, 2, 3, 4, 5]
```

3.9 Dictionaries

Check the SEC *.json* file in Figure 1. JSON is a file format that was origianlly developed to exchange data on the Web. Note how the data in Figure 1 are structured with the symbols "{", "}" and ":".
In Python we can implement this data format like this:

```
apple = {'cik_str':320193, 'ticker':'AAPL', 'title':'Apple Inc.'}
```

This data structure is called a dictionary:

```
type(apple)          #  dict
```

A dictionary consists of pairs of keys ('cik_str', 'ticker', 'title') and values (320193, 'AAPL', 'Apple Inc.'):

```
apple.keys()             # Get all keys of dictionary   (object type: dict_keys).
apple.values()           # Get all values of dictionary (object type: dict_values).
```

We can use the list function to turn the keys and values into lists:

```
list(apple.keys())       #  ['cik_str', 'ticker', 'title']
list(apple.values())     #  [320193, 'AAPL', 'Apple Inc.]
```

We can access the values of the dictionary like this:

```
apple['ticker']          #  'AAPL'
apple['cik_str']         #  '320193'
```

(the term "dictionary" indicates that we can look up its entries similar to searching a traditional dictionary).
But we cannot access a dictionay by index (like a list), so for example, *apple[0]* will generate an error message.

The *requests* library has a *json()* method to turn *json* strings into dictionaries:

```
import requests
symbols = requests.get('https://www.sec.gov/files/company_tickers.json').json()
type(symbols)              #   dict
```

Now "symbols" is a dictionary. List of keys of "symbols":

```
list(symbols.keys())       #  ['0', '1', '2', ...]
```

For example, the first key of "symbols" is '0'. So we can get the first value like this:

```
symbols['0']                #  {'cik_str':320193, 'ticker':'AAPL', 'title':'Apple Inc.'}
```

Note how this value is also a dictionary (so "symbols" is a dictionary of dictionaries, see Figure 2). Apply the *['ticker']* selection to first value of "symbols":

```
symbols['0']['ticker']      #  'AAPL' (First select key '0', and then select key 'ticker')
```

Or get the title of the 2nd entry:

```
symbols['1']['title']       #  'MICROSOFT CORP' (First select key '1', and then select key 'title')
```

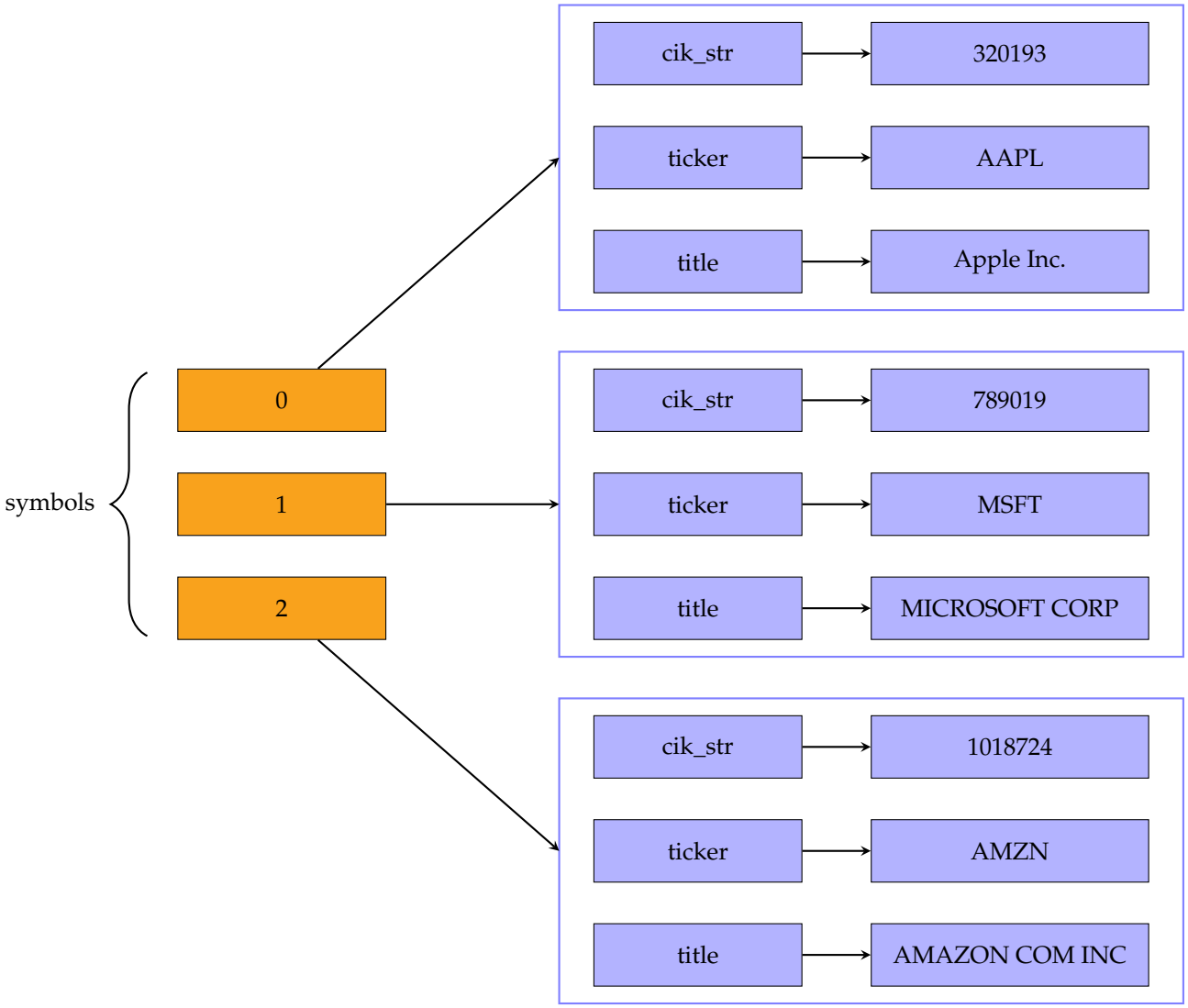
Note that we can access dictionary entries only by its keys and not by position (like a list). We can only use "symbols['1']", because '1' is a key of this dictionary. But for example "symbols[1][2]" does not work (we need to use "symbols['1']['title']" instead).

1 SEC: company names and ticker symbols

sec.gov/files/company_tickers.json


```
{"0":{"cik_str":320193,"ticker":"AAPL","title":"Apple Inc."},"1":{"cik_str":789019,"ticker":"MSFT","title":"MICROSOFT CORP"},"2":
{"cik_str":1018724,"ticker":"AMZN","title":"AMAZON COM INC"},"3":{"cik_str":1652044,"ticker":"GOOG","title":"Alphabet Inc."},"4":
{"cik_str":1326801,"ticker":"FB","title":"Facebook Inc"},"5":{"cik_str":1293451,"ticker":"TCEHY","title":"Tencent Holdings Ltd"},"6":
{"cik_str":1318605,"ticker":"TSLA","title":"Tesla, Inc."},"7":{"cik_str":1577552,"ticker":"BABA","title":"Alibaba Group Holding Ltd"},"8":{"cik_str":1067983,"ticker":"BRK-
A","title":"BERKSHIRE HATHAWAY INC"},"9":{"cik_str":1046179,"ticker":"TSM","title":"TAIWAN SEMICONDUCTOR MANUFACTURING CO LTD"},"10":
{"cik_str":1403161,"ticker":"V","title":"VISA INC."},"11":{"cik_str":104169,"ticker":"WMT","title":"Walmart Inc."},"12":{"cik_str":200406,"ticker":"JNJ","title":"JOHNSON &
JOHNSON"},"13":{"cik_str":19617,"ticker":"JPM","title":"JPMORGAN CHASE & CO"},"14":{"cik_str":1141391,"ticker":"MA","title":"Mastercard Inc"},"15":
{"cik_str":80424,"ticker":"PG","title":"PROCTER & GAMBLE Co"},"16":{"cik_str":884394,"ticker":"SPY","title":"SPDR S&P 500 ETF TRUST"},"17":
{"cik_str":731766,"ticker":"UNH","title":"UNITEDHEALTH GROUP INC"},"18":{"cik_str":1045810,"ticker":"NVDA","title":"NVIDIA CORP"},"19":
{"cik_str":1744489,"ticker":"DIS","title":"Walt Disney Co"},"20":{"cik_str":824046,"ticker":"LVMUY","title":"LVMH MOET HENNESSY LOUIS VUITTON"},"21":
{"cik_str":354950,"ticker":"HD","title":"HOME DEPOT, INC."},"22":{"cik_str":1633917,"ticker":"PYPL","title":"PayPal Holdings, Inc."},"23":
{"cik_str":1445468,"ticker":"IDCBY","title":"Industrial & Commercial Bank of China"},"24":{"cik_str":70858,"ticker":"BAC","title":"BANK OF AMERICA CORP /DE/"},"25":
{"cik_str":732712,"ticker":"VZ","title":"VERIZON COMMUNICATIONS INC"},"26":{"cik_str":796343,"ticker":"ADBE","title":"ADOBE INC."},"27":
{"cik_str":1166691,"ticker":"CMCSA","title":"COMCAST CORP"},"28":{"cik_str":1065280,"ticker":"NFLX","title":"NETFLIX INC"},"29":
{"cik_str":21344,"ticker":"KO","title":"COCA COLA CO"},"30":{"cik_str":320187,"ticker":"NKE","title":"NIKE, Inc."},"31":{"cik_str":1114448,"ticker":"NVS","title":"NOVARTIS
AG"},"32":{"cik_str":1094517,"ticker":"TM","title":"TOYOTA MOTOR CORP/"},"33":{"cik_str":732717,"ticker":"T","title":"AT&T INC."},"34":
{"cik_str":937966,"ticker":"ASML","title":"ASML HOLDING NV"},"35":{"cik_str":310158,"ticker":"MRK","title":"Merck & Co., Inc."},"36":
{"cik_str":78003,"ticker":"PFE","title":"PFIZER INC"},"37":{"cik_str":1737806,"ticker":"PDD","title":"Pinduoduo Inc."},"38":{"cik_str":50863,"ticker":"INTC","title":"INTEL
CORP"},"39":{"cik_str":1108524,"ticker":"CRM","title":"SALESFORCE.COM, INC."},"40":{"cik_str":77476,"ticker":"PEP","title":"PEPSICO INC"},"41":
{"cik_str":1341439,"ticker":"ORCL","title":"ORACLE CORP"},"42":{"cik_str":1800,"ticker":"ABT","title":"ABBOTT LABORATORIES"},"43":
{"cik_str":1341439,"ticker":"ORCL","title":"ORACLE CORP"},"44":{"cik_str":1445474,"ticker":"CICHY","title":"China Construction Bank Corp"},"45":
```


2 "symbols" is a dictionary of dictionaries



3.10 Conditional Statements

Check for inequality:

```
2 > 1                   #  True
```

We can assign a comparison to a variable:

```
x = 5 < 3               #  False
```

What type of variable is x?

```
type(x)                 #  bool
```

Boolean variables can have two values: *True* or *False*.
Check for equality:

```
x = 5                   #  Now x refers to 5.
x == 4                  #  False
x == 5                  #  True
```

Note how we use "==" to test if x equals 5 and we use "=" to set x to 5.
Execute code conditionally:

```
price = 60

if(price < 50):
    print('price is below 50!')
    print('buy!!')
else:                                       #  The else statement is optional
    print('price is above 50!')
    print('sell!!')
```

If we have only one line of code after the if statement we can also write it like this:

```
if(price < 50): print('buy!')
else:           print('sell!')
```

Multiple tests:

```
if  (price < 60): print('buy!')             # if price >= 60 go to next line
elif(price < 70): print('hold')             # if price >= 70 go to next line  (elif is short for "else if")
elif(price < 80): print('sell a little!')   # if price >= 80 go to next line
else:             print('sell!')
```

Assign variable conditonally:

```
trade = 'buy' if (price<50) else 'sell'
```

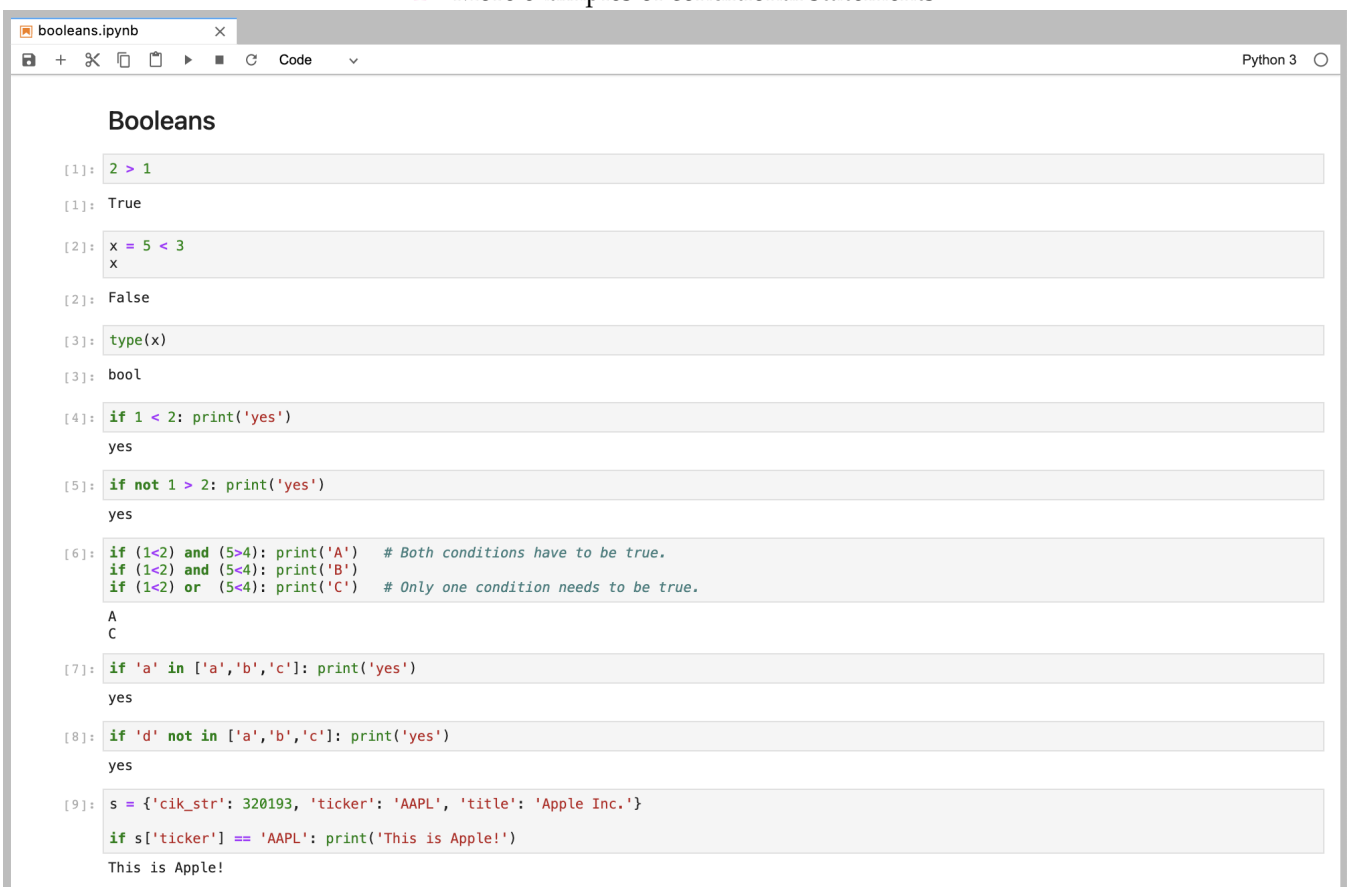
Zero evaluates as False:

```
x =  0
y = -3
if(x): print('got here: x')         #  This will not be printed because 0 counts as false
if(y): print('got here: y')         #  This will be printed
```

1 Python Comparisons

	Symbol
Equal	==
Not equal	!=
Greater than	>
Less than	<
Less or equal	<=
Greater or equal	>=

2 More examples of conditional statements



3.11 Loops

Consider the following print statements:

```
print(1)
print(2)
print(3)
```

This works, but what if we need to print the numbers 1 to 100?
If we need to repeat the same action multiple times, we can use a for-loop like this:

```
for i in range(1,4):    # Loop over i=1,2,3 (4 is not included).
    print(i)            # Repeat the indented code block.
```

This loop repeats the print statement three times, starting with $i = 1$ for the first round, then $i = 2$ and $i = 3$.
If we replace "range(1,4)" with "range(4)", the range starts at 0 (loop over i = 0, 1, 2, 3).
Let's use a loop to search through the SEC ticker symbols file:

```
import requests
symbols = requests.get('https://www.sec.gov/files/company_tickers.json').json()
```

Now we have a dictionay of dictionaries that looks like this:

'0' : {'cik_str':320193, 'ticker":'AAPL', 'title': 'Apple Inc.''},
'1' : {'cik_str':789019, 'ticker":'MSFT', 'title': 'MICROSOFT CORP'},
'2' : {'cik_str':1018724, 'ticker":'AMZN', 'title': 'AMAZON COM INC'},
...

(see page 37). Get the first value:

```
s = symbols['0']    # {"cik_str":320193,  "ticker":"AAPL", "title": "Apple Inc."}
s[ticker]           # "AAPL"
```

Note that the keys of "symbols" are strings, so we have to be careful when use integers to access "symbols":

```
i = 0                   # i is an integer.
symbols[i]              # THIS PRODUCES AN ERROR.
symbols[str(i)]         # This works (we transform 0 into the string '0').
```

How many securities:

```
len(symbols)            # 11228
```

(so the first value of "symbols" is at '0' and the last value is at '11227').
Loop over all entries of "symbols" and search for Oracle (ORCL):

```
for i in range(len(symbols)):     # Repeat code block for i = 0,1,2,...,11227 (11228 not included).
    s = symbols[str(i)]
    if s['ticker'] == 'ORCL':
        print(s['title'], s['ticker'], s['cik_str'])
```

Note how this search returns two identical entries, which means the file contains at least one duplicate.
We can also directly loop over the values of "symbols":

```
for s in symbols.values():  # "s" refers the the 1st value of symbols, then the 2nd value, ...
    if s['ticker'] == 'ORCL':
        print(s['title'], s['ticker'], s['cik_str'])
```

This loop produces the same output as the loop in the previous cell.

1 Loops

loops.ipynb

Code Python 3

Loops

[1]:
```
print(1)
print(2)
print(3)
print(4)
```

```
1
2
3
4
```

[2]:
```
for i in range(1,5):
    print(i)
```

```
1
2
3
4
```

[3]:
```
import requests
symbols = requests.get('https://www.sec.gov/files/company_tickers.json').json()
```

[4]:
```
len(symbols)
```

[4]:
```
11228
```

[5]:
```
for i in range(len(symbols)):    # Repeat code block for i = 0,1,2,...,11228-1 (11228 not included)
    s = symbols[str(i)]
    if s['ticker'] == 'ORCL':
        print(s['title'], s['ticker'], s['cik_str'])
```

```
ORACLE CORP ORCL 1341439
ORACLE CORP ORCL 1341439
```

We can loop over objects:

[6]:
```
x = ['a','b','c']
for i in x:            # Loop over a list.
    print(i)
```

```
a
b
c
```

[7]:
```
x = 'what'
for j in x:            # Loop over a string.
    print(j)
```

```
w
h
a
t
```

[8]:
```
for s in symbols.values():  # "s" refers the the 1st value of symbols, then the 2nd value, ...
    if s['ticker'] == 'ORCL':
        print(s['title'], s['ticker'], s['cik_str'])
```

```
ORACLE CORP ORCL 1341439
ORACLE CORP ORCL 1341439
```

3.12 Constructing Sequences

Consider the following list:

```
x = [1, 2, 3, 4, 5]
```

Suppose we would like square each element of x. We can put a loop directly into a list:

```
[i**2 for i in x]              #  [1, 4, 9, 16, 25]     "Take each element (i) of x and square it".
```

Alternatively we can also use "range(1,6)" to create a sequence of numbers from 1 to 5 (6 not included):

```
[j**2 for j in range(1,6)]     #  [1, 4, 9, 16, 25]     "Take the numbers 1,2,..,5 and square them".
```

This way of creating lists (loops inside lists) is also referred to as list comprehension.
Let's apply this to the SEC ticker symbols file:

```
import requests
symbols = requests.get('https://www.sec.gov/files/company_tickers.json').json()
```

Now we have a dictionay of dictionaries with keys 0,1,2,... and values given by:

{"cik_str":320193, "ticker":"AAPL", "title": "Apple Inc."},
{"cik_str":789019, "ticker":"MSFT", "title": "MICROSOFT CORP"},
{"cik_str":1018724, "ticker":"AMZN", "title": "AMAZON COM INC"},
...

We can access the collection of all these values with "symbols.values()". Create a list of all tickers:

```
[s['ticker'] for s in symbols.values()]    # ['AAPL', 'MSFT', 'AMZN', 'GOOG', ...]
```

Here we loop over "symbols.values()" and apply the ['ticker'] selection to each value.
Create a list of all titles:

```
[s['title'] for s in symbols.values()]     # ['Apple Inc.','MICROSOFT CORP','AMAZON COM INC', ...]
```

We can also combine the loop with a condition:

```
[s['title'] for s in symbols.values() if s['ticker'] == 'MSFT']  # ['MICROSOFT CORP']
```

We can also use the comprehension method to create a dictionary like this:

{ *key* : *value* for *variable* in ... }

For example, create a dictionary with the tickers as keys and the CIKs as values:

```
ticker_cik = {s['ticker']:s['cik_str'] for s in symbols.values()}
```

Here we loop over the values of symbols and then combine the ticker and the CIK as the key-value pairs for our new dictionary. Now we have a dictionay "ticker_cik" that looks like this:

{"AAPL":320193, "MSFT":789019, "AMZN":1018724, ...},

And now we can find CIKs like this:

```
ticker_cik['AAPL']   #  320193
```

1 Creating lists and dictionaries

comprehensions.ipynb

Code Python 3

Comprehensions

[1]:
```
import requests
symbols = requests.get('https://www.sec.gov/files/company_tickers.json').json()
```

[2]:
```
tickers = [s['ticker'] for s in symbols.values()]

tickers[:10]  # Show first 10.
```

[2]:
```
['AAPL', 'MSFT', 'AMZN', 'GOOG', 'TSLA', 'FB', 'TCEHY', 'BABA', 'TSM', 'BRK-A']
```

[3]:
```
names = [s['title'] for s in symbols.values()]

names[:10]  # Show first 10.
```

[3]:
```
['Apple Inc.',
 'MICROSOFT CORP',
 'AMAZON COM INC',
 'Alphabet Inc.',
 'Tesla, Inc.',
 'Facebook Inc',
 'Tencent Holdings Ltd',
 'Alibaba Group Holding Ltd',
 'TAIWAN SEMICONDUCTOR MANUFACTURING CO LTD',
 'BERKSHIRE HATHAWAY INC']
```

Titles of all firms with ticker symbol 'MSFT':

[4]:
```
[s['title'] for s in symbols.values() if s['ticker'] == 'MSFT']
```

[4]:
```
['MICROSOFT CORP']
```

Titles of all firms where the first 3 characters are 'abc':

[5]:
```
[s['title'] for s in symbols.values() if s['title'][:3].lower() == 'abc']
```

[5]:
```
['AbCellera Biologics Inc.',
 'Abcam plc',
 'ABCO Energy, Inc.',
 'Abcam plc',
 'ABCO Energy, Inc.']
```

[6]:
```
ticker_cik = {s['ticker']:s['cik_str'] for s in symbols.values()}
```

[7]:
```
ticker_cik['AAPL']
```

[7]:
```
320193
```

[8]:
```
ticker_cik
```

[8]:
```
{'AAPL': 320193,
 'MSFT': 789019,
 'AMZN': 1018724,
 'GOOG': 1652044,
 'TSLA': 1318605,
 'FB': 1326801,
 'TCEHY': 1293451,
 'BABA': 1577552,
 'TSM': 1046179,
 'BRK-A': 1067983,
 'V': 1403161,
 'JNJ': 200406,
 'WMT': 104169,
 'JPM': 19617,
 'SPY': 884394,
```

3.13 Pandas DataFrames

Table 1 shows the stock prices of Apple and Facebook in the week of March 23, 2020.
In finance we often organize data in tables. The best Python data structure for tables is a pandas dataFrame.
Start by importing the pandas library:

```
import pandas as pd     # Import the pandas library and abbreviate it as "pd"
```

Now we can create a table of Apple and Facebook stock prices for example like this:

```
prices = pd.DataFrame(index=['Mon','Tue','Wed','Thu','Fri'])      #  Generate a table called "prices".

prices['Apple']    = [224.37, 246.88, 245.52, 258.44, 247.74]     #  Add column Apple.
prices['Facebook'] = [148.10, 160.98, 156.21, 163.34, 156.79]     #  Add column Facebook.
```

Figure 2 show the output of this cell.
There are many ways to create dataFrames. For example we can pass a dictionary to *pd.DataFrame()*:

```
row_index = ['a','b','c']                    # This is a list
data      = {'A':[1,2,3], 'B':[4,5,6]}       # This is a dictionary.

w = pd.DataFrame()                           # w is an empty table (no rows, no columns)
x = pd.DataFrame(index=row_index)            # x is an empty table with row index a,b,c.
y = pd.DataFrame(data)                       # y is a table with columns A,B and index 0,1,2 (default index)
z = pd.DataFrame(data, index=row_index)      # z is a table with columns A,B and index a,b,c.
```

We can also use pandas to read files from the web.
For example, read the SEC .json file:

```
pd.read_json('https://www.sec.gov/files/company_tickers.json')
```

Now we have a table with row indexes cik_str, ticker, and title and columns 0,1,2,. . . (Figure 2).
Let's use the pandas *transpose()* method to switch rows and columns:

```
symbols = pd.read_json('https://www.sec.gov/files/company_tickers.json').transpose()
```

And now we can access specific columns like this:

```
symbols['ticker']       # Column 'ticker'.
symbols.ticker          # Same as symbols['ticker'] (if there is no space in the column label).
```

We can filter the table by putting conditions into brackets:

```
symbols[symbols.ticker == 'WMT']                # Get all rows where ticker == 'WMT'.
symbols[symbols.title  == 'Walmart Inc.']       # Get all rows where title  == 'Walmart Inc.'.
```

Google (Alphabet Inc.) has two shares classes with ticker symbols 'GOOG' and 'GOOGL':

```
symbols[symbols.title == 'Alphabet Inc.']       # Returns a table with two entries.
```

We can can get the same result by searching for multiple ticker symbols like this:

```
symbols[(symbols.ticker=='GOOG') | (symbols.ticker=='GOOGL')]         # | means "or".
```

Here we must put the each condition into "()" and we must use the "|" symbol (and not the Python "or" keyword).
Use "&" for an "and" condition:

```
symbols[(symbols.title=='Alphabet Inc.') & (symbols.ticker=='GOOG')]  # & means "and"
```

1 Historical stock prices

		Apple	Facebook
2020-03-23	**Mon**	224.37	148.10
2020-03-24	**Tue**	246.88	160.98
2020-03-25	**Wed**	245.52	156.21
2020-03-26	**Thu**	258.44	163.34
2020-03-27	**Fri**	247.74	156.79

2 Creating a dataFrame

dataFrame.ipynb

Code Python 3

Pandas DataFrames

[1]:
```
import pandas as pd     # Import the pandas library and abbreviate it as "pd"
```

[2]:
```
prices = pd.DataFrame(index = ['Mon','Tue','Wed','Thu','Fri'] ) #  Generate table.
prices['Apple']    = [224.37, 246.88, 245.52, 258.44, 247.74]   #  Add column Apple.
prices['Facebook'] = [148.10, 160.98, 156.21, 163.34, 156.79]   #  Add column Facebook.
prices                                                          #  Show the result
```

[2]:

	Apple	Facebook
Mon	224.37	148.10
Tue	246.88	160.98
Wed	245.52	156.21
Thu	258.44	163.34
Fri	247.74	156.79

[3]:
```
type(prices)            #  What type of object is this?
```

[3]:
```
pandas.core.frame.DataFrame
```

Use pandas to read the SEC company_ticker file:

[4]:
```
pd.read_json('https://www.sec.gov/files/company_tickers.json')
```

[4]:

	0	1	2	3	4	5	6	7	8	9	...	11218	11219	11220
cik_str	320193	789019	1018724	1652044	1318605	1326801	1293451	1577552	1046179	1067983	...	1348952	1354591	1355096
ticker	AAPL	MSFT	AMZN	GOOG	TSLA	FB	TCEHY	BABA	TSM	BRK-A	...	ELU	GEUR	QRTEP
title	Apple Inc.	MICROSOFT CORP	AMAZON COM INC	Alphabet Inc.	Tesla, Inc.	Facebook Inc	Tencent Holdings Ltd	Alibaba Group Holding Ltd	TAIWAN SEMICONDUCTOR MANUFACTURING CO LTD	BERKSHIRE HATHAWAY INC	...	ENTERGY LOUISIANA, LLC	PetLife Pharmaceuticals, Inc.	Qurate Retail, Inc.

3 rows × 11228 columns

[5]:
```
symbols = pd.read_json('https://www.sec.gov/files/company_tickers.json').transpose()
symbols
```

[5]:

	cik_str	ticker	title
0	320193	AAPL	Apple Inc.
1	789019	MSFT	MICROSOFT CORP
2	1018724	AMZN	AMAZON COM INC
3	1652044	GOOG	Alphabet Inc.
4	1318605	TSLA	Tesla, Inc.
...	...	...	...
11223	1326160	DUKB	Duke Energy CORP
11224	1326205	IGCIW	India Globalization Capital, Inc.
11225	1333986	EQH-PA	Equitable Holdings, Inc.
11226	1335105	LIXTW	LIXTE BIOTECHNOLOGY HOLDINGS, INC.
11227	1338065	DCP-PB	DCP Midstream, LP

11228 rows × 3 columns