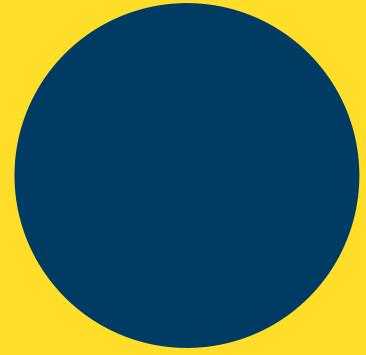




Data Structure

Day 4 - Python

Link Google Colab :
- [Python Basic](#)



Profile

Hello I'm Bayuzen Ahmad
I'm currently working as Data scientist

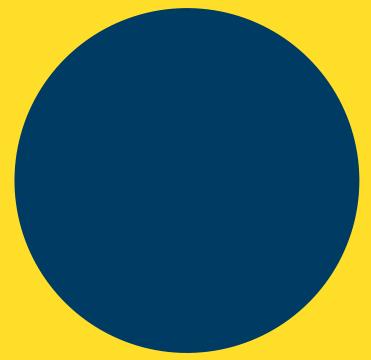
Linkedin : Bayuzen Ahmad
Let's go to connect with me 😊



Table of Contents

1	Introduction to Python & Tools	
2	Data Types : Numerical & Categorical	Day 2
3	Conditional Statement & Looping	
4	Data Structure	Day 4
5	Function	
6	Python Libraries	
7	Exploratory Data Analysis	
8	Data Visualization	Bacaan Tambahan (Optional)
9	Data Preprocessing	





Day 4



SEQUENCE DATA TYPES

SEQUENCE DATA TYPES



In this section we'll cover **sequence data types**, including lists, tuples, and ranges, which are capable of storing many values

TOPICS WE'LL COVER:

List Basics

List Operations

Modifying Lists

List Functions & Methods

Nested Lists

Copying Lists

Tuples

Ranges

GOALS FOR THIS SECTION:

- Learn to create lists and modify list elements
- Apply common list functions and methods
- Understand the different methods for copying lists
- Review the benefits of using tuples and ranges



LISTS

Lists are iterable data types capable of storing many individual elements

- List elements can be any data type, and can be added, changed, or removed at any time

```
random_list = ['snowboard', 10.54, None, True, -1]
```

Lists are created with square brackets []

List elements are separated by commas

```
item_list = ['Snowboard', 'Boots', 'Helmet', 'Goggles', 'Bindings']
```

} While lists can contain multiple data types,
they often contain **one data type**

```
empty_list = []
empty_list
[]
```

} You can create an empty list
and add elements later

```
list('Hello')
['H', 'e', 'l', 'l', 'o']
```

} You can create lists from
other iterable data types



MEMBERSHIP TESTS

Since lists are iterable data types, you can conduct **membership tests** on them

```
sizes_in_stock = ['XS', 'S', 'L', 'XL', 'XXL']  
'M' in sizes_in_stock
```

False

'M' is not an element in *sizes_in_stock*, so False is returned

```
if 'M' in sizes_in_stock:  
    print("I'll take the medium please.")  
elif 'S' in sizes_in_stock:  
    print("It'll be tight but small will do.")  
else:  
    print("I'll wait until you have medium.")
```

It'll be tight but small will do.

'M' is not an element in *sizes_in_stock*, but 'S' is, so the print function under elif is executed



LIST INDEXING

Elements in a list can be accessed via their **index**, or position within the list

- Remember that Python is 0-indexed, so the first element has an index of 0, not 1

```
item_list = ['Snowboard', 'Boots', 'Helmet', 'Goggles', 'Bindings']  
item_list[0]  
'Snowboard'
```

An index of 0 grabs the
first list element

```
item_list[3]  
'Goggles'
```

An index of 3 grabs the
fourth list element



LIST SLICING

Slice notation can also be used to access portions of lists

[start: stop: step size]

```
item_list = ['Snowboard', 'Boots', 'Helmet', 'Goggles', 'Bindings']  
item_list[:3]  
['Snowboard', 'Boots', 'Helmet']
```

list[3] grabs elements 0, 1, and 2

Remember that 'stop' is non-inclusive!

```
item_list[::-2]
```

```
['Snowboard', 'Helmet', 'Bindings']
```

list[::-2] grabs every other element in the list

```
item_list[2:4]
```

```
['Helmet', 'Goggles']
```

list[2:4] grabs the 3rd (index of 2) and 4th (index of 3) elements in the list



UNPACKING LISTS

Lists elements can be **unpacked** into individual variables

```
item_list = ['Snowboard', 'Boots', 'Helmet']
s, b, h = item_list
print(s, b, h)
```

Snowboard Boots Helmet

s, b, and h are now string variables



You need to specify the same number of variables as list elements or you will receive a **ValueError**



CHANGING LIST ELEMENTS

Lists elements can be **changed**, but not added, by using indexing

```
new_items = ['Coat', 'Backpack', 'Snowpants']
new_items[1] = 'Gloves'
new_items
['Coat', 'Gloves', 'Snowpants']
```

The second element in the list has changed from 'Backpack' to 'Gloves'

```
new_items = ['Coat', 'Backpack', 'Snowpants']
new_items[3] = 'Gloves'
new_items
```

The list only has 3 elements, so an index of 3 (the fourth element) does not exist

```
IndexError: list assignment index out of range
```



ADDING LIST ELEMENTS

You can `.append()` or `.insert()` a new element to a list

- `.append(element)` – adds an element to the end of the list
- `.insert(index, element)` – adds an element to the specified index in the list

```
item_list = ['Snowboard', 'Boots', 'Helmet', 'Goggles', 'Bindings']
item_list.append('Coat')
print(item_list)

['Snowboard', 'Boots', 'Helmet', 'Goggles', 'Bindings', 'Coat']
```

'Coat' was added as the last element in the list

```
item_list = ['Snowboard', 'Boots', 'Helmet', 'Goggles', 'Bindings']
item_list.insert(3, 'Coat')
item_list

['Snowboard', 'Boots', 'Helmet', 'Coat', 'Goggles', 'Bindings']
```

'Coat' was added as the fourth element in the list



COMBINING & REPEATING LISTS

Lists can be **combined**, or concatenated, with **+** and **repeated** with *****

```
item_list = ['Snowboard', 'Boots', 'Helmet', 'Goggles', 'Bindings']
new_items = ['Coat', 'Backpack', 'Snowpants']

all_items = item_list + new_items
print(all_items)

['Snowboard', 'Boots', 'Helmet', 'Goggles', 'Bindings', 'Coat',
'Backpack', 'Snowpants']
```

```
new_items * 3

['Coat',
 'Backpack',
 'Snowpants',
 'Coat',
 'Backpack',
 'Snowpants',
 'Coat',
 'Backpack',
 'Snowpants']
```



REMOVING LIST ELEMENTS

There are two ways to **remove** lists elements:

1. The **del** keyword deletes the selected elements from the list

```
new_items = ['Coat', 'Backpack', 'Snowpants']
del new_items[1:3]
new_items
['Coat']
```

The second (index of 1) and third (index of 2) elements were deleted from the list

- **del list_name(slice)**

2. The **.remove()** method deletes the first occurrence of the specified value from the list

```
new_items = ['Coat', 'Backpack', 'Snowpants']
new_items.remove('Coat')
new_items
['Backpack', 'Snowpants']
```

'Coat' was removed from the list

- **.remove(value)**



LIST FUNCTIONS

```
transactions = [10.44, 20.56, 200.14, 1242.66, 2.07, 8.01]
```

len(list_name) Returns the number of elements in a list

```
len(transactions)
```

6

min(list_name) Returns the smallest element in the list

```
min(transactions)
```

2.07

sum(list_name) Returns the sum of the elements in the list

```
sum(transactions)
```

1483.88

max(list_name) Returns the largest element in the list

```
max(transactions)
```

1242.66



SORTING LISTS

There are two ways to **sort** lists elements:

1. The **.sort()** method sorts the list permanently (*in place*)

```
transactions = [10.44, 20.56, 200.14, 1242.66, 2.07, 8.01]
transactions.sort()
transactions
[2.07, 8.01, 10.44, 20.56, 200.14, 1242.66]
```



PRO TIP: Don't sort in place until you're positive the code works as expected and you no longer need to preserve the original list

2. The **sorted** function returns a sorted list, but does not change the original (*not in place*)

```
transactions = [10.44, 20.56, 200.14, 1242.66, 2.07, 8.01]
sorted(transactions)
[2.07, 8.01, 10.44, 20.56, 200.14, 1242.66]

transactions
[10.44, 20.56, 200.14, 1242.66, 2.07, 8.01]
```



LIST METHODS

```
transactions = [10.44, 20.56, 200.14, 1242.66, 2.07, 8.01]
```

.index(*value*) Returns the index of a specified value within a list (returns -1 if not found)

```
transactions.index(200.14)
```

2

.count() Counts the number of times a given value occurs in a list

```
transactions.count(200.14)
```

1

.reverse() Reverses the order of the list elements in place

```
transactions.reverse()  
transactions
```

[8.01, 2.07, 1242.66, 200.14, 20.56, 10.44]



PRO TIP: Use a negative slice to reverse the order "not in place"



NESTED LISTS

Lists stored as elements of another list are known as **nested lists**

```
list_of_lists = [['a', 'b', 'c'],
                 ['d', 'e', 'f'],
                 ['g', 'h', 'i']]
list_of_lists
[['a', 'b', 'c'], ['d', 'e', 'f'], ['g', 'h', 'i']]
```

```
# grab the second element (a list)
list_of_lists[1]
['d', 'e', 'f']
```

Referencing a single index value will return an entire nested list

```
# grab the second element of the second element
list_of_lists[1][1]
'e'
```

Adding a second index value will return individual elements from nested lists



NESTED LISTS

List **methods** & **functions** still work with nested lists

```
list_of_lists = [['a', 'b', 'c'],
                 ['d', 'e', 'f'],
                 ['g', 'h', 'i']]
list_of_lists
[['a', 'b', 'c'], ['d', 'e', 'f'], ['g', 'h', 'i']]
```

```
list_of_lists[1].append('k')
list_of_lists
[['a', 'b', 'c'], ['d', 'e', 'f', 'k'], ['g', 'h', 'i']]
```

'k' is being added as the last element to the second list

```
list_of_lists[2].count('h')
```

1

'h' appears once in the third list



COPYING LISTS

There are 3 different ways to **copy** lists:

1. **Variable assignment** – assigning a list to a new variable creates a “view”
 - *Any changes made to one of the lists will be reflected in the other*
2. **.copy()** – applying this method to a list creates a ‘shallow’ copy
 - *Changes to entire elements (nested lists) will not carry over between original and copy*
 - *Changes to individual elements within a nested list will still be reflected in both*
3. **deepcopy()** – using this function on a list creates entirely independent lists
 - *Any changes made to one of the lists will NOT impact the other*



deepcopy is overkill for the vast majority of uses cases, but worth being aware of



TUPLES

Tuples are iterable data types capable of storing many individual items

- Tuples are very similar to lists, except they are **immutable**
- Tuple items can still be any data type, but they CANNOT be added, changed, or removed once the tuple is created

```
item_tuple = ('Snowboard', 'Boots', 'Helmet', 'Goggles', 'Bindings')  
item_tuple
```

```
('Snowboard', 'Boots', 'Helmet', 'Goggles', 'Bindings')
```

```
item_tuple[3]
```

```
'Goggles'
```

```
item_tuple[:3]
```

```
('Snowboard', 'Boots', 'Helmet')
```

```
len(item_tuple)
```

Tuples are created with parenthesis (), or the *tuple()* function

List operations that don't modify their elements work with tuples as well



WHY USE TUPLES?

1. Tuples require **less memory** than a list

```
import sys

item_list = ['Snowboard', 'Boots', 'Helmet', 'Goggles', 'Bindings']
item_tuple = ('Snowboard', 'Boots', 'Helmet', 'Goggles', 'Bindings')

print(f"The list is {sys.getsizeof(item_list)} " + 'Bytes')
print(f"The tuple is {sys.getsizeof(item_tuple)} " + 'Bytes')
```

The list is 120 Bytes
The tuple is 80 Bytes

The tuple is **33% smaller** than the list

For heavy data processing,
this can be a big difference

2. Operations **execute quicker** on tuples than on lists

```
import timeit

# calculate time of summing list 10000 times
print(timeit.timeit("sum([10.44, 20.56, 200.14, 1242.66, 2.07, 8.01])",
                     number=10000))

# calculate time of summing tuple 10000 times
print(timeit.timeit("sum((10.44, 20.56, 200.14, 1242.66, 2.07, 8.01))",
                     number=10000))
```

0.0076123750004626345
0.003229583000575076

The tuple executed over **twice as fast** as the list



WHY USE TUPLES?

3. Tuples **reduce user error** by preventing modification to data
 - There are cases in which you explicitly do not want others to be able to modify data

4. Tuples are common output in **imported functions**

```
a = 1  
b = 2  
c = 3  
  
a, b, c
```

```
(1, 2, 3)
```

Even asking to return multiple variables in a code cell returns them as a tuple



RANGES

Ranges are sequences of integers generated by a given start, stop, and step size

- They are more memory efficient than tuples, as they don't generate the integers until needed
- They save time, as you don't need to write the list of integers manually in the code
- They are commonly used with loops (*more on that later!*)

```
example_range = range(1, 5, 1)  
  
print(example_range)  
  
range(1, 5)
```

Note that printing a range does NOT return the integers

```
print(list(example_range))  
  
[1, 2, 3, 4]
```

You can retrieve them by converting to a list or tuple

```
print(tuple(range(len('Hello'))))  
  
(0, 1, 2, 3, 4)
```

LOOPS

LOOPS



In this section we'll introduce the concept of **loops**, review different loop types and their components, and cover control statements for refining their logic and handling errors

TOPICS WE'LL COVER:

Loop Basics

For Loops

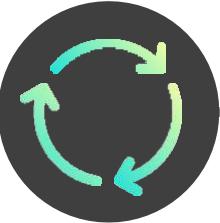
While Loops

Nested Loops

Loop Control

GOALS FOR THIS SECTION:

- Understand different types of loop structures, their components, and common use cases
- Learn to loop over multiple iterable data types in single and nested loop scenarios
- Explore common control statements for modifying loop logic and handling errors



LOOP BASICS

A **loop** is a block of code that will repeat until a given condition is met

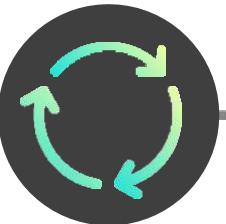
There are **two types** of loops:

FOR LOOPS

- Run a specified number of times
- “*For this many times*”
- This often corresponds with the length of a list, tuple, or other iterable data type
- Should be used when you know how many times the code should run

WHILE LOOPS

- Run until a logical condition is met
- “*While this is TRUE*”
- You usually don’t know how many times this loop will run, which can lead to infinite loop scenarios
(more on that later!)
- Should be used when you don’t know how many times the code should run



LOOP BASICS

A **loop** is a block of code that will repeat until a given condition is met

EXAMPLE

Converting elements in a price list from USD to Euros

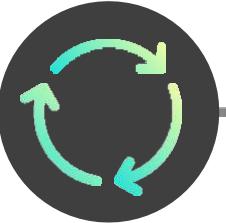
```
exchange_rate = 0.88
usd_list = [5.99, 9.99, 19.99, 24.99, 99.99]

euro_list = [round(usd_list[0] * exchange_rate, 2),
            round(usd_list[1] * exchange_rate, 2),
            round(usd_list[2] * exchange_rate, 2),
            round(usd_list[3] * exchange_rate, 2),
            round(usd_list[4] * exchange_rate, 2)]
euro_list

[5.27, 8.79, 17.59, 21.99, 87.99]
```

In this example we're taking each element in our **usd_list** and multiplying it by the exchange rate to convert it to euros. Notice that we had to write a line of code for *each element in the list*

- Imagine if we had hundreds or thousands of prices!



LOOP BASICS

A **loop** is a block of code that will repeat until a given condition is met

EXAMPLE

Converting elements in a price list from USD to Euros

```
exchange_rate = 0.88
usd_list = [5.99, 9.99, 19.99, 24.99, 99.99]
euro_list = []

for price in usd_list:
    euro_list.append(round(price * exchange_rate, 2))

print(euro_list)
```

[5.27, 8.79, 17.59, 21.99, 87.99]

Now we're using a **For Loop** to cycle through each element in the **usd_list** and convert it to Euros

- We only used *two lines of code* to process the entire list!



Don't worry if the code looks confusing now (we'll cover loop syntax shortly), the key takeaway is that we wrote the conversion **one time** and it was applied until we looped through **all the elements** in the list



FOR LOOPS

A **for loop** will run a specified number of times

- This often corresponds with the length of a list, tuple, or other iterable data type

for item in iterable:

Indicates
a For
Loop

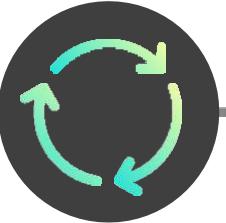
A variable name for
each item in the
iterable

Iterable object
to loop
through

Examples:

- Price
- Product
- Customer

Examples: • List
• Tuple
• String
• Dictionary
• Set



FOR LOOPS

A **for loop** will run a specified number of times

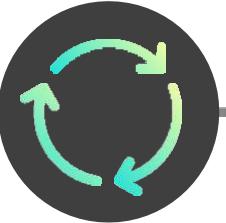
- This often corresponds with the length of a list, tuple, or other iterable data type

```
for item in iterable:  
    do this
```

Code to run until the loop terminates (must be indented!)

Run order:

1. *item = iterable[0]*
2. *item = iterable[1]*
3. *item = iterable[2]*
- n. item = iterable[len(iterable)-1]*



FOR LOOPS

EXAMPLE

Printing individual letters in a string

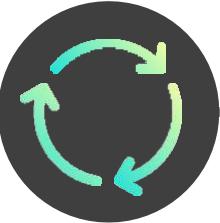
```
iterable = 'Maven'  
for item in iterable:  
    print(item)
```

M ← item = iterable[0]
a ← item = iterable[1]
v ← item = iterable[2]
e ← item = iterable[3]
n ← item = iterable[4]



How does this code work?

- Since '**Maven**' is a string, each letter is an **item** we'll iterate through
- **iterable** = ['M', 'a', 'v', 'e', 'n']
- The code will run and **print** each **item** in the **iterable**



FOR LOOPS

EXAMPLE E

Printing individual letters in a string

```
word = 'Maven'  
for letter in word:  
    print(letter)
```

M ← letter = word[0]
a ← letter = word[1]
v ← letter = word[2]
e ← letter = word[3]
n ← letter = word[4]



How does this code work?

- Since '**Maven**' is a string, we'll iterate through each **letter**
- **word** = ['M', 'a', 'v', 'e', 'n']
- The code will run and **print** each **letter** in the **word**



PRO TIP: Give the components of your loop intuitive names so they are easier to understand



LOOPING OVER ITEMS

Looping over items will run through the items of an iterable

- The loop will run as many times as there are items in the iterable

```
exchange_rate = 0.88
usd_list = [5.99, 9.99, 19.99, 24.99, 99.99]
euro_list = []

for price in usd_list:
    euro_list.append(round(price * exchange_rate, 2))

print(euro_list)

[5.27, 8.79, 17.59, 21.99, 87.99]
```

The for loop here is looping over the items, (elements) of usd_list, so the loop code block runs 5 times (length of the list):

1. $price = usd_list[0] = 5.99$
2. $price = usd_list[1] = 9.99$
3. $price = usd_list[2] = 19.99$
4. $price = usd_list[3] = 24.99$
5. $price = usd_list[4] = 99.99$



PRO TIP: To create a new list (or other data type) with loops, first create an empty list, then append values as your loop iterates



LOOPING OVER INDICES

Looping over indices will run through a range of integers

- You need to specify a range (usually the length of an iterable)
- This range can be used to navigate the indices of iterable objects

```
exchange_rate = 0.88
usd_list = [5.99, 9.99, 19.99, 24.99, 99.99]
euro_list = []

for i in range(len(usd_list)):
    euro_list.append(round(usd_list[i] * exchange_rate, 2))

print(euro_list)
[5.27, 8.79, 17.59, 21.99, 87.99]
```

The index is selecting

the elements in
the list

The for loop here is looping over indices in a range the size of the euro_list, meaning that the code will run 5 times (length of the list):

1. $i = 0$
2. $i = 1$
3. $i = 2$
4. $i = 3$
5. $i = 4$



PRO TIP: If you only need to access the elements of a single iterable, it's a best practice to loop over items instead of indices



LOOPING OVER MULTIPLE ITERABLES

Looping over indices can help with **looping over multiple iterables**, allowing you to use the same index for items you want to process together

EXAMPLE

Printing the price for each inventory item

```
euro_list = [5.27, 8.79, 17.59, 21.99, 87.99]
item_list = ['Snowboard', 'Boots', 'Helmet', 'Goggles', 'Bindings']

for i in range(len(euro_list)):
    print(f"The {item_list[i].lower()} costs {euro_list[i]} euros.")
```

```
The snowboard costs 5.27 euros.
The boots costs 8.79 euros.
The helmet costs 17.59 euros.
The goggles costs 21.99 euros.
The bindings costs 87.99 euros.
```

The for loop here is looping over indices in a range the size of the euro_list, meaning that the code will run 5 times (length of the list)

For the first run:

- $i = 0$
- $item_list[i] = \text{Snowboard}$
- $euro_list[i] = 5.27$



PRO TIP: ENUMERATE

The **enumerate** function will return *both* the index and item of each item in an iterable as it loops through

```
for index, element in enumerate(euro_list):
    print(index, element)
```

Use the index
for multiple
lists!

```
euro_list = [5.27, 8.79, 17.59, 21.99, 87.99]
item_list = ['Snowboard', 'Boots', 'Helmet', 'Goggles', 'Bindings']

for index, element in enumerate(euro_list):
    print(item_list[index], element)
```

```
Snowboard 5.27
Boots 8.79
Helmet 17.59
Goggles 21.99
Bindings 87.99
```



PRO TIP: Use enumerate if you want to loop over an index; it is slightly more efficient and considered best practice, as we are looping over an index derived from the list itself, rather than generating a new object to do so.

We're using the **index** of the *euro_list* to access each *element* from the *item_list*, and then printing each **element** of the *euro_list*



WHILE LOOPS

While loops run until a given logical expression becomes FALSE

- In other words, the loop runs *while* the expression is TRUE

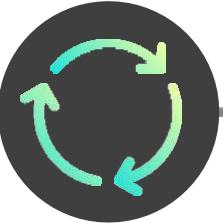
while logical expression:

Indicates
a While
Loop

A logical expression that
evaluates to TRUE or
FALSE

Examples:

- *counter < 10*
- *inventory > 0*
- *revenue > cost*



WHILE LOOPS

While loops run until a given logical expression returns FALSE

- In other words, the loop runs *while* the expression is TRUE

while logical expression:
do this

*Code to run while the
logical expression is
TRUE (must be
indented!)*



WHILE LOOPS

While loops often include **counters** that grow with each iteration

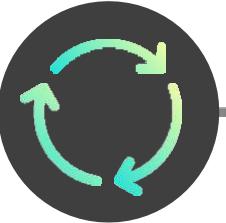
- Counters help us track how many times our loop has run
- They can also serve as a backup condition to exit a loop early (*more on this later!*)

```
counter = 0
while counter < 10:
    counter += 1
    print(counter)
```

- 1 ← Counter increases to 1 in the first iteration
2 ← Counter increases to 2 in the second iteration
. .
10 ← When the counter increments to 10, our condition becomes False, and exits the loop

This is an “addition assignment”, which adds a given number to the existing value of a variable:

```
counter = counter + 1
```



WHILE LOOPS

EXAMPLE

Run a calculation until a goal is reached

```
# starting portfolio balance is 800000
stock_portfolio = 800000
year_counter = 0

while stock_portfolio < 1000000:
    # calculate annual investment income
    investment_income = stock_portfolio * .05 # 5% interest rate

    # add income to end of year portfolio balance
    stock_portfolio += investment_income

    # add one each year
    year_counter += 1

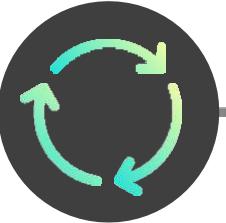
    print(f'At the end of year {year_counter}: '
          + f'My balance is ${round(stock_portfolio, 2)}')
```

At the end of year 1: My balance is \$840000.0
At the end of year 2: My balance is \$882000.0
At the end of year 3: My balance is \$926100.0
At the end of year 4: My balance is \$972405.0
At the end of year 5: My balance is \$1021025.25

The while loop here will run while stock_portfolio is less than 1m

stock_portfolio starts at 800k and increases by 5% of its value in each run:

1. $800k < 1m$ is TRUE
2. $840k < 1m$ is TRUE
3. $882k < 1m$ is TRUE
4. $926k < 1m$ is TRUE
5. $972k < 1m$ is TRUE
6. $1.02m < 1m$ is FALSE (exit)



WHILE LOOPS

EXAMPLE

Calculating bank balance until we're out of money.

```
bank_balance = 5000
month_counter = 0

while bank_balance > 0:
    spending = 1000
    bank_balance -= spending
    month_counter += 1
    print(f'At the end of month {month_counter}: '
          + f'My balance is ${round(bank_balance, 2)}')
```

At the end of month 1: My balance is \$4000
At the end of month 2: My balance is \$3000
At the end of month 3: My balance is \$2000
At the end of month 4: My balance is \$1000
At the end of month 5: My balance is \$0



PRO TIP: Use “`-=`” to subtract a number from a variable instead (subtraction assignment)



INFINITE LOOPS

A while loop that *always* meets its logical condition is known as an **infinite loop**

- These can be caused by incorrect logic or uncertainty in the task being solved

```
stock_portfolio = 800000
year_counter = 0

while stock_portfolio < 1000000:
    investment_income = stock_portfolio * 0 # 0% interest rate
    stock_portfolio += investment_income
    year_counter += 1
    print(f'At the end of year {year_counter}: '
          + f'My balance is ${round(stock_portfolio, 2)}')
```

```
At the end of year 51461: My balance is $800000
At the end of year 51462: My balance is $800000
At the end of year 51463: My balance is $800000
```

KeyboardInterrupt



This indicates a
manually stopped
execution

The while loop here will run while stock_portfolio is less than 1m, which will always be the case, as it's not growing due to 0% interest



If your program is stuck in an infinite loop, you will need to **manually stop it** and modify your logic





NESTED LOOPS

Loops can be **nested** within another loop

- The nested loop is referred to as an *inner* loop (the other is an *outer* loop)
- These are often used for navigating nested lists and similar data structures

EXAMPLE *Printing items along with their sizes*

```
item_list = ['Snowboard', 'Boots']
size_list = ['small', 'medium', 'large']

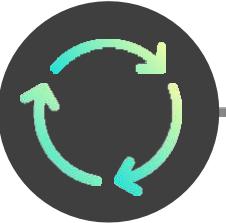
for item in item_list:
    for size in size_list:
        print(f"{item} is available in {size}.")
```

Snowboard is available in small. ← item_list[0], size_list[0]
Snowboard is available in medium. ← item_list[0], size_list[1]
Snowboard is available in large. ← item_list[0], size_list[2]
Boots is available in small. ← item_list[1], size_list[0]
Boots is available in medium. ← item_list[1], size_list[1]
Boots is available in large. ← item_list[1], size_list[2]



How does this code work?

- The inner loop (`size_list`) will run completely for each iteration of the outer loop (`item_list`)
- In this case, the inner loop iterated three times for each of the two iterations of the outer loop, for a total of six print statements
- **NOTE:** There is no limit to how many layers of nested loops can occur



NESTED LOOPS

EXAMPLE

Printing items along with their sizes

```
item_list = ['Snowboard', 'Boots']
size_list = ['small', 'medium', 'large']

for item in item_list:
    for size in size_list:
        print(f"{item} is available in {size}.")
```

Snowboard is available in small.

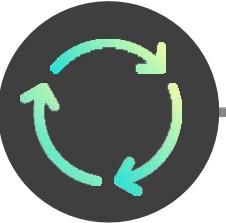


item_list (outer loop)

- 1 Snowboard
- 2 Boots

size_list (inner loop)

- 1 small
- 2 medium
- 3 large



NESTED LOOPS

EXAMPLE

Printing items along with their sizes

```
item_list = ['Snowboard', 'Boots']
size_list = ['small', 'medium', 'large']

for item in item_list:
    for size in size_list:
        print(f"{item} is available in {size}.")
```

Snowboard is available in small.
Snowboard is available in medium.

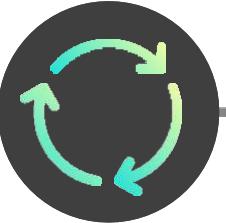


item_list (outer loop)

1	Snowboard
2	Boots

size_list (inner loop)

0	small
1	medium
2	large



NESTED LOOPS

EXAMPLE

Printing items along with their sizes

```
item_list = ['Snowboard', 'Boots']
size_list = ['small', 'medium', 'large']

for item in item_list:
    for size in size_list:
        print(f"{item} is available in {size}.")
```

Snowboard is available in small.
Snowboard is available in medium.
Snowboard is available in large.

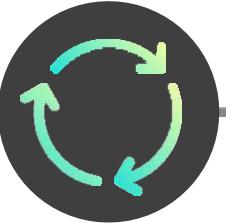


item_list (outer loop)

1	Snowboard
2	Boots

size_list (inner loop)

0	small
1	medium
2	large



NESTED LOOPS

EXAMPLE

Printing items along with their sizes

```
item_list = ['Snowboard', 'Boots']
size_list = ['small', 'medium', 'large']

for item in item_list:
    for size in size_list:
        print(f"{item} is available in {size}.")
```

Snowboard is available in small.
Snowboard is available in medium.
Snowboard is available in large.
Boots is available in small.

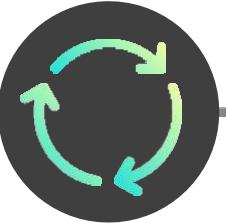


item_list (outer loop)

1	Snowboard
2	Boots

size_list (inner loop) loop

0	small
1	medium
2	large



NESTED LOOPS

EXAMPLE

Printing items along with their sizes

```
item_list = ['Snowboard', 'Boots']
size_list = ['small', 'medium', 'large']

for item in item_list:
    for size in size_list:
        print(f"{item} is available in {size}.")
```

Snowboard is available in small.
Snowboard is available in medium.
Snowboard is available in large.
Boots is available in small.
Boots is available in medium.

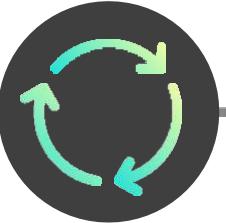


item_list (outer loop)

1 Snowboard
2 Boots

size_list (inner loop)

1 small
2 medium
3 large



NESTED LOOPS

EXAMPLE

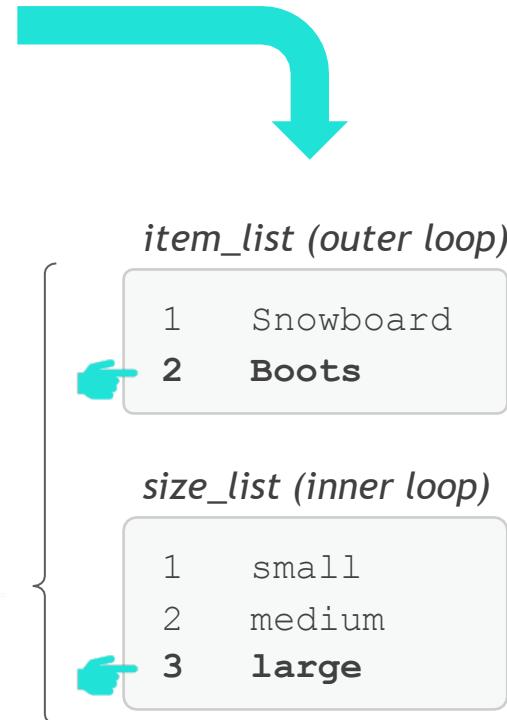
Printing items along with their sizes

```
item_list = ['Snowboard', 'Boots']
size_list = ['small', 'medium', 'large']

for item in item_list:
    for size in size_list:
        print(f"{item} is available in {size}.")
```

Snowboard is available in small.
Snowboard is available in medium.
Snowboard is available in large.
Boots is available in small.
Boots is available in medium.
Boots is available in large.

*You exit a nested loop
when
all its loops are
completed*





LOOP CONTROL

Loop control statements help refine loop behavior and handle potential errors

- These are used to change the flow of loop execution based on certain conditions

**Brea
k**



Stops the loop before completion

Good for avoiding infinite loops & exiting loops early

**Continu
e**



Skips to the next iteration in the loop

Good for excluding values that you don't want to process in a loop

**Pas
s**



Serves as a placeholder for future code

Good for avoiding run errors with incomplete code logic

**Try,
Except**



Help with error and exception handling

Good for resolving errors in a loop without stopping its execution midway



BREAK

Triggering a **break** statement will exit the loop that it lives in

- This helps exit potential infinite loops when they can't be avoided by refining our logic
- It also helps set logical conditions to exit for loops early

```
subtotals = [15.98, 899.97, 799.97, 117.96, 5.99,
             599.99, 24.99, 1799.94, 99.99]
revenue = 0

for subtotal in subtotals:
    revenue += subtotal
    print(round(revenue, 2))
    if revenue > 2000:
        break

15.98
915.95
1715.92
1833.88
1839.87
2439.86
```

The *for* loop here would normally run the length of the entire *subtotals* list (9 iterations), but the **break** statement triggers once the *revenue* is greater than 2,000 after the 6th transaction



BREAK

Triggering a **break** statement will exit the loop that it lives in

- This helps exit potential infinite loops when they can't be avoided by refining our logic
- It also helps set logical conditions to exit for loops early

```
stock_portfolio = 100
year_counter = 0

while stock_portfolio < 1000000:
    investment_income = stock_portfolio * .05
    stock_portfolio += investment_income
    year_counter += 1
    print(f'My balance is ${round(stock_portfolio, 2)} in year {year_counter}')
    # break if i can't retire in 30 years
    if year_counter >= 30:
        print('Guess I need to save more.')
        break
```

```
My balance is $105.0 in year 1
My balance is $110.25 in year 2
My balance is $115.76 in year 3
My balance is $121.55 in year 4
```

```
My balance is $411.61 in year 29
My balance is $432.19 in year 30
Guess I need to save more.
```

The while loop here will run while stock_portfolio is less than 1,000,000 (this would take 190 iterations/years)

A **break** statement is used inside an IF function here to exit the code in case the year_counter is greater than 30



PRO TIP: Use a counter and a combination of IF and break to set a max number of iterations



CONTINUE

Triggering a **continue** statement will move on to the next iteration of the loop.

- No other lines in that iteration of the loop will run
 - This is often combined with logical criteria to exclude values you don't want to process

```
item_list = ['ski-extreme', 'snowboard-basic', 'snowboard-extreme',
             'snowboard-comfort', 'ski-comfort', 'ski-backcountry']

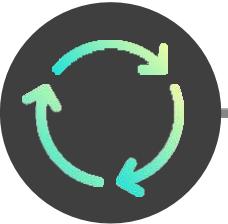
snowboards = []

for item in item_list:
    if 'ski' in item:
        continue
    snowboards.append(item)

print(snowboards)
```

['snowboard-basic', 'snowboard-extreme', 'snowboard-comfort']

A *continue* statement is used inside an IF statement here to avoid appending “ski” items to the snowboards list



PASS

A **pass** statement serves as a placeholder for future code

- Nothing happens and the loop continues to the next line of code

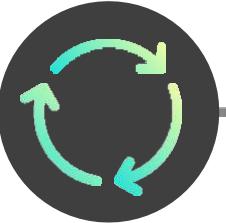
```
item_list = ['ski-extreme', 'snowboard-basic', 'snowboard-extreme',
             'snowboard-comfort', 'ski-comfort', 'ski-backcountry']

snowboards = []

for item in item_list:
    if 'ski' in item:
        pass # need to write complicated logic later!
    snowboards.append(item)

snowboards
['ski-extreme',
 'snowboard-basic',
 'snowboard-extreme',
 'snowboard-comfort',
 'ski-comfort',
 'ski-backcountry']
```

The **pass** statement is used in place of the eventual logic that will live there, avoiding an error in the meantime



TRY, EXCEPT

The **try** & **except** statements resolve errors in a loop without stopping its execution

- **Try:** indicates the first block of code to run (which could result in an error)
- **Except:** indicates an optional block of code to run in case of an error in the try block

```
price_list = [5.99, None, 19.99, 24.99, 0, '74.99', 99.99]

# loop to calculate how many of each item I can buy
for price in price_list:
    affordable_quantity = 50//price # My budget is 50 dollars
    print(f"I can buy {affordable_quantity} of these.")
```

`TypeError: unsupported operand type(s) for //: 'int' and 'NoneType'`

*This for loop was stopped by a
TypeError in the second
iteration*



TRY, EXCEPT

The **try** & **except** statements resolve errors in a loop without stopping its execution

- **Try:** indicates the first block of code to run (which could result in an error)
- **Except:** indicates an optional block of code to run in case of an error in the try block

```
price_list = [5.99, None, 19.99, 24.99, 0, '74.99', 99.99]

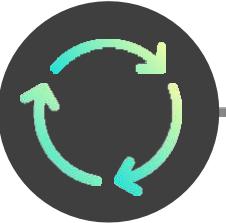
# loop to calculate how many of each item I can buy
for price in price_list:
    affordable_quantity = 50//price # My budget is 50 dollars
    print(f"I can buy {affordable_quantity} of these.)
```

Placing the code in a **try** statement handles the errors via the **except** statement without stopping the loop

TypeError: unsupported operand type(s) for //: 'int' and 'NoneType'

```
I can buy 8 of these.
The price seems to be missing.
I can buy 2 of these.
I can buy 2 of these.
The price seems to be missing.
The price seems to be missing.
I can buy 0 of these.
```

Are 0 and '74.99' missing prices, or do we need to treat these exceptions differently?



TRY, EXCEPT

The **try** & **except** statements resolve errors in a loop without stopping its execution

- **Try:** indicates the first block of code to run (which could result in an error)
- **Except:** indicates an optional block of code to run in case of an error in the try block

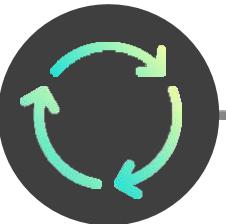
```
price_list = [5.99, None, 19.99, 24.99, 0, '74.99', 99.99]

for price in price_list:
    try:
        affordable_quantity = 50//price
```

*The 0 price in the
price_list returns a
ZeroDivisionError*

```
50//0
```

```
ZeroDivisionError: integer division or modulo by zero
```



TRY, EXCEPT

The **try** & **except** statements resolve errors in a loop without stopping its execution

- **Try:** indicates the first block of code to run (which could result in an error)
- **Except:** indicates an optional block of code to run in case of an error in the try block

```
price_list = [5.99, None, 19.99, 24.99, 0, '74.99', 99.99]

# loop to calculate how many of each item I can buy
for price in price_list:
    try:
        affordable_quantity = 50//price # My budget is 50 dollars
        print(f"I can buy {affordable_quantity} of these.")
    except ZeroDivisionError:
        print("This product is free, I can take as many as I like.")
    except:
        print("That's not a number")
```

```
I can buy 8.0 of these.
That's not a number
I can buy 2.0 of these.
I can buy 2.0 of these.
This product is free, I can take as many as I like.
That's not a number
I can buy 0.0 of these.
```

If anything in the **try** block returns a `ZeroDivisionError`, the first **except** statement will run
The second **except** statement will run on any other error types



PRO TIP: Add multiple `except` statements for different error types to handle each scenario differently

DICTIONARIES & SETS

DICTIONARIES & SETS



In this section we'll cover **dictionaries** and **sets**, two iterable data types with helpful use cases that allow for quick information retrieval and store unique values

TOPICS WE'LL COVER:

List Limitations

Dictionary Basics

Modifying Dictionaries

Dictionary Methods

Nested Dictionaries

Sets

GOALS FOR THIS SECTION:

- Learn to store, append, and look up data in dictionaries and some helpful dictionary methods
- Build dictionaries using the `zip()` function
- Use set operations to find relationships in the values across multiple datasets



LIST LIMITATIONS

Lists can be inefficient when you need to look up specific values, as they can only be accessed via indexing

EXAMPLE

Looking up the inventory status for goggles

```
inventory_status = [['skis', 'in stock'], ['snowboard', 'sold out'],
                     ['goggles', 'sold out'], ['boots', 'in stock']]

item_looking_for = 'goggles'

for item in inventory_status:
    if item[0] == item_looking_for:
        print(item[1])

sold out
```

The nested list is used to pair unique items with their inventory status ('in stock' or 'sold out')

To find the status for a specific item, you need to iterate through the list to find the item of interest, then grab its corresponding inventory status



DICTIONARY BASICS

Dictionary values can be lists, so to access individual attributes:

1. Retrieve the **list** by looking up its **key**
2. Retrieve the **list element** by using its **index**

EXAMPLE

Looking up the stock quantity for skis

```
item_details = {'skis': [249.99, 10, 'in stock'],
                'snowboard': [219.99, 0, 'sold out'],
                'goggles': [99.99, 0, 'sold out'],
                'boots': [79.99, 7, 'in stock']}
```

```
item_details['skis']
```

```
[249.99, 10, 'in stock']
```

The key is the item name, and the value is a list storing item price, inventory, and inventory status

```
item_details['skis'][1]
```

```
10
```

This returns the second element (index of 1) in the list stored as the value for the key 'skis'



DICTIONARY BASICS

You can conduct **membership tests** on dictionary keys

```
item_details = {'skis': [249.99, 10, 'in stock'],
                 'snowboard': [219.99, 0, 'sold out'],
                 'goggles': [99.99, 0, 'sold out'],
                 'boots': [79.99, 7, 'in stock']}

print('skis' in item_details)
print('bindings' in item_details)
```

True

False

And you can **loop through*** them

```
for item in item_details:
    print(item, item_details[item])
```

```
skis [249.99, 10, 'in stock']
snowboard [219.99, 0, 'sold out']
goggles [99.99, 0, 'sold out']
boots [79.99, 7, 'in stock']
```

Note that the items that are being looped over are the dictionary keys



MODIFYING DICTIONARIES

Referencing a new key and assigning it a value will **add a new key-value pair**, while referencing an existing key will **overwrite the existing pair**

```
item_details = {'skis': [249.99, 10, 'in stock'],
                'snowboard': [219.99, 0, 'sold out'],
                'goggles': [99.99, 0, 'sold out'],
                'boots': [79.99, 7, 'in stock']}
```

Notice that 'bindings' is not in the dictionary keys

```
item_details['bindings'] = [149.99, 4, 'in stock']
item_details
```

```
{'skis': [249.99, 10, 'in stock'],
 'snowboard': [219.99, 0, 'sold out'],
 'goggles': [99.99, 0, 'sold out'],
 'boots': [79.99, 7, 'in stock'],
 'bindings': [149.99, 4, 'in stock']}
```

Therefore, assigning a value to a key of 'bindings' adds the key-value pair to the dictionary

```
item_details['bindings'] = [139.99, 0, 'out of stock']
item_details
```

```
{'skis': [249.99, 10, 'in stock'],
 'snowboard': [219.99, 0, 'sold out'],
 'goggles': [99.99, 0, 'sold out'],
 'boots': [79.99, 7, 'in stock'],
 'bindings': [139.99, 0, 'out of stock']}
```

Now that 'bindings' is a key in the dictionary, assigning a value to a key of 'bindings' overwrites the value in the dictionary for that key



MODIFYING DICTIONARIES

Use the **del** keyword to delete key-value pairs

```
item_details
```

```
{'skis': [249.99, 10, 'in stock'],
'snowboard': [219.99, 0, 'sold out'],
'goggles': [99.99, 0, 'sold out'],
'boots': [79.99, 7, 'in stock'],
'bindings': [149.99, 4, 'in stock']}
```

```
del item_details['boots']
```

```
item_details
```

```
{'skis': [249.99, 10, 'in stock'],
'snowboard': [219.99, 0, 'sold out'],
'goggles': [99.99, 0, 'sold out'],
'bindings': [139.99, 0, 'out of stock']}
```

This has deleted the key-value pair with a key of 'boots' from the dictionary



DICTIONARY METHODS

keys

Returns the keys from a dictionary

`.keys()`**values**

Returns the values from a dictionary

`.values()`**items**

Returns key value pairs from a dictionary as a list of tuples

`.items()`**get**

Returns a value for a given key, or an optional value if the key isn't found

`.get(key, value if key not found)`**update**

Appends specified key-value pairs, including entire dictionaries

`.update (key:value pairs)`



KEYS

The `.keys()` method returns the keys from a dictionary

```
item_details = {'skis': [249.99, 10, 'in stock'],
                'snowboard': [219.99, 0, 'sold out'],
                'goggles': [99.99, 0, 'sold out'],
                'boots': [79.99, 7, 'in stock']}
item_details.keys()
dict_keys(['skis', 'snowboard', 'goggles', 'boots'])
```

`.keys()` returns a **view object** that represents the keys as a list
(this is more memory efficient than creating a list)

```
for item in item_details.keys():
    print(item)
```

```
skis
snowboard
goggles
boots
```

This view object can be iterated through, which has the same behavior as looping through the dictionary keys directly

```
key_list = list(item_details.keys())
print(key_list)
['skis', 'snowboard', 'goggles', 'boots']
```

This view object can be converted into a list or a tuple if needed



VALUES

The `.values()` method returns the values from a dictionary

```
item_details = {'skis': [249.99, 10],  
                'snowboard': [219.99, 0],  
                'goggles': [99.99, 0],  
                'boots': [79.99, 7]}  
  
item_details.values()  
  
dict_values([[249.99, 10], [219.99, 0], [99.99, 0], [79.99, 7]])
```

`.values()` returns a **view object** that represents the values as a list
(this is more memory efficient than creating a list)

```
price_list = []  
for attribute in item_details.values():  
    price_list.append(attribute[0])  
  
price_list  
  
[249.99, 219.99, 99.99, 79.99]
```

This view object can be looped through as well
Here we're grabbing the first element from each of the lists returned by `.values()` and appending them to a new list



ITEMS

The `.items()` method returns key-value pairs from a dictionary as a list of tuples

```
item_details = {'skis': [249.99, 10],  
                'snowboard': [219.99, 0],  
                'goggles': [99.99, 0],  
                'boots': [79.99, 7]}  
  
item_details.items()  
  
dict_items([('skis', [249.99, 10]), ('snowboard', [219.99, 0]),  
           ('goggles', [99.99, 0]), ('boots', [79.99, 7]))]
```

`.items()` returns a **view object** that represents the key-value pairs as a list of tuples

```
for key, value in item_details.items():  
    print(f'The {key} costs {value[0]}.')
```

```
The skis costs 249.99.  
The snowboard costs 219.99.  
The goggles costs 99.99.  
The boots costs 79.99.
```

You can **unpack** the tuple to retrieve individual keys and values

In this case, the variable 'key' is assigned to the key in the tuple, and 'value' is assigned to the dictionary value



ITEMS

The `.items()` method returns key-value pairs from a dictionary as a list of tuples

```
item_details = {'skis': [249.99, 10],  
                'snowboard': [219.99, 0],  
                'goggles': [99.99, 0],  
                'boots': [79.99, 7]}  
  
item_details.items()  
  
dict_items([('skis', [249.99, 10]), ('snowboard', [219.99, 0]),  
           ('goggles', [99.99, 0]), ('boots', [79.99, 7]))]
```

`.items()` returns a **view object** that represents the key-value pairs as a list of tuples

```
for item, item_attributes in item_details.items():  
    print(f'The {item} costs {item_attributes[0]}.)')
```

```
The skis costs 249.99.  
The snowboard costs 219.99.  
The goggles costs 99.99.  
The boots costs 79.99.
```

You can give these variables intuitive names, although `k, v` is common to represent keys and values



GET

The `.get()` method returns the values associated with a dictionary key

- It won't return a `KeyError` if the key isn't found
- You can specify an optional value to return if the key is not found

```
item_details = {'skis': [249.99, 10, 'in stock'],
                'snowboard': [219.99, 0, 'sold out'],
                'goggles': [99.99, 0, 'sold out'],
                'boots': [79.99, 7, 'in stock']}

item_details.get('boots')

[79.99, 7, 'in stock']
```

.get() returns the value associated with the 'boots' key

```
item_details['bindings']

KeyError: 'bindings'
```

```
item_details.get('bindings')
```

```
item_details.get('bindings', "Sorry we don't carry that item.")
```

```
"Sorry we don't carry that item."
```

The difference between using `.get()` and simply entering the key directly is that `.get()` will not return an error if the key is not found

And you can specify an optional value to return if the key is not found

- `.get(key, value if key not found)`



UPDATE

The `.update()` method appends key-value pairs to a dictionary

```
item_details = {'skis': [249.99, 10, 'in stock'],
                'snowboard': [219.99, 0, 'sold out'],
                'goggles': [99.99, 0, 'sold out'],
                'boots': [79.99, 7, 'in stock']}

item_details.update({'bindings': [139.99, 0, 'out of stock']})
item_details
```

```
{'skis': [249.99, 10, 'in stock'],
 'snowboard': [219.99, 0, 'sold out'],
 'goggles': [99.99, 0, 'sold out'],
 'boots': [79.99, 7, 'in stock'],
 'bindings': [139.99, 0, 'out of stock']}
```

`.update()` appends new key-value pairs to a dictionary, in this case a single pair for a key of 'bindings'

- `.update(key:value pairs)`

```
new_items = {'scarf': [19.99, 100, 'in stock'], 'snowpants': 'N/A'}

item_details.update(new_items)
item_details
```

```
{'skis': [249.99, 10, 'in stock'],
 'snowboard': [219.99, 0, 'sold out'],
 'goggles': [99.99, 0, 'sold out'],
 'boots': [79.99, 7, 'in stock'],
 'scarf': [19.99, 100, 'in stock'],
 'snowpants': 'N/A'}
```

This is the preferred way to **combine dictionaries**

As a reminder, dictionary values do not need to be the same type; note that the value for 'snowpants' is 'N/A', while the values for the rest of the keys are lists



ZIP

The **zip()** function combines two iterables, like lists, into a single iterator of tuples

- `first_iterable[0]` gets paired with `second_iterable[0]`, and so on

```
item_list = ['skis', 'snowboard', 'goggles', 'boots']
price_list = [249.99, 219.99, 99.99, 79.99]
inventory = [10, 0, 0, 7]

zip(price_list, inventory)

<zip at 0x7fb0f012f580>
```

Here, we're zipping together two lists and returning a **zip object** that contains the instructions for pairing the i^{th} object from each iterable

```
item_attributes = list(zip(price_list, inventory))

item_attributes

[(249.99, 10), (219, 0), (99, 0), (99.99, 7)]
```

When you create a list from the zip object, you get a list with the i^{th} element from each iterable paired together in a tuple

```
list(zip(item_list, price_list, inventory))

[('skis', 249.99, 10),
 ('snowboard', 219, 0),
 ('goggles', 99, 0),
 ('boots', 99.99, 7)]
```

Any number of iterables can be combined this way



ZIP

The `zip()` function is commonly used to build dictionaries

```
item_details = [[249.99, 10, 'in stock'],
                [219.99, 0, 'sold out'],
                [99.99, 0, 'sold out'],
                [79.99, 7, 'in stock']]

item_names = ['skis', 'snowboard', 'goggles', 'boots']

item_dict = dict(zip(item_names, item_details))

item_dict
{'skis': [249.99, 10, 'in stock'],
 'snowboard': [219.99, 0, 'sold out'],
 'goggles': [99.99, 0, 'sold out'],
 'boots': [79.99, 7, 'in stock']}
```

When creating a dictionary from the `zip` object, the elements of the first iterable become the **keys**, and the second become the **values**

```
dict(zip(item_list, price_list, inventory))
```

`ValueError: dictionary update sequence element #0 has length 3; 2 is required`

Note that you can only create a dictionary from a `zip` object with two iterables

```
item_dict = dict(zip(item_list, zip(price_list, inventory)))

item_dict
{'skis': (249.99, 10),
 'snowboard': (219, 0),
 'goggles': (99, 0),
 'boots': (99.99, 7)}
```

But you can zip iterables together within the second argument



NESTED DICTIONARIES

You can **nest dictionaries** as values of another dictionary

- The nested dictionary is referred to as an *inner* dictionary (the other is an *outer* dictionary)

```
item_history = {
    2019: {"skis": [249.99, 10, "in stock"], "snowboard": [219.99, 0, "sold out"]},
    2020: {"skis": [259.99, 10, "in stock"], "snowboard": [229.99, 0, "sold out"]},
    2021: {"skis": [269.99, 10, "in stock"], "snowboard": [239.99, 0, "sold out"]},
}
```

```
item_history
```

```
{2019: {'skis': [249.99, 10, 'in stock'],
'snowboard': [219.99, 0, 'sold out']},
2020: {'skis': [259.99, 10, 'in stock'],
'snowboard': [229.99, 0, 'sold out']},
2021: {'skis': [269.99, 10, 'in stock'],
'snowboard': [239.99, 0, 'sold out']}}}
```

The outer dictionary here has years as keys, and inner dictionaries as values

The inner dictionaries have items as keys, and lists with item attributes as values

```
item_history[2020]
```

```
{'skis': [259.99, 10, 'in stock'], 'snowboard': [229.99, 0, 'sold out']}
```

To access an inner dictionary, reference the outer dictionary key

```
item_history[2020]['skis']
```

```
[259.99, 10, 'in stock']
```

To access the values of an inner dictionary, reference the outer dictionary key, then the inner dictionary key of interest



SETS

A **set** is a collection of unique values

- Sets are **unordered**, which means their values cannot be accessed via index or key
- Sets are mutable (values can be added/removed), but set *values* must be **unique & immutable**

```
my_set = {'snowboard', 'snowboard', 'skis', 'snowboard', 'sled'}  
my_set  
{'skis', 'sled', 'snowboard'}
```

Sets can be created with
curly braces `{}`

The absence of colons
differentiates them from
dictionaries

```
my_set = set(['snowboard', 'snowboard', 'skis', 'snowboard', 'sled'])  
my_set  
{'skis', 'sled', 'snowboard'}
```

Sets can also be created via
conversion using `set()`

Note that duplicate values
are automatically removed
when created



WORKING WITH SETS

You can conduct **membership tests** on sets

```
my_set
```

```
{'skis', 'sled', 'snowboard'}
```

```
'snowboard' in my_set
```

```
True
```

You can **loop through** them

```
for value in my_set:  
    print(value)
```

```
snowboard  
sled  
skis
```

But you **can't index** them (*they are unordered*)

```
my_set[0]
```

```
TypeError: 'set' object is not subscriptable
```



SET OPERATIONS

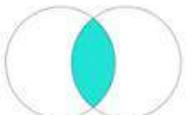
Python has useful **operations** that can be performed between sets



union

Returns all unique values in both sets

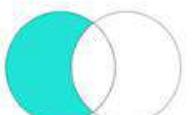
`set1.union(set2)`



intersection

Returns values present in both sets

`set1.intersection(set2)`



difference

Returns values present in set 1, but not set 2

`set1.difference(set2)`



symmetric difference

Returns values not shared between sets
(opposite of intersection)

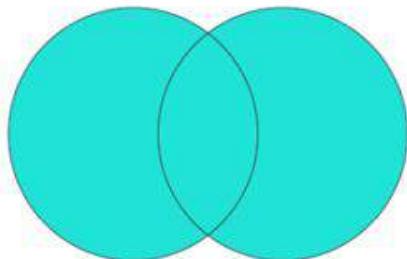
`set1.symmetric_difference(set2)`



PRO TIP: Chain set operations to capture the relationship between three or more sets, for example - `set1.union(set2).union(set3)`



UNION



Union returns all unique values in both sets

```
friday_items = {'snowboard', 'snowboard', 'skis', 'snowboard', 'sled'}  
saturday_items = {'goggles', 'helmet', 'snowboard', 'skis', 'goggles'}  
  
friday_items.union(saturday_items)  
  
{'goggles', 'helmet', 'skis', 'sled', 'snowboard'}
```

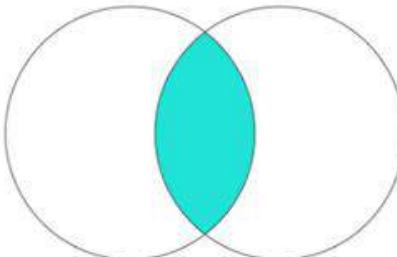
All values from both sets are returned, without duplicates

```
sunday_items = {'coffee'}  
  
friday_items.union(saturday_items).union(sunday_items)  
  
{'coffee', 'goggles', 'helmet', 'skis', 'sled', 'snowboard'}
```

All values from the three sets are returned by chaining two union operations



INTERSECTION



Intersection returns the values present in both sets

```
friday_items = {'snowboard', 'snowboard', 'skis', 'snowboard', 'sled'}  
saturday_items = {'goggles', 'helmet', 'snowboard', 'skis', 'goggles'}  
  
friday_items.intersection(saturday_items)  
  
{'skis', 'snowboard'}
```

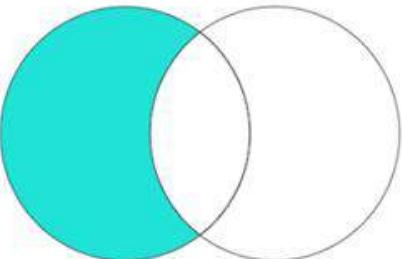
Only the values in both sets are returned, without duplicates

```
sunday_items = {'coffee'}  
  
friday_items.intersection(saturday_items).intersection(sunday_items)  
  
set()
```

Since no value is present in all three sets, an empty set is returned



DIFFERENCE



Difference returns the values present in set 1, but not set 2 (*the order matters*)

```
friday_items = {'snowboard', 'snowboard', 'skis', 'snowboard', 'sled'}  
saturday_items = {'goggles', 'helmet', 'snowboard', 'skis', 'goggles'}  
  
friday_items.difference(saturday_items)  
  
{'sled'}
```

'sled' is the only value in friday_items
that is NOT in saturday_items

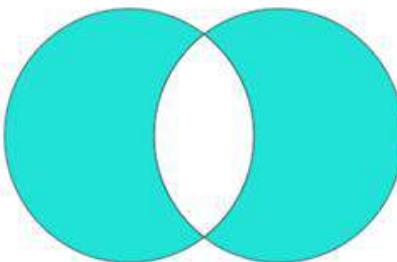
```
saturday_items - friday_items  
  
{'goggles', 'helmet'}
```

If you reverse the order, the output changes – 'goggles' and 'helmet' are
in saturday_items but NOT in friday_items

Note that the subtraction sign can be used instead of difference



SYMMETRICAL DIFFERENCE



Symmetrical difference returns all values not shared between sets

```
friday_items = {'snowboard', 'snowboard', 'skis', 'snowboard', 'sled'}  
saturday_items = {'goggles', 'helmet', 'snowboard', 'skis', 'goggles'}  
  
friday_items.symmetric_difference(saturday_items)  
  
{'goggles', 'helmet', 'sled'}
```

'sled' is only in set 1, and 'goggles' and 'helmet' are only in set 2



SET USE CASES

1. Sets are more efficient than lists for performing **membership tests**

```
time_list = list(range(1000000))
time_set = set(range(1000000))
```

Using

Lists

```
%%time
100000 in time_list
```

CPU times: user 5.07 ms, sys: 454 µs, total: 5.53 ms

Using

Sets

```
%%time
100000 in time_set
```

CPU times: user 5 µs, sys: 1e+03 ns, total: 6 µs



Sets are implemented as **hash tables**, which makes looking up values extremely fast; the downside is that they cannot preserve order (lists rely on dynamic arrays that preserve order but have slower performance)



SET USE CASES

2. Sets can **gather unique values** efficiently without looping

*Using
Lists*

```
%%time
unique_items = []

for item in shipments_today:
    if item not in unique_items:
        unique_items.append(item)

unique_items
```

```
CPU times: user 27 µs, sys: 1 µs, total: 28 µs
['ski', 'snowboard', 'helmet', 'hat', 'goggles']
```

*Using
Sets*

```
%%time
list(set(shipments_today))
```

```
CPU times: user 9 µs, sys: 0 ns, total: 9 µs
['snowboard', 'ski', 'helmet', 'hat', 'goggles']
```



SET USE CASES

3. Set operations can find the **data shared, or not shared, between items** without looping

*Using
Lists*

```
shipment_today = ['ski', 'snowboard', 'ski', 'ski', 'helmet', 'hat', 'goggles']
shipment_yesterday = ['hat', 'goggles', 'snowboard', 'hat', 'bindings']

unique_today = []
for item_t in shipment_today:
    if item_t not in shipment_yesterday:
        if item_t not in unique_today:
            unique_today.append(item_t)

unique_today

['ski', 'helmet']
```

*Using
Sets*

```
set(shipment_today).difference(set(shipment_yesterday))

{'helmet', 'ski'}
```

FUNCTIONS

FUNCTIONS



In this section we'll learn to write custom **functions** to boost efficiency, import external functions stored in modules or packages, and write comprehensions

TOPICS WE'LL COVER:

Function Components

Defining Functions

Variable Scope

Modules

Packages

Lambda Functions

Comprehensions

GOALS FOR THIS SECTION:

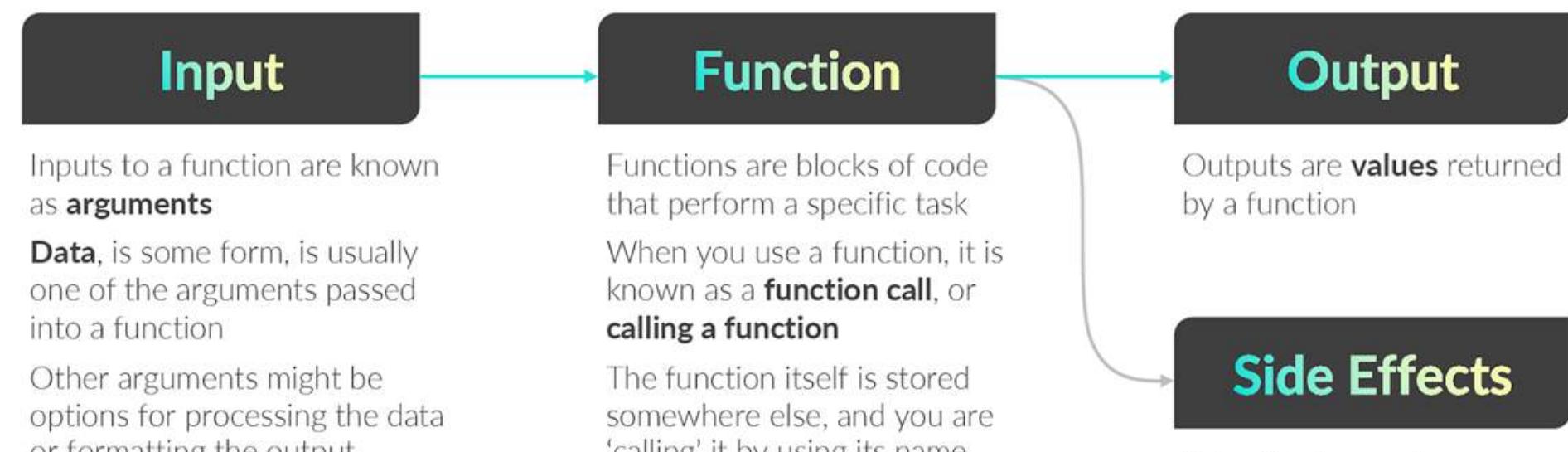
- Identify the key components of Python functions
- Define custom functions and manage variable scope
- Practice importing packages, which are external collections of functions
- Learn how to write lambda functions and apply functions with map()
- Learn how to write comprehensions, powerful expressions used to create iterables



FUNCTIONS

Functions are reusable blocks of code that perform specific tasks when called

- They take in data, perform a set of actions, and return a result



Functions help **boost efficiency** as programmers immensely!

THE ANATOMY OF MAX

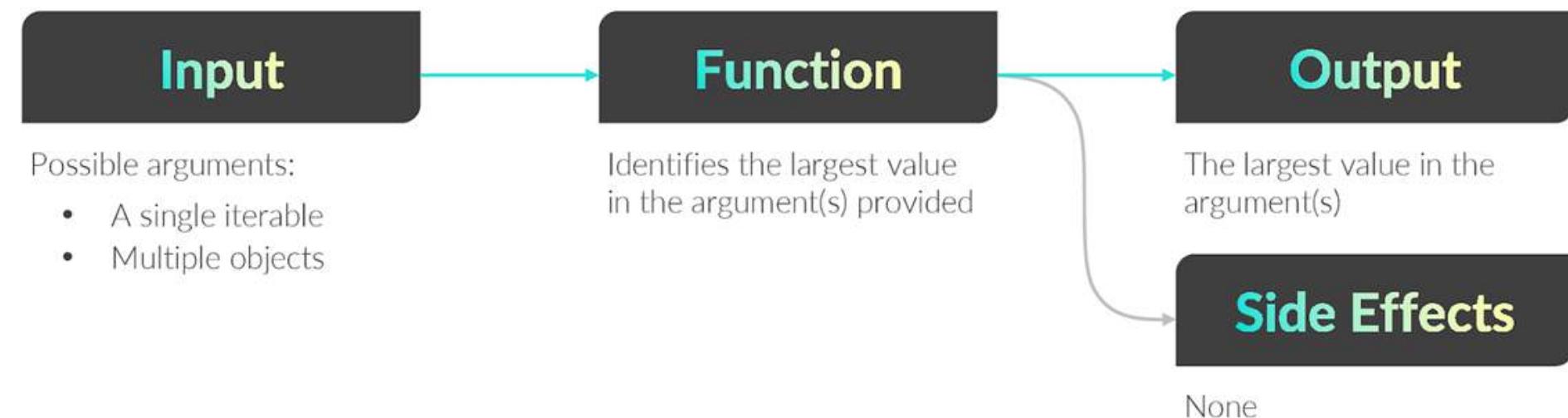
The built-in **max()** function works the same way (as do others like `len()`, `sum()`, etc.)

```
In [5]: max?

Docstring:
max(iterable, *[, default=obj, key=func]) -> value
max(arg1, arg2, *args, *[, key=func]) -> value

With a single iterable argument, return its biggest item. The
default keyword-only argument specifies an object to return if
the provided iterable is empty.
With two or more arguments, return the largest argument.

The documentation provides details on the
input, function, and output for max()
```



THE ANATOMY OF MAX

The built-in **max()** function works the same way (as do others like **len()**, **sum()**, etc.)

```
price_list = [5.99, 19.99, 24.99, 0, 74.99, 99.99]  
max(price_list)
```





DEFINING A FUNCTION

```
def function_name(arguments):
```



Indicates you're defining a new function



An intuitive name to call your function by



Variable names for the function inputs, separated by commas

Examples:

- avg
- usd_converter

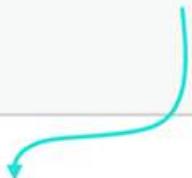


Functions have the same **naming rules** and best practices as variables – use 'snake_case'!



DEFINING A FUNCTION

```
def function_name(arguments):  
    do this
```



*Code block for the function
that uses the arguments to
perform a specific task*



DEFINING A FUNCTION

```
def function_name(arguments):  
    do this  
    return output
```

Ends the function
(without it, the function
returns None)

Values to return
(usually variables)



DEFINING A FUNCTION

EXAMPLE

Defining a function that concatenates two words, separated by a space

```
def concatenator(string1, string2):
    combined_string = string1 + ' ' + string2
    return combined_string
```

Here we're defining a function called **concatenator**, which accepts two arguments, combines them with a space , and returns the result

```
concatenator('Hello', 'World!')
```

```
'Hello World!'
```

When we call this function with two string arguments, the combined string is returned

```
def concatenator(string1, string2):
    return string1 + ' ' + string2
```

Note that we don't need a code block before return, here we're combining strings in the return statement



WHEN TO DEFINE A FUNCTION?

You should take time to **define a function** when:

- ✓ You find yourself copying & pasting a block of code repeatedly
- ✓ You know you will use a block of code again in the future
- ✓ You want to share a useful piece of logic with others
- ✓ You want to make a complex program more readable



PRO TIP: There are no hard and fast rules, but in data analysis you'll often have a similar workflow for different projects – by taking the time to package pieces of your data cleaning or analysis workflow into functions, you are saving your future self (and your colleagues') time



THE DOCSTRING

You can create a **docstring** for your function

- This is used to embed text describing its arguments and the actions it performs

Use **triple quotes** inside the function to create its docstring

```
def concatenator(string1, string2):
    """combines two strings, separated with a space

    Args:
        string1 (str): string to put before space
        string2 (str): string to put after space

    Returns:
        str: string1 and string2 separated by a space
    """
    return string1 + " " + string2
```

Use `'?` to retrieve the docstring, just like with built-in functions

```
concatenator?
```

```
Signature: concatenator(string1, string2)
Docstring:
combines two strings, separated with a space

Args:
    string1 (str): string to put before space
    string2 (str): string to put after space

Returns:
    str: string1 and string2 separated by a space
File:      /var/folders/f8/075hbnj13wb0f9yzh9k4nyz00000
gn/T/ipykernel_21060/3833548733.py
Type:     function
```



PRO TIP: Take time to create a docstring for your function, especially if you plan to share it with others.
What's the point in creating reusable code if you need to read all of it to understand how to use it?



ARGUMENT TYPES

There are several **types of arguments** that can be passed on to a function:

- **Positional** arguments are passed in the order they were defined in the function
- **Keyword** arguments are passed in any order by using the argument's name
- **Default** arguments pass a preset value if nothing is passed in the function call
- ***args** arguments pass any number of positional arguments as tuples
- ****kwargs** arguments pass any number of keyword arguments as dictionaries



ARGUMENT TYPES

Positional arguments are passed in the order they were defined in the function

```
def concatenator(string1, string2):  
    return string1 + ' ' + string2
```

```
concatenator('Hello', 'World!')
```

```
'Hello World!'
```

```
concatenator('World!', 'Hello')
```

```
'World! Hello'
```

The first value passed in the function will be string1, and the second will be string2

Therefore, changing the order of the inputs changes the output



ARGUMENT TYPES

Keyword arguments are passed in any order by using the argument's name

```
def concatenator(string1, string2):  
    return string1 + ' ' + string2
```

```
concatenator('Hello', 'World!')
```

```
'Hello World!'
```

```
concatenator(string2='World!', string1='Hello')
```

```
'Hello World!'
```

} By specifying the value to pass for each argument, the order no longer matters

```
concatenator(string2='World!', 'Hello')
```

```
SyntaxError: positional argument follows keyword argument
```

} Keyword arguments **cannot** be followed by positional arguments

```
concatenator('Hello', string2='World!')
```

```
'Hello World!'
```

} Positional arguments **can** be followed by keyword arguments (the first argument is typically reserved for primary input data)



ARGUMENT TYPES

Default arguments pass a preset value if nothing is passed in the function call

```
def concatenator(string1, string2='World!'):
    return string1 + ' ' + string2
```

```
concatenator('Hola')
```

```
'Hola World!'
```

```
concatenator('Hola', 'Mundo!')
```

```
'Hola Mundo!'
```

Assign a default value by using '=' when defining the function

Since a single argument was passed, the second argument defaults to 'World!'

By specifying a second argument, the default value is no longer used

```
def concatenator(string1='Hello', string2):
    return string1 + ' ' + string2
```

```
SyntaxError: non-default argument follows default argument
```

Default arguments must come after arguments without default values



ARGUMENT TYPES

***args** arguments pass any number of positional arguments as tuples

```
def concatenator(*args):
    new_string = ''
    for arg in args:
        new_string += (arg + ' ')
    return new_string.rstrip()
```

```
concatenator('Hello', 'world!', 'How', 'are', 'you?')
```

```
'Hello world! How are you?'
```

Using '*' before the argument name allows users to enter any number of strings for the function to concatenate

Since the arguments are passed as a tuple, we can loop through them or unpack them

```
def concatenator(*words):
    new_string = ''
    for word in words:
        new_string += (word + ' ')
    return new_string.rstrip()
```

```
concatenator('Hello', 'world!')
```

```
'Hello world!'
```

It's not necessary to use 'args' as long as the asterisk is there

Here we're using 'words' as the argument name, and only passing through two words

ARGUMENT TYPES

****kwargs** arguments pass any number of keyword arguments as dictionaries

```
def concatenator(**words):
    new_string = ''
    for word in words.values():
        new_string += (word + ' ')
    return new_string.rstrip()
```

```
concatenator(a='Hello', b ='there!',
             c="What's", d='up?')
```

"Hello there! What's up?"

Using `**` before the argument name allows users to enter any number of keyword arguments for the function to concatenate

Note that since the arguments are passed as dictionaries, you need to use the `.values()` method to loop through them



PRO TIP: Use `**kwargs` arguments to unpack dictionaries and pass them as keyword arguments

```
def exponentiator(constant, base, exponent):
    return constant * (base**exponent)
```

```
param_dict = {'constant': 2, 'base': 3, 'exponent': 2}
```

```
exponentiator(**param_dict)
```

The `exponentiator` function has three arguments: `constant`, `base`, and `exponent`

Note that the dictionary keys in `'param_dict'` match the argument names for the function

By using `**` to pass the dictionary to the function, the dictionary is unpacked, and the value for each key is mapped to the corresponding argument



RETURN VALUES

Functions can **return multiple values**

```
def concatenator(*words):
    sentence = ''
    for word in words:
        sentence += word + ' '
    last_word = words[-1]
    return sentence.rstrip(), last_word
```

```
concatenator('Hello', 'world!', 'How', 'are', 'you?')
```

```
('Hello world! How are you?', 'you?')
```

The values to return must be separated by commas

This returns a tuple of the specified values

```
sentence, last_word = concatenator('Hello', 'world!', 'How', 'are', 'you?')
print(sentence)
print(last_word)
```

```
Hello world! How are you?  
you?
```

The variable 'sentence' is assigned to the first element returned in the tuple, so if the order was switched, it would store 'you?'

You can unpack the tuple into variables during the function call



RETURN VALUES

Functions can return multiple values as **other types of iterables** as well

```
def concatenator(*words):
    sentence = ''
    for word in words:
        sentence += word + ' '
    last_word = words[-1]
    return [sentence.rstrip(), last_word]
```

```
concatenator('Hello', 'world!', 'How', 'are', 'you?')
```

```
['Hello world! How are you?', 'you?']
```

Wrap the comma-separated return values in square brackets to return them inside a list

```
def concatenator(*words):
    sentence = ''
    for word in words:
        sentence += word + ' '
    last_word = words[-1]
    return {sentence.rstrip(): last_word}
```

```
concatenator('Hello', 'world!', 'How', 'are', 'you?')
```

```
{'Hello world! How are you?': 'you?'}
```

*Or use dictionary notation to create a dictionary
(this could be useful as input for another function!)*



VARIABLE SCOPE

The **variable scope** is the region of the code where the variable was assigned

1. **Local scope** – variables created inside of a function

- *These cannot be referenced outside of the function*

2. **Global scope** – variables created outside of functions

- *These can be referenced both inside and outside of functions*

```
def concatenator(*words):
    global sentence
    sentence = ''
    for word in words:
        sentence += word + ' '
    last_word = words[-1]
    return sentence.rstrip(), last_word

concatenator('Hello', 'world!', 'How', 'are', 'you?')
print(sentence)

Hello world! How are you?
```

Since the variable 'sentence' is assigned inside of the concatenator function, it has local scope

Trying to print this variable outside of the function will then return a *NameError*



CHANGING VARIABLE SCOPE

You can **change variable scope** by using the *global* keyword

```
def concatenator(*words):
    global sentence
    sentence = ''
    for word in words:
        sentence += word + ' '
    last_word = words[-1]
    return sentence.rstrip(), last_word

concatenator('Hello', 'world!', 'How', 'are', 'you?')
print(sentence)
```

Hello world! How are you?

By declaring the variable 'sentence' as global, it is now recognized outside of the function it was defined in



CHANGING VARIABLE SCOPE

You can **change variable scope** by using the *global* keyword

```
def concatenator(*words):
    sentence = ''
    for word in words:
        sentence += word + ' '
    last_word = words[-1]
    global sentence
    return sentence.rstrip(), last_word

concatenator('Hello', 'world!', 'How', 'are', 'you?')
print(sentence)
```

SyntaxError: name 'sentence' is assigned to before global declaration

Note that the variable must be declared as *global* **before it is assigned a value**, or you will receive a *SyntaxError*



PRO TIP: While it might be tempting, declaring global variables within a function is considered bad practice in most cases – imagine if you borrowed this code and it overwrote an important variable! Instead, use ‘return’ to deliver the values you want and assign them to local variables

CREATING MODULES

To save your functions, **create a module** in Jupyter by using the `%%writefile` magic command and the .py extension

```
%%writefile saved_functions.py
def concatenator(*words):
    sentence = ''
    for word in words:
        sentence += word + ' '
    last_word = words[-1]
    return sentence.rstrip(), last_word
```

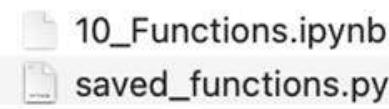
```
Writing saved_functions.py
%%writefile saved_functions.py
def concatenator(*words):
    sentence = ''
    for word in words:
        sentence += word + ' '
    last_word = words[-1]
    return sentence.rstrip(), last_word

def multiplier(num1, num2):
    return num1 * num2

Overwriting saved_functions.py
```

This creates a Python module that you can import functions from

- Follow `%%writefile` with the name of the file and the `.py` extension
- By default, the `.py` file is stored in the same folder as the notebook
- You can share functions easily by sending this file to a friend or colleague!



Multiple functions can be saved to the same module



IMPORTING MODULES

To **import saved functions**, you can either import the entire module or import specific functions from the module

```
import saved_functions
saved_functions.concatenate('Hello', 'world!')
('Hello world!', 'world!')
saved_functions.multiplier(5, 10)
50
```

import module

reads in external Python modules

If you import the entire module, you need to reference it when calling its functions, in the form of **module.function()**

```
from saved_functions import concatenate, multiplier
concatenate('Hello', 'world!')
('Hello world!', 'world!')
multiplier(5, 10)
```

from module import function

imports specific functions from modules

By importing specific functions, you don't need to reference the entire module name when calling a function



This method can lead to **naming conflicts** if another object has the same name

50



IMPORTING EXTERNAL FUNCTIONS

The same method used to import your own modules can be used to import external modules, packages, and libraries

- **Modules** are individual .py files
- **Packages** are collections of modules
- **Libraries** are collections of packages (*library and package are often used interchangeably*)

```
dir(saved_functions)
```

Use the dir() function to view the contents for any of the above

```
[ '__builtins__',  
  '__cached__',  
  '__doc__',  
  '__file__',  
  '__loader__',  
  '__name__',  
  '__package__',  
  '__spec__',  
  'concatenator',  
  'multiplier']
```

- This is showing the directory for the saved_functions module
- Modules have many attributes associated with them that the functions will follow
- '__file__' is the name of the .py file

These are the two functions inside the module



PRO TIP: NAMING CONFLICTS

Importing external functions can lead to **naming conflicts**

```
def sqrt(number):
    return f'The square root of {number} is its square root.'

sqrt(10)

'The square root of 10 is its square root.'
```

```
from math import sqrt

sqrt(10)

3.1622776601683795
```

In this case we have a `sqrt` function we defined ourselves, and another that we imported

In scenarios like these, only the **most recently created** object with a given name will be recognized

```
def sqrt(number):
    return f'The square root of {number} is its square root.'
```

```
import math as m

print(m.sqrt(10))
print(sqrt(10))

3.1622776601683795
The square root of 10 is its square root.
```

To avoid conflicts, refer to the package name or an alias with imported functions



INSTALLING PACKAGES

While many come preinstalled, you may need to **install a package** yourself

'! sends the following code to
your operating system's shell

(Yes, this means you can use the
command line via Jupyter)

!conda install openpyxl

The name of the
package to install

conda is a package
manager for Python

install tells Python to install
the following package



DO NOT install packages like this

It can lead to installing packages in the wrong instance of Python



```
import sys  
!conda install --yes --prefix {sys.prefix} openpyxl
```

Adding this extra code snippet ensures the
package gets installed correctly

PRO TIP: MANAGING PACKAGES

Use Anaconda to **review all installed packages** and their versions

```
!conda list
```

notebook	0.4.8	py39necasrcos_0
numpy	1.21.2	py39h4b4dc7a_0
numpy-base	1.21.2	py39he0bd621_0
openpyxl	3.0.9	pyhd3eb1b0_0

conda list returns a list of installed packages and versions

```
import sys
```

```
!conda install --yes --prefix {sys.prefix} openpyxl
```

```
!conda install --yes --prefix {sys.prefix} package_name=1.2.3
```

The **environments tab** in Anaconda Navigator also shows the installed packages and versions

Name	Description	Version
alabaster	Configurable, python 2+3 compatible sphinx theme.	0.7.12
anaconda	Simplifies package management and deployment of anaconda	2021.11
anaconda-client	Anaconda cloud command line client library	1.9.0
anaconda-project	Tool for encapsulating, running, and reproducing data science projects	0.10.1

conda install updates a package to its latest version; to install a specific version, use `package_name=<version>`

You may need to install an older version of a package so it is compatible with another package you are working with



ESSENTIAL PACKAGES FOR ANALYTICS

Python has incredible packages for **data analysis** beyond math and openpyxl

Here are some to explore further:





MAP

The **map()** function is an efficient way to apply a function to all items in an iterable

- `map(function, iterable)`

```
def currency_formatter(number):
    return '$' + str(number)
```

```
price_list = [5.99, 19.99, 24.99, 0, 74.99, 99.99]
```

```
map(currency_formatter, price_list)
```

```
<map at 0x7fa430944d90>
```

```
list(map(currency_formatter, price_list))
```

```
['$5.99', '$19.99', '$24.99', '$0', '$74.99', '$99.99']
```

The `map` function returns a **map object** - which saves memory until we've told Python what to do with the object

You can convert the `map` object into a list or other iterable



LAMBDA FUNCTIONS

Lambda functions are single line, anonymous functions that are only used briefly

- `lambda` arguments: expression

```
(lambda x: x**2) (3)
```

```
9
```

Lambda functions can be called on a single value, but typically aren't used for this

```
(lambda x: x**2, x**3)(3)
```

```
NameError
```

They cannot have multiple outputs or expressions

```
(lambda x, y: x * y if y > 5 else x / y)(6, 5)
```

```
1.2
```

They can take multiple arguments and leverage conditional logic

```
price_list = [5.99, 19.99, 24.99, 0, 74.99, 99.99]
exchange_rate = 0.88

converted = map(lambda x: round(x * exchange_rate, 2), price_list)
list(converted)

[5.27, 17.59, 21.99, 0.0, 65.99, 87.99]
```

They are usually leveraged in combination with a function like `map()` or in a comprehension.



PRO TIP: COMPREHENSIONS

Comprehensions can generate sequences from other sequences

Syntax: `new_list = [expression for member in other_iterable (if condition)]`

EXAMPLE

Creating a list of Euro prices from USD prices

Before, you needed a for loop to create the new list

```
usd_list = [5.99, 19.99, 24.99, 0, 74.99, 99.99]
exchange_rate = 0.88
euro_list = []

for price in usd_list:
    euro_list.append(round(price * exchange_rate, 2))

euro_list

[5.27, 17.59, 21.99, 0.0, 65.99, 87.99]
```

Now, you can use comprehensions to do this with a single line of code

```
exchange_rate = 0.88
usd_list = [5.99, 9.99, 19.99, 24.99, 99.99]

euro_list = [round(price * exchange_rate, 2) for price in usd_list]
euro_list

[5.27, 8.79, 17.59, 21.99, 87.99]
```

```
euro_list = [round(price * exchange_rate, 2) for price in usd_list if price < 10]
euro_list

[5.27, 0.0]
```

You can even leverage conditional logic to create very powerful expressions!



PRO TIP: DICTIONARY COMPREHENSIONS

Comprehensions can also **create dictionaries** from other iterables

Syntax: `new_dict = {key: value for key, value in other_iterable(if condition)}`

These can be expressions (including function calls!)

EXAMPLE

*Creating a dictionary of inventory costs per item (stock quantity * price)*

```
items = ['skis', 'snowboard', 'goggles', 'boots']
status = [[5, 249.99], [0, 219.99], [0, 99.99], [12, 79.99]]

inventory_costs = {k: round(v[0] * v[1], 2) for k, v in zip(items, status)}

inventory_costs
{'skis': 1249.95, 'snowboard': 0.0, 'goggles': 0.0, 'boots': 959.88}
```

This is creating a dictionary by using **items** as keys, and the product of **status[0]** and **status[1]** as values

`zip()` is being used to stitch the two lists together into a single iterable

```
inventory_costs = {
    k: round(v[0] * v[1], 2) for k, v in zip(items, status) if v[0] > 0
}

inventory_costs
{'skis': 1249.95, 'boots': 959.88}
```

You can still use conditional logic!



PRO TIP: COMPREHENSIONS VS MAP

Comprehensions, like map(), can apply functions to an entire sequence

```
def currency_converter(price, exchange_rate=.88):
    return round(float(price) * exchange_rate, 2)
```

Note that exchange_rate is a default argument equal to .88

```
price_list = [5.99, 19.99, 24.99, 0, 74.99, 99.99]
[currency_converter(price) for price in price_list]
[5.27, 17.59, 21.99, 0.0, 65.99, 87.99]
```

Here, in both the comprehension and map(), price_list is being passed as a positional argument to the currency_converter function, and the default exchange_rate value is applied

```
list(map(currency_converter, price_list))
[5.27, 17.59, 21.99, 0.0, 65.99, 87.99]
```

But what if I want to change the exchange rate?



PRO TIP: COMPREHENSIONS VS MAP

Comprehensions, like map(), can apply functions to an entire sequence

```
def currency_converter(price, exchange_rate=.88):
    return round(float(price) * exchange_rate, 2)
```

```
price_list = [5.99, 19.99, 24.99, 0, 74.99, 99.99]
[currency_converter(price, exchange_rate=.85) for price in price_list]
```

```
import functools
list(map(functools.partial(currency_converter, exchange_rate=.85), price_list))
[5.09, 16.99, 21.24, 0.0, 63.74, 84.99]
```

Note that exchange_rate is a default argument equal to .88

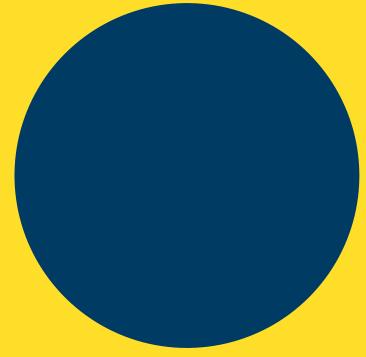
In the comprehension, the exchange_rate argument is easy to specify

To do so with map(), you need to create a partial function (outside the course scope)



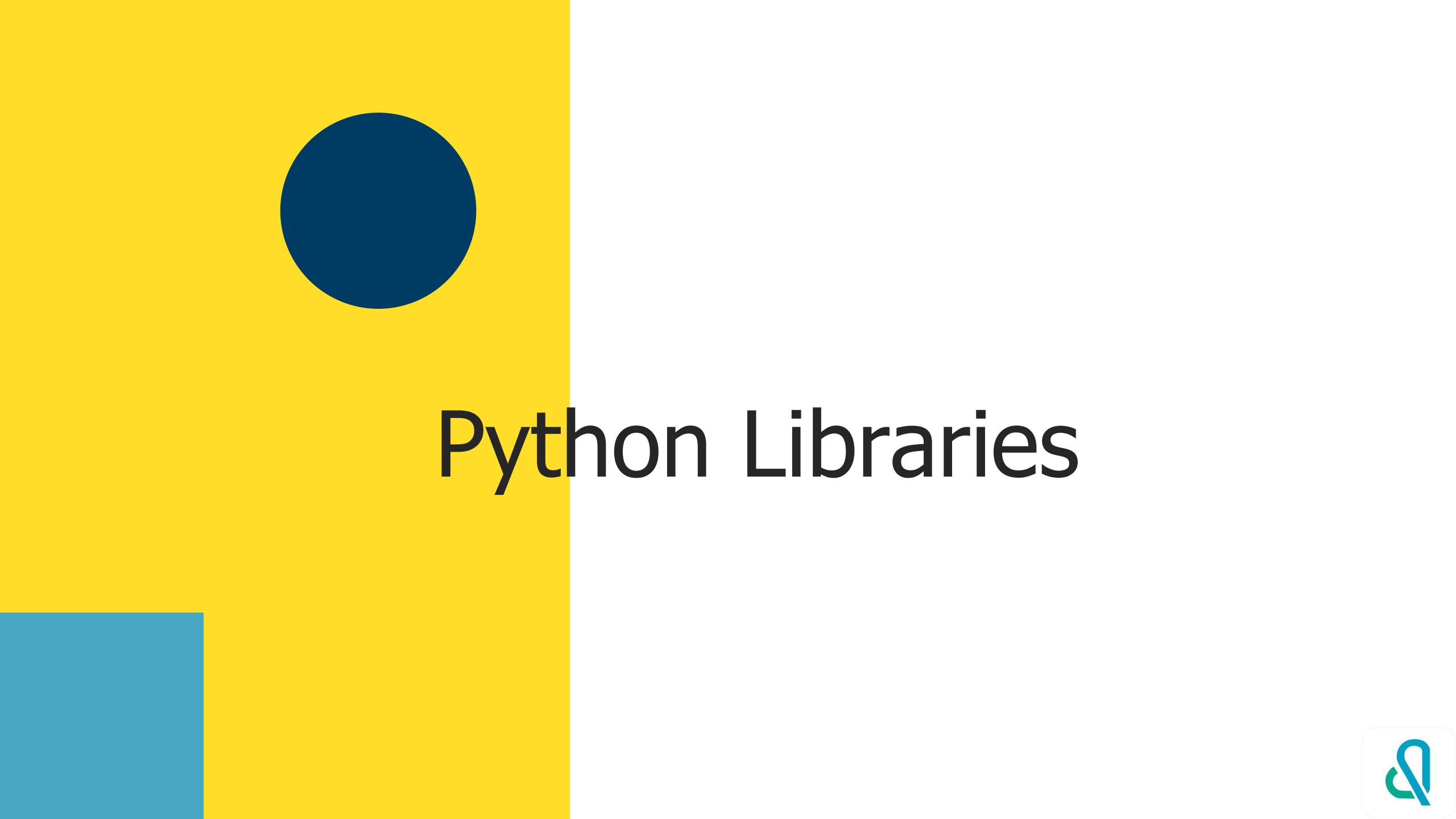
In general, both methods provide an efficient way to apply a function to a list – Comprehensions are usually preferred, as they are more efficient and highly readable, but there will be instances when using map with lambda is preferred (*like manipulating a Pandas Column*)

You may prefer not to use comprehensions for particularly complex logic (*nested loops, multiple sequences, lots of conditions*), but for most use cases comprehensions are a best practice for creating new sequences based off others.



Tambahan



The background of the slide features a large yellow square on the left side. In the top-left corner of this yellow area is a solid dark blue circle. At the bottom-left corner, there is a vertical teal bar.

Python Libraries



INTRO TO PANDAS & NUMPY

INTRO TO PANDAS & NUMPY



In this section we'll introduce **Pandas** & **NumPy**, two critical Python libraries that help structure data in arrays & DataFrames and contain built-in functions for data analysis

TOPICS WE'LL COVER:

[Intro to Pandas & NumPy](#)

[NumPy Array Basics](#)

[Array Creation](#)

[Array Indexing & Slicing](#)

[Array Operations](#)

[Vectorization & Broadcasting](#)

GOALS FOR THIS SECTION:

- Convert Python lists to NumPy arrays, and create new arrays from scratch using functions
- Apply array indexing, slicing, methods, and functions to perform operations on NumPy arrays
- Understand the concepts of vectorization and broadcasting, which are critical in making NumPy and Pandas more efficient than base Python



MEET PANDAS



Pandas is Python's most widely used library for data analysis, and contains functions for accessing, aggregating, joining, and analyzing data

Its data structure, the DataFrame, is analogous to SQL tables or Excel worksheets

Custom indices & column titles make working with datasets more intuitive!



	id	date	store_nbr	family	sales	onpromotion
0	0	2013-01-01	1	AUTOMOTIVE	0.0	0
1	1	2013-01-01	1	BABY CARE	0.0	0
2	2	2013-01-01	1	BEAUTY	0.0	0
3	3	2013-01-01	1	BEVERAGES	0.0	0
4	4	2013-01-01	1	BOOKS	0.0	0



MEET NUMPY



NumPy is an open-source library that is the universal standard for working with numerical data in Python, and forms the foundation of other libraries like Pandas. Pandas DataFrames are built on NumPy arrays and can leverage NumPy functions.

The indices, column names, and data columns are all stored as NumPy arrays

Pandas just adds convenient wrappers and functions!

	id	date	store_nbr	family	sales	onpromotion
0	0	2013-01-01	1	AUTOMOTIVE	0.0	0
1	1	2013-01-01	1	BABY CARE	0.0	0
2	2	2013-01-01	1	BEAUTY	0.0	0
3	3	2013-01-01	1	BEVERAGES	0.0	0
4	4	2013-01-01	1	BOOKS	0.0	0



NUMPY ARRAYS

NumPy arrays are fixed-size containers of items that are more efficient than Python lists or tuples for data processing

- They only store a single data type (*mixed data types are stored as a string*)
- They can be one dimensional or multi-dimensional
- Array elements can be modified, but the array size cannot change

```
import numpy as np  
  
sales = [0, 5, 155, 0, 518, 0, 1827, 616, 317, 325]  
  
sales_array = np.array(sales)  
sales_array  
  
array([ 0, 5, 155, 0, 518, 0, 1827, 616, 317, 325])
```

'np' is the standard alias for the NumPy library

NumPy's **array** function converts Python lists and tuples into NumPy arrays



ARRAY PROPERTIES

NumPy arrays have these key properties:

- **ndim** – the number of dimensions (axes) in the array
- **shape** – the size of the array for each dimension
- **size** – the total number of elements in the array
- **dtype** – the data type of the elements in the array

```
sales = [0, 5, 155, 0, 518, 0, 1827, 616, 317, 325]
sales_array = np.array(sales)
type(sales_array)
numpy.ndarray
```

```
print(f"ndim: {sales_array.ndim}")
print(f"shape: {sales_array.shape}")
print(f"size: {sales_array.size}")
print(f"dtype: {sales_array.dtype}")
```

ndim: 1 → The sales_array has 1 dimension
shape: (10,) → The dimension has a size of 10
size: 10 → The array has 10 elements total
dtype: int64 → The elements are stored as 64-bit integers

} NumPy arrays are a **ndarray** Python data type, which stands for n-dimensional array



ARRAY PROPERTIES

NumPy arrays have these key properties:

- **ndim** – the number of dimensions (axes) in the array
- **shape** – the size of the array for each dimension
- **size** – the total number of elements in the array
- **dtype** – the data type of the elements in the array

```
sales = [[0, 5, 155, 0, 518], [0, 1827, 616, 317, 325]]  
sales_array = np.array(sales)  
  
array([[ 0, 5, 155, 0, 518],  
       [0, 1827, 616, 317, 325]])
```

Converting a nested list creates a multi-dimensional array, where each nested list is a dimension

NOTE: The nested lists must be of equal length

```
print(f"ndim: {sales_array.ndim}")  
print(f"shape: {sales_array.shape}")  
print(f"size: {sales_array.size}")  
print(f"dtype: {sales_array.dtype}")
```

ndim: 2 → The sales_array has 2 dimensions
shape: (2, 5) → The first dimension has a size of 2 (rows) and the second a size of 5 (columns)
size: 10 → It has 10 elements total
dtype: int64 → The elements are stored as 64-bit integers



ARRAY CREATION

As an alternative to converting lists, you can **create arrays** using functions

ones

Creates an array of ones of a given size, as float by default

`np.ones((rows, cols), dtype)`

zeros

Creates an array of zeros of a given size, as float by default

`np.zeros((rows, cols), dtype)`

arange

Creates an array of integers with given start & stop values, and a step size (only stop is required, and is not inclusive)

`np.arange(start, stop, step)`

linspace

Creates an array of floats with given start & stop values with n elements, separated by a consistent step size (stop is inclusive)

`np.linspace(start, stop, n)`

reshape

Changes an array into the specified dimensions, if compatible

`np.array.reshape(rows, cols)`



ARRAY CREATION

As an alternative to converting lists, you can **create arrays** using functions

`np.ones((rows, cols), dtype)`

```
np.ones(4,)
```

```
array([1., 1., 1., 1.])
```

`np.zeros((rows, cols), dtype)`

```
np.zeros((2, 5), dtype=int)
```

```
array([[0, 0, 0, 0, 0],  
       [0, 0, 0, 0, 0]])
```

`np.arange(start, stop, step)`

```
np.arange(10)
```

```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

start is 0 and step is 1 by default

stop is not inclusive

`np.linspace(start, stop, n)`

```
np.linspace(0, 100, 5)
```

```
array([ 0., 25., 50., 75., 100.])
```

stop is inclusive

`np.array.reshape(rows, cols)`

```
np.arange(1, 9, 2).reshape(2, 2)
```

```
array([[1, 3],  
       [5, 7]])
```



RANDOM NUMBER ARRAYS

You can create **random number arrays** from a variety of distributions using NumPy functions and methods (great for sampling and simulation!)

default_rng

Creates a random number generator (the seed is for reproducibility)

random

Returns n random numbers from a uniform distribution between 0 and 1

normal

Returns n random numbers from a normal distribution with a given mean and standard deviation

np.default_rng(seed)

rng.random(n)

rng.normal(mean, stdev, n)

'rng' is the standard variable name for
the default_rng number generator



RANDOM NUMBER ARRAYS

You can create **random number arrays** from a variety of distributions using NumPy functions and methods (*great for sampling and simulation!*)

```
from numpy.random import default_rng  
  
rng = default_rng(12345)  
  
random_array = rng.random(10)  
random_array  
  
array([0.22733602, 0.31675834, 0.79736546, 0.67625467, 0.39110955,  
       0.33281393, 0.59830875, 0.18673419, 0.67275604, 0.94180287])
```

} First, we're creating a random number generator with a seed of 12345 and assigning it to 'rng' using **default_rng**

Then we're using the **random** method on 'rng' to return an array with 10 random numbers

```
rng = default_rng(12345)  
mean, stddev = 5, 1  
random_normal = rng.normal(mean, stddev, size=10)  
random_normal  
  
array([3.57617496, 6.26372846, 4.12933826, 4.74082677, 4.92465669,  
       4.25911535, 3.6322073 , 5.6488928 , 5.36105811, 3.04713694])
```

} Here we're using the **normal** method on 'rng' to return an array with 10 random numbers from a normal distribution with a mean of 5 and a st. deviation of 1



PRO TIP: Even though it's optional, make sure to **set a seed** when generating random numbers to ensure you and others can recreate the work you've done (the value for the seed is less important)



INDEXING & SLICING ARRAYS

Indexing & slicing one-dimensional arrays is the same as base Python

- `array[index]` – indexing to access a single element (0-indexed)
- `array[start:stop:step size]` – slicing to access a series of elements (`stop` is not inclusive)

```
product_array
```

```
array(['fruits', 'vegetables', 'cereal', 'dairy', 'eggs', 'snacks',
       'beverages', 'coffee', 'tea', 'spices'], dtype='<U10')
```

```
print(product_array[1])
print(product_array[-1])
```

```
vegetables
spices
```

```
product_array[:5]
```

```
array(['fruits', 'vegetables', 'cereal', 'dairy', 'eggs'], dtype='<U10')
```

```
product_array[5::2]
```

```
array(['snacks', 'coffee', 'spices'], dtype='<U10')
```

This grabs the **second** and **last** elements of `product_array`

This grabs the **first five** elements of `product_array`

This starts at the **sixth** element and grabs **every other** element **until the end** of `product_array`



INDEXING & SLICING ARRAYS

Indexing & slicing two-dimensional arrays requires an extra index or slice

- `array[row index, column index]` – indexing to access a single element (0-indexed)
- `array[start:stop:step size, start:stop:step size]` – slicing to access a series of elements

```
product_array2D = product_array.reshape(2, 5)
product_array2D

array([['fruits', 'vegetables', 'cereal', 'dairy', 'eggs'],
       ['snacks', 'beverages', 'coffee', 'tea', 'spices']], dtype='<U10')
```

```
product_array2D[1, 2]
'coffee'
```

This goes to the **second** row
and grabs the **third** element

```
product_array2D[:, 2:]
array([['cereal', 'dairy', 'eggs'],
       ['coffee', 'tea', 'spices']], dtype='<U10')
```

This goes to **all** rows and grabs
all the elements starting from
the **third** in each row

```
product_array2D[1:, :]
array([['snacks', 'beverages', 'coffee', 'tea', 'spices']], dtype='<U10')
```

This goes to the **second** row
and grabs **all** its elements



ARRAY OPERATIONS

Arithmetic operators can be used to perform **array operations**

```
sales = [[0, 5, 155, 0, 518], [0, 1827, 616, 317, 325]]  
sales_array = np.array(sales)  
sales_array
```

```
array([[ 0,    5,   155,    0,   518],  
       [ 0, 1827,   616,   317,   325]])
```

```
sales_array + 2
```

```
array([[ 2,    7,   157,    2,   520],  
       [ 2, 1829,   618,   319,   327]])
```

```
quantity = sales_array[0, :]  
price = sales_array[1, :]
```

```
quantity * price
```

```
array([ 0, 9135, 95480, 0, 168350])
```



Array operations are applied via **vectorization** and **broadcasting**, which eliminates the need to loop through the array's elements

This **adds 2** to every element in the array

This assigns **all** the elements in the **first row** to 'quantity'
Then assigns **all** the elements in the **second row** to 'price'
Finally, it multiplies the corresponding elements in each array:
(0*0, 5*1827, 155*616, 0*317, and 518*325)



FILTERING ARRAYS

You can **filter arrays** by indexing them with a logical test

- Only the array elements in positions where the logical test returns True are returned

```
sales_array
```

```
array([[ 0,  5, 155,  0, 518],  
       [ 0, 1827, 616, 317, 325]])
```

```
sales_array != 0
```

```
array([[False,  True,  True, False,  True],  
       [False,  True,  True,  True,  True]])
```

```
sales_array[sales_array != 0]
```

```
array([ 5, 155, 518, 1827, 616, 317, 325])
```

Performing a logical test on a NumPy array returns a **Boolean array** with the results of the logical test on each array element

Indexing an array with a Boolean array returns an array with the elements where the Boolean value is **True**



FILTERING ARRAYS

You can filter arrays with **multiple logical tests**

- Use `|` for **or** conditions and `&` for **and** conditions

```
sales_array
```

```
array([[ 0,    5,  155,    0,  518],  
       [ 0, 1827,   616,  317,  325]])
```

```
sales_array[(sales_array == 616) | (sales_array < 100)]
```

```
array([ 0,    5,    0,    0,  616])
```

```
sales_array[(sales_array > 100) & (sales_array < 500)]
```

```
array([155, 317, 325])
```

```
mask = (sales_array > 100) & (sales_array < 500)
```

```
sales_array[mask]
```

This returns an array with elements equal to 616 **or** less than 100

This returns an array with elements greater than 100 **and** less than 500



PRO TIP: Store complex filtering criteria in a variable (known as a Boolean mask)



FILTERING ARRAYS

You can filter arrays based on **values in other arrays**

- Use the Boolean array returned from the other array to index the array you want to filter

```
sales_array
```

```
array([ 0,  5, 155,  0, 518])
```

```
product_array
```

```
array(['fruits', 'vegetables', 'cereal', 'dairy', 'eggs'], dtype='<U10')
```

```
product_array[sales_array > 0]
```

```
array(['vegetables', 'cereal', 'eggs'], dtype='<U10')
```



This returns the elements from `product_array` where values in `sales_array` are greater than 0



MODIFYING ARRAY VALUES

You can **modify array values** by assigning new ones

```
sales_array
```

```
array([ 0,  5, 155,  0, 518])
```

```
sales_array[1] = 25
```

```
sales_array
```

```
array([ 0, 25, 155,  0, 518])
```

```
sales_array[sales_array == 0] = 5
```

```
sales_array
```

```
array([ 5, 25, 155,  5, 518])
```

This *assigns a single value via indexing*

This *filters the zero values in sales_array and assigns them a new value of 5*



THE WHERE FUNCTION

The **where()** NumPy function performs a logical test and returns a given value if the test is True, or another if the test is False

```
np.where(logical test,  
         value if True,  
         value if False)
```

Calls the NumPy
function library

A logical expression that
evaluates to True or False

Value to return when
the expression is True

Value to return when
the expression is False



THE WHERE FUNCTION

The **where()** NumPy function performs a logical test and returns a given value if the test is True, or another if the test is False

```
inventory_array
```

```
array([ 12, 102, 18, 0, 0])
```

```
product_array
```

```
array(['fruits', 'vegetables', 'cereal', 'dairy', 'eggs'], dtype='<U10')
```

```
np.where(inventory_array <= 0, "Out of Stock", "In Stock")
```

```
array(['In Stock', 'In Stock', 'In Stock', 'Out of Stock', 'Out of Stock'],  
      dtype='<U12')
```

```
np.where(inventory_array <= 0, "Out of Stock", product_array)
```

```
array(['fruits', 'vegetables', 'cereal', 'Out of Stock', 'Out of Stock'],  
      dtype='<U12')
```

If *inventory* is zero or negative,
assign 'Out of Stock', otherwise
assign 'In Stock'

If *inventory* is zero or negative,
assign 'Out of Stock', otherwise
assign the *product_array* value



ARRAY AGGREGATION METHODS

Array aggregation methods let you calculate metrics like sum, mean, and max

`sales_array`

```
array([[ 0,    5,  155,    0,  518],  
       [ 0, 1827,   616,   317,  325]])
```

array.sum() *Returns the sum of all values in an array*

`sales_array.sum()`

3763

array.mean() *Returns the average of the values in an array*

`sales_array.mean()`

376.3

array.max() *Returns the largest value in an array*

`sales_array.max()`

1827

array.min() *Returns the smallest value in an array*

`sales_array.min()`

0



ARRAY AGGREGATION METHODS

You can also aggregate across **rows** or **columns**

`sales_array`

```
array([[ 0,  5, 155,  0, 518],  
       [ 0, 1827, 616, 317, 325]])
```

`array.sum()` Returns the sum of all values in an array

`sales_array.sum()`

3763

`array.sum(axis=0)` Aggregates across rows

`sales_array.sum(axis=0)`

```
array([ 0, 1832, 771, 317, 84
```

`array.sum(axis=1)` Aggregates across columns

`sales_array.sum(axis=1)`

```
array([ 678, 3085])
```





ARRAY FUNCTIONS

Array functions let you perform other aggregations like median and percentiles

```
sales_array
```

```
array([[ 0,    5,  155,    0,  518],  
       [ 0, 1827,   616,   317,  325]])
```

np.median(array) Returns the median value in an array

```
np.median(sales_array)
```

236.0

np.percentile(array, n) Returns a value in the n^{th} percentile in an array

```
np.percentile(sales_array, 90)
```

737.0999999999996

← This uses linear interpolation by default



ARRAY FUNCTIONS

You can also return a **unique** list of values or the **square root** for each number

```
sales_array
```

```
array([[ 0,      5,    155,      0,    518],  
       [ 0,  1827,   616,   317,   325]])
```

np.unique(array) *Returns the unique values in an array*

```
np.unique(sales_array)
```

```
array([ 0,      5,    155,   317,   325,    518,    616,  1827])
```

np.sqrt(array) *Returns the square root of each value in an array*

```
np.sqrt(sales_array)
```

```
array([[ 0.          ,  2.23606798, 12.4498996 ,  0.          , 22.75961335],  
       [ 0.          , 42.74342055, 24.81934729, 17.80449381, 18.02775638]])
```



SORTING ARRAYS

The sort() method will **sort arrays** in place

- Use the axis argument to specify the dimension to sort by

```
sales_array
```

```
array([[ 0,  5, 155,  0, 518],  
       [ 0, 1827, 616, 317, 325]])
```

```
sales_array.sort()
```

```
sales_array
```

```
array([[ 0,  0,  5, 155, 518],  
       [ 0, 317, 325, 616, 1827]])
```

```
sales_array.sort(axis=0)
```

```
sales_array
```

```
array([[ 0,  5, 155,  0, 325],  
       [ 0, 1827, 616, 317, 518]])
```

axis=1 by default, which sorts a two-dimensional array row by row

axis=0 will sort by columns



VECTORIZATION

Vectorization is the process of pushing array operations into optimized C code, which is easier and more efficient than writing for loops

```
def for_loop_multiply_lists(list1, list2):
    product_list = []
    for element1, element2 in zip(tuple1, tuple2):
        product_list.append(element1 * element2)
    return product_list
```

```
def multiply_arrays(array1, array2):
    return array1 * array2
```

```
list1 = list(range(1000))
list2 = list(range(1000))
```

```
%%timeit -r 5 -n 10000
for_loop_multiply_lists(list1, list2)

75.8 µs ± 2 µs per loop (mean ± std. dev. of 5 runs, 10000 loops each)
```

```
array1 = np.array(list1)
array2 = np.array(list2)
```

```
%%timeit -r 5 -n 10000
multiply_arrays(array1, array2)

876 ns ± 44.7 ns per loop (mean ± std. dev. of 5 runs, 10000 loops each)
```

} Function that multiplies two Python **lists**

} Function that multiplies two NumPy **arrays**

} Generating and multiplying two lists

} Converting and multiplying two arrays
~86 times faster!



VECTORIZATION

Vectorization is the process of pushing array operations into optimized C code, which is easier and more efficient than writing for loops

```
def for_loop_multiply_lists(list1, list2):
    product_list = []
    for element1, element2 in zip(tuple1, tuple2):
        product_list.append(element1 * element2)
    return product_list

def multiply_arrays(array1, array2):
    return array1 * array2

list1 = list(range(1000))
list2 = list(range(1000))

%%timeit -r 5 -n 10000
for_loop_multiply_lists(list1, list2)

75.8 µs ± 2 µs per loop (mean ± std. dev. of 5 runs, 10000 loops each)

array1 = np.array(list1)
array2 = np.array(list2)

%%timeit -r 5 -n 10000
multiply_arrays(array1, array2)

876 ns ± 44.7 ns per loop (mean ± std. dev. of 5 runs, 10000 loops each)
```



PRO TIP: Use vectorized operations whenever possible when manipulating data, and avoid writing loops

Converting and multiplying two arrays
~86 times faster!



BROADCASTING

Broadcasting lets you perform vectorized operations with arrays of different sizes, where NumPy will expand the smaller array to 'fit' the larger one

- Single values (scalars) can be broadcast into arrays of any dimension

```
test_array = np.array([[1, 2, 3], [1, 2, 3], [1, 2, 3]])
```

```
test_array
```

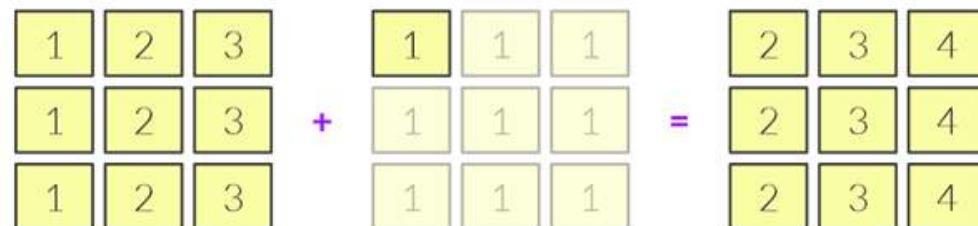
```
array([[1, 2, 3],  
       [1, 2, 3],  
       [1, 2, 3]])
```

```
test_array + 1
```

```
array([[2, 3, 4],  
       [2, 3, 4],  
       [2, 3, 4]])
```



How does this code work?





BROADCASTING

Broadcasting lets you perform vectorized operations with arrays of different sizes, where NumPy will expand the smaller array to ‘fit’ the larger one

- Single values (scalars) can be broadcast into arrays of any dimension
- Dimensions with a length greater than one must be the same size

```
test_array = np.array([[1, 2, 3], [1, 2, 3], [1, 2, 3]])
```

```
test_array
```

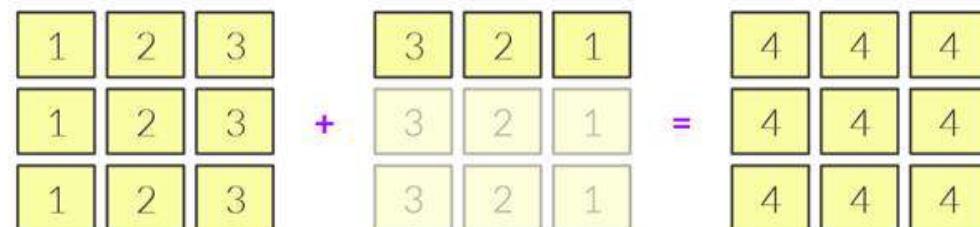
```
array([[1, 2, 3],  
       [1, 2, 3],  
       [1, 2, 3]])
```

```
test_array + np.array([3, 2, 1])
```

```
array([[4, 4, 4],  
       [4, 4, 4],  
       [4, 4, 4]])
```



How does this code work?





BROADCASTING

Broadcasting lets you perform vectorized operations with arrays of different sizes, where NumPy will expand the smaller array to 'fit' the larger one

- Single values (scalars) can be broadcast into arrays of any dimension
- Dimensions with a length greater than one must be the same size

```
test_array = np.array([[1, 2, 3], [1, 2, 3], [1, 2, 3]])
```

```
test_array
```

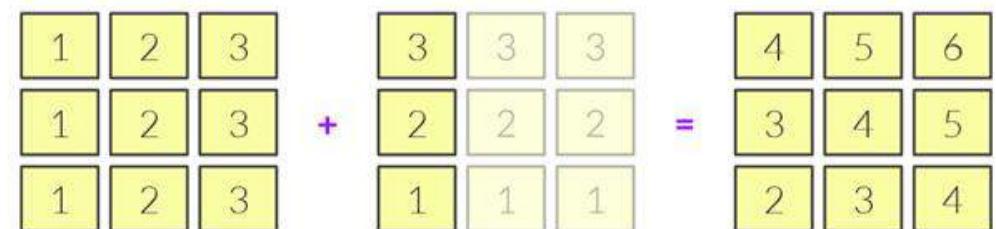
```
array([[1, 2, 3],  
       [1, 2, 3],  
       [1, 2, 3]])
```

```
test_array + np.array([3, 2, 1]).reshape(3, 1)
```

```
array([[4, 5, 6],  
       [3, 4, 5],  
       [2, 3, 4]])
```



How does this code work?





BROADCASTING

Broadcasting lets you perform vectorized operations with arrays of different sizes, where NumPy will expand the smaller array to 'fit' the larger one

- Single values (scalars) can be broadcast into arrays of any dimension
- Dimensions with a length greater than one must be the same size

```
test_array = np.array([[1, 2, 3], [1, 2, 3], [1, 2, 3]])
```

```
test_array
```

```
array([[1, 2, 3],  
       [1, 2, 3],  
       [1, 2, 3]])
```

```
test_array + np.array([2, 1])
```

```
ValueError: operands could not be broadcast  
together with shapes (3,3) (2,)
```



How does this code work?

$$\begin{matrix} 1 & 2 & 3 \\ 1 & 2 & 3 \\ 1 & 2 & 3 \end{matrix} + \begin{matrix} 2 & 1 & ? \\ 2 & 1 & ? \\ 2 & 1 & ? \end{matrix} = \text{!}$$



BROADCASTING

Broadcasting lets you perform vectorized operations with arrays of different sizes, where NumPy will expand the smaller array to 'fit' the larger one

- Single values (scalars) can be broadcast into arrays of any dimension
- Dimensions with a length greater than one must be the same size

```
test_array = np.array([[1, 2, 3], [1, 2, 3], [1, 2, 3]])
```

```
test_array
```

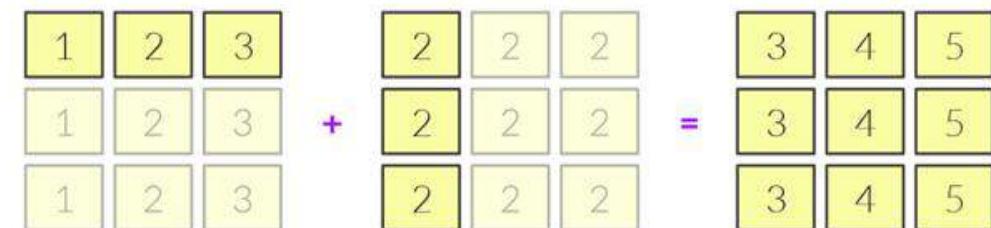
```
array([[1, 2, 3],  
       [1, 2, 3],  
       [1, 2, 3]])
```

```
test_array[0, :] + test_array[:, 1].reshape(3, 1)
```

```
array([[3, 4, 5],  
       [3, 4, 5],  
       [3, 4, 5]])
```



How does this code work?





BROADCASTING

Broadcasting lets you perform vectorized operations with arrays of different sizes, where NumPy will expand the smaller array to 'fit' the larger one

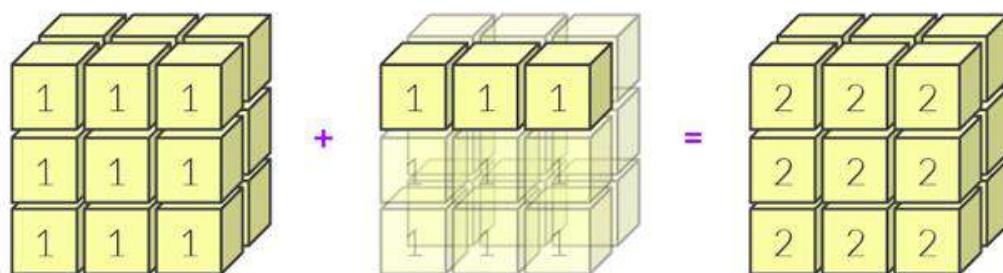
- Single values (scalars) can be broadcast into arrays of any dimension
- Dimensions with a length greater than one must be the same size

```
np.ones((2, 3, 3), dtype=int) + np.ones(3, dtype=int)
```

```
array([[[2, 2, 2],  
       [2, 2, 2],  
       [2, 2, 2]],  
  
      [[2, 2, 2],  
       [2, 2, 2],  
       [2, 2, 2]])
```



How does this code work?





BROADCASTING

Broadcasting lets you perform vectorized operations with arrays of different sizes, where NumPy will expand the smaller array to 'fit' the larger one

- Single values (scalars) can be broadcast into arrays of any dimension
- Dimensions with a length greater than one must be the same size



Compatible shapes:

Array 1	Array 2	Output
(3, 3)	(100, 3, 3)	(100, 3, 3)
(3, 1, 5, 1)	(4, 1, 6)	(3, 4, 5, 6)



Incompatible shapes:

Array 1	Array 2
(4, 3)	(100, 3, 3)
(3, 1, 5, 1)	(4, 2, 6)

PANDAS SERIES

SERIES



In this section we'll introduce Pandas **Series**, the Python equivalent of a column of data, and cover their basic properties, creation, manipulation, and useful functions for analysis

TOPICS WE'LL COVER:

Pandas Series Basics

Series Indexing

Sorting & Filtering

Operations & Aggregations

Handling Missing Data

Applying Custom Functions

GOALS FOR THIS SECTION:

- Understand the relationship between Pandas Series and NumPy arrays
- Use the `.loc()` and `.iloc()` methods to access Series data by their indices or values
- Learn to sort, filter, and aggregate Pandas Series using methods and functions
- Apply custom functions using conditional logic to Pandas Series



PANDAS SERIES

Series are Pandas data structures built on top of NumPy arrays

- Series also contain an **index** and an **optional name**, in addition to the array of data
- They can be created from other data types, but are usually imported from external sources
- Two or more Series grouped together form a Pandas DataFrame

```
import numpy as np
import pandas as pd
sales = [0, 5, 155, 0, 518, 0, 1827, 616, 317, 325]
sales_series = pd.Series(sales, name="Sales")
sales_series
```

The index is an array of integers starting at 0 by default, but it can be modified

0	0
1	5
2	155
3	0
4	518
5	0
6	1827
7	616
8	317
9	325

Name: Sales, dtype: int64

'pd' is the standard alias for the Pandas library

Pandas' Series function converts Python lists and NumPy arrays into Pandas Series

The name argument lets you specify a name

The series name and data type are stored as well



SERIES PROPERTIES

Pandas Series have these key properties:

- **values** – the data array in the Series
- **index** – the index array in the Series
- **name** – the optional name for the Series (*useful for accessing columns in a DataFrame*)
- **dtype** – the data type of the elements in the values array

```
sales_series.values
```

The index is a range of integers from 0 to 10

```
array([ 0, 5, 155, 0, 518, 0, 1827, 616, 317, 325])
```

```
sales_series.index
```

```
RangeIndex(start=0, stop=10, step=1)
```

```
sales_series.name
```

```
'Sales'
```

```
sales_series.dtype
```

```
dtype('int64')
```



PANDAS DATA TYPES

Pandas data types mostly expand on their base Python and NumPy equivalents

Numeric:

Data Type	Description	Bit Sizes
bool	Boolean True/False	8
int64 (default)	Whole numbers	8, 16, 32, 64
float64 (default)	Decimal numbers	8, 16, 32, 64
boolean	Nullable Boolean True/False	8
Int64 (default)	Nullable whole numbers	8, 16, 32, 64
Float64 (default)	Nullable decimal numbers	32, 64

*Gray = NumPy data type

*Yellow = Pandas data type



We'll review the nuances of data types
in depth when covering **DataFrames**

Object / Text:

Data Type	Description
object	Any Python object
string	Only contains strings or text
category	Maps categorical data to a numeric array for efficiency

Time Series:

Data Type	Description
datetime64	A single moment in time (January 4, 2015, 2:00:00 PM)
timedelta64	The duration between two dates or times (10 days, 3 seconds, etc.)
period	A span of time (a day, a week, etc.)



TYPE CONVERSION

You can **convert the data type** in a Pandas Series by using the `.astype()` method and specifying the desired data type (*if compatible*)

`sales_series`

```
0      0  
1      5  
2    155      These are integers  
3      0  
4    518  
Name: Sales, dtype: int64
```

`sales_series.astype("float")`

```
0      0.0  
1      5.0  
2    155.0      This converts them to floats  
3      0.0  
4    518.0  
Name: Sales, dtype: float64
```

`sales_series.astype("bool")`

```
0    False  
1    True      This converts them to Booleans  
2    True      (0 is False, others are True)  
3   False  
4    True  
Name: Sales, dtype: bool
```

`sales_series.astype("datetime64")`

This attempts to convert them to the Datetime datatype, but isn't compatible

```
ValueError: The 'datetime64' dtype has no unit.
```



THE INDEX

The **index** lets you easily access “rows” in a Pandas Series or DataFrame

```
sales = [0, 5, 155, 0, 518]
sales_series = pd.Series(sales, name="Sales")
sales_series
```

Here we're using the
default integer index,
which is preferred

```
{ 0      0
 1      5
 2    155
 3      0
 4    518
Name: Sales, dtype: int64}
```

```
sales_series[2]
```

```
155
```

```
sales_series[2:4]
```

```
2    155
3      0
```

```
Name: Sales, dtype: int64
```

You can **index** and **slice** Series like
other sequence data types, but we'll
learn a better method



CUSTOM INDICES

There are cases where it's applicable to use a **custom index** for accessing rows

```
sales = [0, 5, 155, 0, 518]
items = ["coffee", "bananas", "tea", "coconut", "sugar"]

sales_series = pd.Series(sales, index=items, name="Sales")
```

```
sales_series
coffee      0
bananas     5
tea       155
coconut     0
sugar      518
Name: Sales, dtype: int64
```

*Custom indices can be assigned when
creating the series or by assignment*

```
sales_series.index = ["coffee", "bananas", "tea", "coconut", "sugar"]
```

```
sales_series
coffee      0
bananas     5
tea       155
coconut     0
sugar      518
Name: Sales, dtype: int64
```



This will become more relevant
when working with **datetimes**
(covered later in the course!)



CUSTOM INDICES

There are cases where it's applicable to use a **custom index** for accessing rows

```
sales = [0, 5, 155, 0, 518]
items = ["coffee", "bananas", "tea", "coconut", "sugar"]

sales_series = pd.Series(sales, index=items, name="Sales")

sales_series
```

```
coffee      0
bananas     5
tea        155
coconut     0
sugar       518
Name: Sales, dtype: int64
```

```
sales_series["tea"]
```



```
155
```

```
sales_series["bananas":"coconut"]
```



```
bananas     5
tea        155
coconut     0
Name: Sales, dtype: int64
```

Note that slicing custom indices
makes the stop point **inclusive**

You can still **index** and **slice**
to retrieve Series values using
the custom indices



THE ILOC METHOD

The `.iloc[]` method is the preferred way to access values by their positional index

- This method works even when Series have a custom, non-integer index
- It is more efficient than slicing and is recommended by Pandas' creators

`df.iloc[row position, column position]`

Series or DataFrame
to access values from

The row position(s) for the
value(s) you want to access

The column position(s) for the
value(s) you want to access

Examples:

- 0 (single row)
- [5, 9] (multiple rows)
- [0:11] (range of rows)



We'll use the column position
argument once we start working
with Pandas **DataFrames**



THE ILOC METHOD

The `.iloc[]` method is the preferred way to access values by their positional index

- This method works even on Series with a custom, non-integer index
- It is more efficient than slicing and is recommended by Pandas' creators

Note that this Series has a custom index

sales_series	
coffee	0
bananas	5
tea	155
coconut	0
sugar	518
Name: Sales, dtype: int64	

`sales_series.iloc[2]`

155

`sales_series.iloc[2:4]`

tea 155
coconut 0

Name: Sales, dtype: int64

This returns the value in the 3rd position (0-indexed), even though the custom index for that value is "tea"

This returns the values from the 3rd to the 4th position (stop is non-inclusive)



THE LOC METHOD

The `.loc[]` method is the preferred way to access values by their custom labels

```
df.loc[row label, column label]
```

Series or DataFrame
to access values from

The custom row index for the
value(s) you want to access

The custom column index for
the value(s) you want to access

Examples:

- "pizza" (single row)
- ["mike", "ike"] (multiple rows)
- ["jan":"dec"] (range of rows)



THE LOC METHOD

The `.loc[]` method is the preferred way to access values by their custom labels

*The custom indices
are the labels*

```
sales_series
coffee      0
bananas     5
tea        155
coconut      0
sugar       518
Name: Sales, dtype: int64
```

```
sales_series.loc["tea"]
```

155

```
sales_series.loc["bananas":"coconut"]
```

```
bananas      5
tea        155
coconut      0
Name: Sales, dtype: int64
```



The `.loc[]` method works even when the indices are integers, but if they are custom integers not ordered from 0 to n-1, the rows will be returned based on the **labels** themselves and NOT their numeric position

Note that slices are **inclusive** when using custom labels



DUPLICATE INDEX VALUES

It is possible to have **duplicate index values** in a Pandas Series or DataFrame

- Accessing these indices by their label using `.loc[]` returns all corresponding rows

```
sales = [0, 5, 155, 0, 518]
items = ["coffee", "coffee", "tea", "coconut", "sugar"]

sales_series = pd.Series(sales, index=items, name="Sales")
```

```
sales_series
```

```
coffee    0
coffee    5
tea      155
coconut   0
sugar    518
Name: Sales, dtype: int64
```

Note that 'coffee' is used as an index value twice

```
sales_series.loc["coffee"]
```

```
coffee    0
coffee    5
Name: Sales, dtype: int64
```

This returns both rows with the same label



Warning! Duplicate index values are generally not advised, but there are some edge cases where they are useful



RESETTING THE INDEX

You can **reset the index** in a Pandas Series or DataFrame back to the default range of integers by using the `.reset_index()` method

- By default, the existing index will become a new column in a DataFrame

`sales_series`

```
coffee      0  
coffee      5  
tea       155  
coconut     0  
sugar      518  
Name: Sales, dtype: int64
```

`sales_series.reset_index()`

	index	Sales
0	coffee	0
1	coffee	5
2	tea	155
3	coconut	0
4	sugar	518

This returns a DataFrame by default, with the previous index values stored as a new column

`sales_series.reset_index(drop=True)`

```
0      0  
1      5  
2    155  
3      0  
4    518  
Name: Sales, dtype: int64
```

Use `drop=True` when resetting the index if you don't want the previous index values stored



FILTERING SERIES

You can **filter a Series** by passing a logical test into the `.loc[]` accessor (like arrays!)

```
sales_series
```

```
coffee      0
coffee      5
tea     155
coconut     0
sugar     518
Name: Sales, dtype: int64
```

```
sales_series.loc[sales_series > 0]
```

```
coffee      5
tea     155
sugar     518
Name: Sales, dtype: int64
```

```
mask = (sales_series > 0) & (sales_series.index == "coffee")
```

```
sales_series.loc[mask]
```

```
coffee      5
Name: Sales, dtype: int64
```

This returns all rows from `sales_series` with a value greater than 0

This uses a **mask** to store complex logic and returns all rows from `sales_series` with a greater than 0 and an index equal to "coffee"



LOGICAL OPERATORS & METHODS

You can use these **operators** & **methods** to create Boolean filters for logical tests

Description	Python Operator	Pandas Method
Equal	==	.eq()
Not Equal	!=	.ne()
Less Than or Equal	<=	.le()
Less Than	<	.lt()
Greater Than or Equal	>=	.ge()
Greater Than	>	.gt()
Membership Test	in	.isin()
Inverse Membership Test	not in	~.isin()

```
sales_series  
coffee      0  
bananas    5  
tea       155  
coconut     0  
sugar      518  
Name: Sales, dtype: int64
```

Python Operator:

```
sales_series == 5
```

```
coffee  False  
coffee  True  
tea    False  
coconut False  
sugar  False  
Name: Sales, dtype: bool
```

Pandas Method:

```
sales_series.eq(5)
```

```
coffee  False  
coffee  True  
tea    False  
coconut False  
sugar  False  
Name: Sales, dtype: bool
```



LOGICAL OPERATORS & METHODS

You can use these **operators** & **methods** to create Boolean filters for logical tests

Description	Python Operator	Pandas Method
Equal	==	.eq()
Not Equal	!=	.ne()
Less Than or Equal	<=	.le()
Less Than	<	.lt()
Greater Than or Equal	>=	.ge()
Greater Than	>	.gt()
Membership Test	in	.isin()
Inverse Membership Test	not in	~.isin()

The Python operators 'in' and 'not in' won't work for many operations, so the Pandas method must be used

`sales_series`

```
coffee      0
bananas     5
tea       155
coconut     0
sugar      518
Name: Sales, dtype: int64
```

```
sales_series.index.isin(["coffee", "tea"])
```

```
array([ True,  True,  True, False, False])
```

```
~sales_series.index.isin(["coffee", "tea"])
```

```
array([False, False, False,  True,  True])
```

The tilde '`~`' inverts Boolean values!



SORTING SERIES

You can **sort Series** by their values or their index

1. The `.sort_values()` method sorts a Series by its values in ascending order

```
sales_series.sort_values()
```

```
coffee      0  
coconut    0  
coffee      5  
tea        155  
sugar      518  
Name: Sales, dtype: int64
```

```
sales_series.sort_values(ascending=False)
```

```
sugar      518  
tea        155  
coffee      5  
coffee      0  
coconut    0  
Name: Sales, dtype: int64
```

Specify `ascending=False`
to sort in descending order

2. The `.sort_index()` method sorts a Series by its index in ascending order

```
sales_series.sort_index()
```

```
coconut    0  
coffee     0  
coffee     5  
sugar     518  
tea       155  
Name: Sales, dtype: int64
```

```
sales_series.sort_index(ascending=False)
```

```
tea        155  
sugar     518  
coffee     0  
coffee     5  
coconut    0  
Name: Sales, dtype: int64
```



ARITHMETIC OPERATORS & METHODS

You can use these **operators** & **methods** to perform numeric operations on Series

Operation	Python Operator	Pandas Method
Addition	+	.add()
Subtraction	-	.sub(), .subtract()
Multiplication	*	.mul(), .multiply()
Division	/	.div(), .truediv(), .divide()
Floor Division	//	.floordiv()
Modulo	%	.mod()
Exponentiation	**	.pow()

monday_sales	monday_sales + 2	monday_sales.add(2)
0 0	0 2	0 2
1 5	1 7	1 7
2 155	2 157	2 157
3 0	3 2	3 2
4 518	4 520	4 520
dtype: int64		dtype: int64

These both add two to every row

	"\$" + monday_sales.astype("float").astype("string")
0	\$0.0
1	\$5.0
2	\$155.0
3	\$0.0
4	\$518.0
dtype: string	

This uses string arithmetic to add a dollar sign,
converts to float to add decimals (cents), then
converts back to a string



STRING METHODS

The Pandas str accessor lets you access many **string methods**

- These methods all return a Series (*split* returns multiple series)

String Method	Description
.strip(), .lstrip(), .rstrip()	Removes all leading and/or trailing characters (spaces by default)
.upper(), .lower()	Converts all characters to upper or lower case
.slice(start:stop:step)	Applies a slice to the strings in a Series
.count("string")	Counts all instances of a given string
.contains("string")	Returns True if a given string is found; False if not
.replace("a", "b")	Replaces instances of string "a" with string "b"
.split("delimiter", expand=True)	Splits strings based on a given delimiter string, and returns a DataFrame with a Series for each split
.len()	Returns the length of each string in a Series
.startswith("string"), .endswith("string")	Returns True if a string starts or ends with given string; False if not

```
prices
```

```
0    $3.99
1    $5.99
2   $22.99
3    $7.99
4   $33.99
dtype: object
```

```
prices.str.contains("3")
```

```
0    True
1   False
2   False
3   False
4    True
dtype: bool
```

```
clean = prices.str.strip("$").astype("float")
```

```
clean
```

```
0    3.99
1    5.99
2   22.99
3    7.99
4   33.99
dtype: float64
```

The **str** accessor lets you access the string methods

This is removing the dollar sign, then converting to float



NUMERIC SERIES AGGREGATION

You can use these methods to **aggregate numerical Series**

Method	Description
.count()	Returns the number of items
.first(), .last()	Returns the first or last item
.mean(), .median()	Calculates the mean or median
.min(), .max()	Returns the smallest or largest value
.argmax(), .argmin()	Returns the index for the smallest or largest values
.std(), .var()	Calculates the standard deviation or variance
.mad()	Calculates the mean absolute deviation
.prod()	Calculates the product of all the items
.sum()	Calculates the sum of all the items
.quantile()	Returns a specified percentile, or list of percentiles

```
sales_series
```

```
coffee      0.0
coffee      5.0
tea        155.0
coconut     NaN
sugar       518.0
Name: Sales, dtype: float64
```

```
sales_series.sum()
```

```
678.0
```

```
sales_series.loc["coffee"].sum()
```

```
5.0
```

```
sales_series.quantile([0.25, 0.50, 0.75])
```

```
0.25      3.75
0.50     80.00
0.75    245.75
Name: Sales, dtype: float64
```



CATEGORICAL SERIES AGGREGATION

You can use these methods to **aggregate categorical Series**

Method	Description
.unique()	Returns an array of unique items in a Series
.nunique()	Returns the number of unique items
.value_counts()	Returns a Series of unique items and their frequency

```
items  
0    coffee  
1    coffee  
2      tea  
3   coconut  
4     sugar  
dtype: object
```

```
items.value_counts()  
coffee    2  
tea      1  
coconut  1  
sugar    1  
dtype: int64
```

```
items.value_counts(normalize=True)  
coffee    0.4  
tea      0.2  
coconut  0.2  
sugar    0.2  
dtype: float64
```

Specify `normalize=True` to return the percentage of total for each category

```
items.unique()
```

```
array(['coffee', 'tea', 'coconut', 'sugar'], dtype=object)
```

4

```
items.nunique()
```



MISSING DATA

Missing data in Pandas is often represented by NumPy “NaN” values

- This is more efficient than Python’s “None” data type
- Pandas treats NaN values as a float, which allows them to be used in vectorized operations

```
sales = [0, 5, 155, np.nan, 518]
sales_series = pd.Series(sales, name="Sales")
sales_series
```

```
0      0.0
1      5.0
2    155.0
3      NaN
4    518.0
Name: Sales, dtype: float64
```

np.nan creates a NaN value

*These are rarely created by hand,
and typically appear when reading
in data from external sources*

*If NaN was not present here,
the data type would be int64*

```
sales_series + 2
```

```
0      2.0
1      7.0
2    157.0
3      NaN
4    520.0
Name: Sales, dtype: float64
```

*Arithmetic operations performed
on NaN values will return NaN*

```
sales_series.add(2, fill_value=0)
```

```
0      2.0
1      7.0
2    157.0
3      2.0
4    520.0
Name: Sales, dtype: float64
```

*Most operation methods include a
'fill_value' argument that lets you
pass a value instead of NaN*



MISSING DATA

Pandas released its own **missing data type**, NA, in December 2020

- This allows missing values to be stored as integers, instead of needing to convert to float
- This is still a new feature, but most bugs end up converting the data to NumPy's NaN

```
sales = [0, 5, 155, pd.NA, 518]
sales_series = pd.Series(sales, name="Sales", dtype="Int16")
sales_series
```

0 0
1 5
2 155
3 <NA>
4 518
Name: Sales, dtype: Int16

} **pd.NA** creates an NA value

Note that if **dtype="Int16"** wasn't specified, the values would be stored as objects



At this time, **neither np.NaN nor pd.NA are perfect**, but pd.NA functionality should continue to improve, and having a nullable integer is usually worth it (more on that in the next section!)



IDENTIFYING MISSING DATA

The `.isna()` and `.value_counts()` methods let you **identify missing data** in a Series

- The `.isna()` method returns True if a value is missing, and False otherwise

checklist	checklist.isna()	checklist.isna().sum()	.isna().sum() returns the count of NaN values
0 COMPLETE 1 NaN 2 NaN 3 NaN 4 COMPLETE dtype: object	0 False 1 True 2 True 3 True 4 False dtype: bool	3	

You can use this as a Boolean mask!

- The `.value_counts()` method returns unique values and their frequency

checklist.value_counts()	checklist.value_counts(dropna=False)	Most methods ignore NaN values, so you need to specify <code>dropna=False</code> to return the count of NaN values
COMPLETE 2 dtype: int64	NaN 3 COMPLETE 2 dtype: int64	



HANDLING MISSING DATA

The `.dropna()` and `.fillna()` methods let you **handle missing data** in a Series

- The `.dropna()` method removes NaN values from your Series or DataFrame

checklist		checklist.dropna()	Note that the index has gaps, so you can use <code>.reset_index()</code> to restore the range of integers
0	COMPLETE	0	
1	NaN	4	COMPLETE
2	NaN		<code>dtype: object</code>
3	NaN		
4	COMPLETE		
	<code>dtype: object</code>		

- The `.fillna(value)` method replaces NaN values with a specified value

checklist		checklist.fillna("INCOMPLETE")	
0	COMPLETE	0	COMPLETE
1	NaN	1	INCOMPLETE
2	NaN	2	INCOMPLETE
3	NaN	3	INCOMPLETE
4	COMPLETE	4	COMPLETE
	<code>dtype: object</code>	<code>dtype: object</code>	



THE APPLY METHOD

The `.apply()` method lets you apply custom functions to Pandas Series

- This function will not be vectorized, so it's not as efficient as native functions

```
def discount(price):
    if price > 20:
        return round(price * 0.9, 2)
    return price
```

This function applies a 10% discount to prices over 20

```
clean_wholesale
```

```
0    3.99
1    5.99
2   22.99
3    7.99
4   33.99
dtype: float64
```

```
clean_wholesale.apply(discount)
```

```
0    3.99
1    5.99
2  20.69
3    7.99
4  30.59
dtype: float64
```

Discount applied!

```
clean_wholesale.apply(lambda x: round(x * 0.9, 2) if x > 20 else x)
```

```
0    3.99
1    5.99
2  20.69
3    7.99
4  30.59
dtype: float64
```

*You can also use **Lambda** functions for one-off tasks!*



THE WHERE METHOD

Pandas' **.where()** method lets you manipulate data based on a logical condition

```
df.where(logical test,  
         value if False,  
         inplace=False)
```

Series or DataFrame
to evaluate data from

Whether to perform
the operation in place
(default is False)

A logical expression that
evaluates to True or False

Value to return when
the expression is False



Heads up! This is different
from NumPy's where function



THE WHERE METHOD

Pandas' **.where()** method lets you manipulate data based on a logical condition

```
clean_wholesale
```

```
0    3.99  
1    5.99  
2   22.99  
3    7.99  
4   33.99  
dtype: float64
```

```
clean_wholesale.where(clean_wholesale <= 20, round(clean_wholesale * 0.9, 2))
```

```
0    3.99  
1    5.99  
2    20.69  
3    7.99  
4    30.59  
dtype: float64
```



This expression returns *False* if the price is greater than 20, and the value if *false* statement for the discount is applied



THE WHERE METHOD

Pandas' **.where()** method lets you manipulate data based on a logical condition

```
clean_wholesale
```

```
0    3.99  
1    5.99  
2   22.99  
3    7.99  
4   33.99  
dtype: float64
```

```
clean_wholesale.where(~(clean_wholesale > 20), round(clean_wholesale * 0.9, 2))
```

```
0    3.99  
1    5.99  
2   20.69  
3    7.99  
4   30.59  
dtype: float64
```



You can use a tilde '`~`' to invert the Boolean values and turn this into a "value if True"



CHAINING WHERE

You can **chain .where() methods** to combine logical expressions

```
clean_wholesale
```

```
0    3.99  
1    5.99  
2   22.99  
3    7.99  
4   33.99  
dtype: float64
```

```
(clean_wholesale  
    .where(~(clean_wholesale > 20), round(clean_wholesale * 0.9, 2))  
    .where(clean_wholesale > 10, 0)  
)
```

```
0    0.00  
1    0.00  
2   20.69  
3    0.00  
4   30.59  
dtype: float64
```

The first where method applies a 90% discount if a price is greater than 20

The second applies a value of 0 when a price is NOT greater than 10



NUMPY VS. PANDAS WHERE

NumPy's where function is often more convenient & useful than Pandas' method

```
clean_wholesale
```

```
0      3.99
1      5.99
2     22.99
3      7.99
4     33.99
dtype: float64
```

```
np.where(clean_wholesale > 20, "Discounted", "Normal Price")
```

```
array(['Normal Price', 'Normal Price', 'Discounted', 'Normal Price',
       'Discounted'], dtype='<U12')
```



*Note that this returns a NumPy array that
you'd need to convert into a Pandas Series*

DATAFRAMES



THE PANDAS DATAFRAME

DataFrames are Pandas “tables” made up from columns and rows

- Each column of data in a DataFrame is a Pandas Series that shares the same row index
- The column headers work as a column index that contains the Series names

The row index points to the corresponding row in each Series
(axis = 0)

The column index points to each individual Series
(axis = 1)

Each column is a Pandas Series

id	date	store_nbr	family	sales	onpromotion
0	0 2013-01-01	1	AUTOMOTIVE	0.0	0
1	1 2013-01-01	1	BABY CARE	0.0	0
2	2 2013-01-01	1	BEAUTY	0.0	0
3	3 2013-01-01	1	BEVERAGES	0.0	0
4	4 2013-01-01	1	BOOKS	0.0	0



DATAFRAME PROPERTIES

DataFrames have these key properties:

- **shape** – the number of rows and columns in a DataFrame (*the index is not considered a column*)
- **index** – the row index in a DataFrame, represented as a range of integers (*axis=0*)
- **columns** – the column index in a DataFrame, represented by the Series names (*axis=1*)
- **axes** – the row and column indices in a DataFrame
- **dtypes** – the data type for each Series in a DataFrame (*they can be different!*)

```
df.shape
```

```
(3000888, 6)
```

```
df.columns
```

```
Index(['id', 'date', 'store_nbr', 'family', 'sales', 'onpromotion'], dtype='object')
```

```
df.axes
```

```
[RangeIndex(start=0, stop=3000888, step=1),  
 Index(['id', 'date', 'store_nbr', 'family', 'sales', 'onpromotion'], dtype='object')]
```

```
df.index
```

```
RangeIndex(start=0, stop=3000888, step=1)
```

```
df.dtypes
```

id	int64
date	object
store_nbr	int64
family	object
sales	float64
onpromotion	int64
dtype:	object

Pandas will try to guess the data types when creating a DataFrame (we'll modify them later!)



CREATING A DATAFRAME

You can **create a DataFrame** from a Python dictionary or NumPy array by using the Pandas DataFrame() function

```
pd.DataFrame(  
    {"id": [1, 2],  
     "store_nbr": [1, 2],  
     "family": ["POULTRY", "PRODUCE"]  
    }  
)
```

This creates a DataFrame from a Python dictionary
Note that the keys are used as column names

	id	store_nbr	family
0	1	1	POULTRY
1	2	2	PRODUCE



CREATING A DATAFRAME

You'll more likely **create a DataFrame** by reading in a flat file (csv, txt, or tsv) with Pandas `read_csv()` function

```
import pandas as pd  
  
retail_df = pd.read_csv("retail/train.csv")  
  
retail_df
```

	id	date	store_nbr	family	sales	onpromotion
0	0	2013-01-01	1	AUTOMOTIVE	0.000	0
1	1	2013-01-01	1	BABY CARE	0.000	0
2	2	2013-01-01	1	BEAUTY	0.000	0
3	3	2013-01-01	1	BEVERAGES	0.000	0
4	4	2013-01-01	1	BOOKS	0.000	0
...
3000883	3000883	2017-08-15	9	POULTRY	438.133	0
3000884	3000884	2017-08-15	9	PREPARED FOODS	154.553	1
3000885	3000885	2017-08-15	9	PRODUCE	2419.729	148
3000886	3000886	2017-08-15	9	SCHOOL AND OFFICE SUPPLIES	121.000	8
3000887	3000887	2017-08-15	9	SEAFOOD	16.000	0

3000888 rows x 6 columns

While the `read_csv()` function has more arguments for manipulating the data during the import process, all you need is the file path and name to get started!



PRO TIP: Pandas is a great alternative to Excel when dealing with large datasets!



Alert

This data set is too large for the Excel grid. If you save this workbook, you'll lose data that wasn't loaded.



EXPLORING A DATAFRAME

You can **explore a DataFrame** with these Pandas methods:

head

Returns the first *n* rows of the DataFrame (5 by default)

`df.head(nrows)`

tail

Returns the last *n* rows of the DataFrame (5 by default)

`df.tail(nrows)`

sample

Returns *n* rows from a random sample (1 by default)

`df.sample(n)rows`

info

Returns key details on a DataFrame's size, columns, and memory usage

`df.info()`

describe

Returns descriptive statistics for the columns in a DataFrame (only numeric columns by default; use the 'include' argument to specify more columns)

`df.describe(include)`



HEAD & TAIL

The `.head()` and `.tail()` methods return the top or bottom rows in a DataFrame

- This is a great way to QA data upon import!

```
retail_df.head()
```

	id	date	store_nbr	family	sales	onpromotion
0	0	2013-01-01	1	AUTOMOTIVE	0.0	0
1	1	2013-01-01	1	BABY CARE	0.0	0
2	2	2013-01-01	1	BEAUTY	0.0	0
3	3	2013-01-01	1	BEVERAGES	0.0	0
4	4	2013-01-01	1	BOOKS	0.0	0

This returns the top 5 rows by default

```
retail_df.tail(3)
```

	id	date	store_nbr	family	sales	onpromotion
3000885	3000885	2017-08-15	9	PRODUCE	2419.729	148
3000886	3000886	2017-08-15	9	SCHOOL AND OFFICE SUPPLIES	121.0	8
3000887	3000887	2017-08-15	9	SEAFOOD	16.0	0

You can specify the number of rows to return, in this case the bottom 3



SAMPLE

The `.sample()` method returns a random sample of rows from a DataFrame

```
retail_df.sample()
```

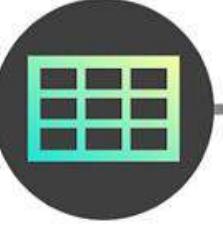
	id	date	store_nbr	family	sales	onpromotion	
	1476628	1476628	2015-04-11	40	EGGS	77.0	0

} This returns 1 row by default

```
retail_df.sample(5, random_state=12345)
```

	id	date	store_nbr	family	sales	onpromotion	
	2862475	2862475	2017-05-30	25	LIQUOR,WINE,BEER	92.0	2
	940501	940501	2014-06-13	48	BABY CARE	0.0	0
	1457967	1457967	2015-04-01	17	PLAYERS AND ELECTRONICS	0.0	0
	1903307	1903307	2015-12-07	12	SEAFOOD	3.0	0
	196280	196280	2013-04-21	16	PREPARED FOODS	66.0	0

} You can specify the number of rows to return, in this case 5, and set a random_state to ensure your sample can be reproduced later if needed



INFO

The `.info()` method returns details on a DataFrame's properties and memory usage

```
retail_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 3000888 entries, 0 to 3000887
Data columns (total 6 columns):
 #   Column      Dtype  
 0   id          int64  
 1   date         object 
 2   store_nbr    int64  
 3   family       object 
 4   sales        float64
 5   onpromotion  int64  
dtypes: float64(1), int64(3), object(2)
memory usage: 137.4+ MB
```

Rows & columns in the DataFrame {
Position, name, and data type for each column {
Memory usage {

INFO



The `.info()` method returns details on a DataFrame's properties and memory usage

```
retail_df.info(show_counts=True)
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 3000888 entries, 0 to 3000887
Data columns (total 6 columns):
 #   Column      Non-Null Count   Dtype  
---  --          -----           ----- 
 0   id          3000888 non-null    int64  
 1   date         3000888 non-null    object  
 2   store_nbr    3000888 non-null    int64  
 3   family        3000888 non-null    object  
 4   sales         3000888 non-null    float64 
 5   onpromotion   3000888 non-null    int64  
dtypes: float64(1), int64(3), object(2)
memory usage: 137.4+ MB
```

The `.info()` method will show non-null counts on a DataFrame with less than ~1.7m rows, but you can specify `show_counts=True` to ensure they are always displayed

This is a great way to quickly identify missing values - if the non-null count is less than the total number of rows, then the difference is the number of NaN values in that column!

(In this case there are none)



DESCRIBE

The `.describe()` method returns key statistics on a DataFrame's columns

```
retail_df.describe()
```

	id	store_nbr	sales	onpromotion
Non-null values	count	3.000888e+06	3.000888e+06	3.000888e+06
Mean & standard deviation values	mean	1.500444e+06	2.750000e+01	3.577757e+02
	std	8.662819e+05	1.558579e+01	1.101998e+03
Quartile values	min	0.000000e+00	1.000000e+00	0.000000e+00
	25%	7.502218e+05	1.400000e+01	0.000000e+00
	50%	1.500444e+06	2.750000e+01	1.100000e+01
	75%	2.250665e+06	4.100000e+01	1.958473e+02
	max	3.000887e+06	5.400000e+01	1.247170e+05

Only numeric columns by default

DESCRIBE



The `.describe()` method returns key statistics on a DataFrame's columns

```
retail_df.describe(include="all").round()
```

Unique values,
most common
value (top), and
its frequency

	id	date	store_nbr	family	sales	onpromotion
count	3000888.0	3000888	3000888.0	3000888	3000888.0	3000888.0
unique	NaN	1684	NaN	33	NaN	NaN
top	NaN	2013-01-01	NaN	AUTOMOTIVE	NaN	NaN
freq	NaN	1782	NaN	90936	NaN	NaN
mean	1500444.0	NaN	28.0	NaN	358.0	3.0
std	866282.0	NaN	16.0	NaN	1102.0	12.0
min	0.0	NaN	1.0	NaN	0.0	0.0
25%	750222.0	NaN	14.0	NaN	0.0	0.0
50%	1500444.0	NaN	28.0	NaN	11.0	0.0
75%	2250665.0	NaN	41.0	NaN	196.0	0.0
max	3000887.0	NaN	54.0	NaN	124717.0	741.0

Use `include="all"` to return statistics for all columns, or choose a specific data type to include

Note that the `.round()` method suppresses scientific notation and makes the output more readable



ACCESING DATAFRAME COLUMNS

You can **access a DataFrame column** by using bracket or dot notation

- Dot notation only works for valid Python variable names (no spaces, special characters, etc.), and if the column name is not the same as an existing variable or method

```
retail_df['family']
```

```
0          AUTOMOTIVE
1        BABY CARE
2         BEAUTY
3      BEVERAGES
4         BOOKS
...
3000883      POULTRY
3000884 PREPARED FOODS
3000885      PRODUCE
3000886 SCHOOL AND OFFICE SUPPLIES
3000887      SEAFOOD
Name: family, Length: 3000888, dtype: object
```

```
retail_df.family
```

```
0          AUTOMOTIVE
1        BABY CARE
2         BEAUTY
3      BEVERAGES
4         BOOKS
...
3000883      POULTRY
3000884 PREPARED FOODS
3000885      PRODUCE
3000886 SCHOOL AND OFFICE SUPPLIES
3000887      SEAFOOD
Name: family, Length: 3000888, dtype: object
```



PRO TIP: Even though you'll see many examples of dot notation in use, stick to bracket notation for single columns of data as it is less likely to cause issues



ACCESSING DATAFRAME COLUMNS

You can **use Series operations** on DataFrame columns (each column is a Series!)

Number of unique values in a column

```
retail_df["family"].nunique()
```

33

Mean of values in a column

```
retail_df["sales"].mean()
```

357.77574911262707

First 5 unique values in a column with their frequencies

```
retail_df["family"].value_counts().iloc[:5]
```

AUTOMOTIVE	90936
HOME APPLIANCES	90936
SCHOOL AND OFFICE SUPPLIES	90936
PRODUCE	90936
PREPARED FOODS	90936

Name: family, dtype: int64

Rounded sum of values in a column

```
retail_df["sales"].sum().round()
```

1073644952.0



ACCESSING DATAFRAME COLUMNS

You can **select multiple columns** with a list of column names between brackets

- This is ideal for selecting non-consecutive columns in a DataFrame

```
retail_df[['family', 'store_nbr']]
```

	family	store_nbr
0	AUTOMOTIVE	1
1	BABY CARE	1
2	BEAUTY	1
3	BEVERAGES	1
4	BOOKS	1
...
3000883	POULTRY	9
3000884	PREPARED FOODS	9
3000885	PRODUCE	9
3000886	SCHOOL AND OFFICE SUPPLIES	9
3000887	SEAFOOD	9

3000888 rows × 2 columns

```
retail_df[["family", "store_nbr"]].iloc[:5]
```

	family	store_nbr
0	AUTOMOTIVE	1
1	BABY CARE	1
2	BEAUTY	1
3	BEVERAGES	1
4	BOOKS	1



PRO TIP: Use .loc() to access more than one column of data – column bracket notation should primarily be used for creating new columns and quick exploration



ACCESING DATA WITH ILOC

The `.iloc()` accessor filters DataFrames by their row and column indices

- The first parameter accesses rows, and the second accesses columns

First 5 rows,
all columns

```
retail_df.iloc[:5, :]
```

	id	date	store_nbr	family	sales	onpromotion
0	0	2013-01-01	1	AUTOMOTIVE	0.0	0
1	1	2013-01-01	1	BABY CARE	0.0	0
2	2	2013-01-01	1	BEAUTY	0.0	0
3	3	2013-01-01	1	BEVERAGES	0.0	0
4	4	2013-01-01	1	BOOKS	0.0	0



ACCESING DATA WITH ILOC

The `.iloc()` accessor filters DataFrames by their row and column indices

- The first parameter accesses rows, and the second accesses columns

All rows,
2-4 columns

```
retail_df.iloc[:, 1:4]
```

	date	store_nbr	family
0	2013-01-01	1	AUTOMOTIVE
1	2013-01-01	1	BABY CARE
2	2013-01-01	1	BEAUTY
3	2013-01-01	1	BEVERAGES
4	2013-01-01	1	BOOKS
...
3000883	2017-08-15	9	POULTRY
3000884	2017-08-15	9	PREPARED FOODS
3000885	2017-08-15	9	PRODUCE
3000886	2017-08-15	9	SCHOOL AND OFFICE SUPPLIES
3000887	2017-08-15	9	SEAFOOD



ACCESING DATA WITH ILOC

The `.iloc()` accessor filters DataFrames by their row and column indices

- The first parameter accesses rows, and the second accesses columns

First 5 rows,
2-4 columns

```
retail_df.iloc[:5, 1:4]
```

	date	store_nbr	family
0	2013-01-01	1	AUTOMOTIVE
1	2013-01-01	1	BABY CARE
2	2013-01-01	1	BEAUTY
3	2013-01-01	1	BEVERAGES
4	2013-01-01	1	BOOKS



ACCESING DATA WITH LOC

The `.loc()` accessor filters DataFrames by their row and column labels

- The first parameter accesses rows, and the second accesses columns

All rows,
"date" column

`retail_df.loc[:, "date"]`

```
0      2013-01-01  
1      2013-01-01  
2      2013-01-01  
3      2013-01-01  
4      2013-01-01  
      ...  
3000883 2017-08-15  
3000884 2017-08-15  
3000885 2017-08-15  
3000886 2017-08-15  
3000887 2017-08-15  
Name: date, Length: 3000888
```

This is a Series

`retail_df.loc[:, ["date"]]`

date
0 2013-01-01
1 2013-01-01
2 2013-01-01
3 2013-01-01
4 2013-01-01
...
3000883 2017-08-15
3000884 2017-08-15
3000885 2017-08-15
3000886 2017-08-15
3000887 2017-08-15

Wrap single columns
in brackets to return
a DataFrame

3000888 rows x 1 columns



ACCESING DATA WITH LOC

The `.loc()` accessor filters DataFrames by their row and column labels

- The first parameter accesses rows, and the second accesses columns

All rows,
"date" & "sales"
columns
(list of columns)

`retail_df.loc[:, ["date", "sales"]]`

	date	sales
0	2013-01-01	0.000
1	2013-01-01	0.000
2	2013-01-01	0.000
3	2013-01-01	0.000
4	2013-01-01	0.000
...
3000883	2017-08-15	438.133
3000884	2017-08-15	154.553
3000885	2017-08-15	2419.729
3000886	2017-08-15	121.000
3000887	2017-08-15	16.000
3000888 rows × 2 columns		

All rows,
"date" through
"sales" columns
(slice of columns)

`retail_df.loc[:, "date":"sales"]`

	date	store_nbr	family	sales
0	2013-01-01	1	AUTOMOTIVE	0.0
1	2013-01-01	1	BABY CARE	0.0
2	2013-01-01	1	BEAUTY	0.0
3	2013-01-01	1	BEVERAGES	0.0
4	2013-01-01	1	BOOKS	0.0
...
3000883	2017-08-15	9	POULTRY	438.133
3000884	2017-08-15	9	PREPARED FOODS	154.553
3000885	2017-08-15	9	PRODUCE	2419.729
3000886	2017-08-15	9	SCHOOL AND OFFICE SUPPLIES	121.0
3000887	2017-08-15	9	SEAFOOD	16.0
3000888 rows × 4 columns				



DROPPING ROWS & COLUMNS

The `.drop()` method **drops rows and columns** from a DataFrame

- Specify axis=0 to drop rows by label, and axis=1 to drop columns

```
retail_df.drop("id", axis=1).head()
```

	date	store_nbr	family	sales	onpromotion
0	2013-01-01	1	AUTOMOTIVE	0.0	0
1	2013-01-01	1	BABY CARE	0.0	0
2	2013-01-01	1	BEAUTY	0.0	0
3	2013-01-01	1	BEVERAGES	0.0	0
4	2013-01-01	1	BOOKS	0.0	0

This returns the first 5 rows of the `retail_df` DataFrame without the "id" column

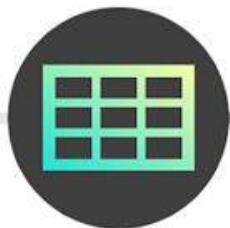
```
retail_df.drop(["id", "onpromotion"], inplace=True, axis=1)  
retail_df.head()
```

	date	store_nbr	family	sales
0	2013-01-01	1	AUTOMOTIVE	0.0
1	2013-01-01	1	BABY CARE	0.0
2	2013-01-01	1	BEAUTY	0.0
3	2013-01-01	1	BEVERAGES	0.0
4	2013-01-01	1	BOOKS	0.0

You can specify `inplace=True` to permanently remove rows or columns from a DataFrame



PRO TIP: Drop unnecessary columns early in your workflow to save memory and make DataFrames more manageable (ideally, they shouldn't be imported - more on that later!)



DROPPING ROWS & COLUMNS

The `.drop()` method **drops rows and columns** from a DataFrame

- Specify axis=0 to drop rows by label, and axis=1 to drop columns

```
retail_df.drop([0], axis=0).head()
```

	date	store_nbr	family	sales
1	2013-01-01	1	BABY CARE	0.0
2	2013-01-01	1	BEAUTY	0.0
3	2013-01-01	1	BEVERAGES	0.0
4	2013-01-01	1	BOOKS	0.0
5	2013-01-01	1	BREAD/BAKERY	0.0

This returns the first 5 rows of the `retail_df` DataFrame after removing the first row

Note that the row label is passed as a list

```
retail_df.drop(range(5), axis=0).head()
```

	date	store_nbr	family	sales
5	2013-01-01	1	BREAD/BAKERY	0.0
6	2013-01-01	1	CELEBRATION	0.0
7	2013-01-01	1	CLEANING	0.0
8	2013-01-01	1	DAIRY	0.0
9	2013-01-01	1	DELI	0.0

You can pass a range to remove rows with consecutive labels, in this case 0-4



You'll typically drop rows via **slicing** or **filtering**, but it's worth being aware that `.drop()` can be used as well



IDENTIFYING DUPLICATE ROWS

The `.duplicated()` method **identifies duplicate rows** of data

- Specify `subset=column(s)` to look for duplicates across a subset of columns

product_df		
	product	price
0	Dairy	2.56
1	Dairy	2.56
2	Dairy	4.55
3	Vegetables	2.74
4	Fruits	5.44

product_df.shape		
(5,	2)	
product_df.nunique()		
product	3	
price	4	
		dtype: int64

If the number of unique values for a column is less than the total number of rows, then that column contains duplicate values



IDENTIFYING DUPLICATE ROWS

The `.duplicated()` method **identifies duplicate rows** of data

- Specify subset=column(s) to look for duplicates across a subset of columns

product_df

	product	price
0	Dairy	2.56
1	Dairy	2.56
2	Dairy	4.55
3	Vegetables	2.74
4	Fruits	5.44

product_df.duplicated()

```
0  False  
1  True  
2  False  
3  False  
4  False  
dtype: bool
```

The `.duplicated()` method returns `True` for the second row here because it is a duplicate of the first row



IDENTIFYING DUPLICATE ROWS

The `.duplicated()` method **identifies duplicate rows** of data

- Specify subset=column(s) to look for duplicates across a subset of columns

product_df

	product	price
0	Dairy	2.56
1	Dairy	2.56
2	Dairy	4.55
3	Vegetables	2.74
4	Fruits	5.44

product_df.duplicated(subset='product')

```
0  False  
1  True  
2  True  
3  False  
4  False  
dtype: bool
```

Specifying `subset='product'` will only look for duplicates in that column

In this case rows 2 and 3 are duplicates of the first row ("Dairy")



DROPPING DUPLICATE ROWS

The `.drop_duplicates()` method **drops duplicate rows** from a DataFrame

- Specify `subset=column(s)` to look for duplicates across a subset of columns

`product_df`

	product	price
0	Dairy	2.56
1	Dairy	2.56
2	Dairy	4.55
3	Vegetables	2.74
4	Fruits	5.44

`product_df.drop_duplicates()`

	product	price
0	Dairy	2.56
2	Dairy	4.55
3	Vegetables	2.74
4	Fruits	5.44



This removed the second row from the `product_df` DataFrame, as it is a duplicate of the first row

Note that the row index now has a gap between 0 & 2



DROPPING DUPLICATE ROWS

The `.drop_duplicates()` method **drops duplicate rows** from a DataFrame

- Specify `subset=column(s)` to look for duplicates across a subset of columns

product_df

	product	price
0	Dairy	2.56
1	Dairy	2.56
2	Dairy	4.55
3	Vegetables	2.74
4	Fruits	5.44

product_df.drop_duplicates(subset="product", keep="last", ignore_index=True)

	product	price
0	Dairy	4.55
1	Vegetables	2.74
2	Fruits	5.44



How does this code work?

- `subset="product"` will look for duplicates in the product column (*index 0, 1, and 2 for "Dairy"*)
- `keep="last"` will keep the final duplicate row, and drop the rest
- `ignore_index=True` will reset the index so there are no gaps



IDENTIFYING MISSING DATA

You can **identify missing data** by column using the `.isna()` and `.sum()` methods

- The `.info()` method can also help identify null values

`product_df`

	product	price	product_id
0	<NA>	2.56	1
1	Dairy	<NA>	2
2	Dairy	4.55	3
3	<NA>	2.74	4
4	Fruits	<NA>	5

`product_df.isna().sum()`

```
product      2
price        2
product_id   0
dtype: int64
```

This is a Series

`product_df.info()`

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 5 entries, 0 to 4
Data columns (total 3 columns):
 #   Column       Non-Null Count   Dtype  
 ---  --          --          --      
 0   product     3 non-null       object 
 1   price       3 non-null       object 
 2   product_id  5 non-null       int64  
dtypes: int64(1), object(2)
memory usage: 248.0+ bytes
```

The `.isna()` method returns a DataFrame with Boolean values (True for NAs, False for others)

The `.sum()` method adds these for each column (True=1, False=0) and returns the summarized results

The difference between the total entries and non-null values for each column gives you the missing values in each



HANDLING MISSING DATA

Like with Series, the `.dropna()` and `.fillna()` methods let you **handle missing data** in a DataFrame by either removing them or replacing them with other values

`product_df`

	product	price
0	<NA>	2.56
1	Dairy	<NA>
2	Dairy	4.55
3	<NA>	2.74
4	Fruits	<NA>

`product_df.fillna(0)`

	product	price
0	0	2.56
1	Dairy	0.00
2	Dairy	4.55
3	0	2.74
4	Fruits	0.00

`product_df.fillna({"price": 0})`

	product	price
0	<NA>	2.56
1	Dairy	0.00
2	Dairy	4.55
3	<NA>	2.74
4	Fruits	0.00

Use a dictionary
to specify a value
for each column

`product_df.dropna()`

	product	price
2	Dairy	4.55

This drops any row
with missing values

`product_df.dropna(subset=["price"])`

	product	price
0	<NA>	2.56
2	Dairy	4.55
3	<NA>	2.74

Use subset to drop rows
with missing values in
specified columns



FILTERING DATAFRAMES

You can **filter the rows in a DataFrame** by passing a logical test into the .loc[] accessor, just like filtering a Series or a NumPy array

```
retail_df.loc[retail_df["date"] == "2016-10-28"]
```

		id	date	store_nbr	family	sales	onpromotion
2482326	2482326	2482326	2016-10-28	1	AUTOMOTIVE	8.000	0
2482327	2482327	2482327	2016-10-28	1	BABY CARE	0.000	0
2482328	2482328	2482328	2016-10-28	1	BEAUTY	9.000	1
2482329	2482329	2482329	2016-10-28	1	BEVERAGES	2576.000	38
2482330	2482330	2482330	2016-10-28	1	BOOKS	0.000	0
...
2484103	2484103	2484103	2016-10-28	9	POULTRY	391.292	24
2484104	2484104	2484104	2016-10-28	9	PREPARED FOODS	78.769	1
2484105	2484105	2484105	2016-10-28	9	PRODUCE	993.760	5
2484106	2484106	2484106	2016-10-28	9	SCHOOL AND OFFICE SUPPLIES	0.000	0
2484107	2484107	2484107	2016-10-28	9	SEAFOOD	3.000	1

1782 rows × 6 columns

This filters the retail_df DataFrame and only returns rows where the date is equal to "2016-10-28"



FILTERING DATAFRAMES

You can **filter the columns in a DataFrame** by passing them into the `.loc[]` accessor as a list or a slice

```
retail_df.loc[retail_df["date"] == "2016-10-28", ["date", "sales"]].head()
```

	date	sales
2482326	2016-10-28	8.0
2482327	2016-10-28	0.0
2482328	2016-10-28	9.0
2482329	2016-10-28	2576.0
2482330	2016-10-28	0.0

Row filter

Column filter

This filters the `retail_df` DataFrame to the columns selected, and only returns rows where the date is equal to "2016-10-28"



FILTERING DATAFRAMES

You can **apply multiple filters** by joining the logical tests with an “&” operator

- Try creating a Boolean mask for creating filters with complex logic

```
conds = retail_df["family"].isin(["CLEANING", "DAIRY"]) & (retail_df["sales"] > 0)  
retail_df.loc[conds]
```

	id	date	store_nbr	family	sales	onpromotion
568	568	2013-01-01	25	CLEANING	186.0	0
569	569	2013-01-01	25	DAIRY	143.0	0
1789	1789	2013-01-02	1	CLEANING	1060.0	0
1790	1790	2013-01-02	1	DAIRY	579.0	0
1822	1822	2013-01-02	10	CLEANING	1110.0	0

The Boolean mask here is filtering the DataFrame for rows where the family is “CLEANING” or “DAIRY”, **and** the sales are greater than 0



PRO TIP: QUERY

The `.query()` method lets you use SQL-like syntax to filter DataFrames

- You can specify any number of filtering conditions by using the “and” & “or” keywords

```
retail_df.query("family in ['CLEANING', 'DAIRY'] and sales > 0")
```

	id	date	store_nbr	family	sales	onpromotion
568	568	2013-01-01	25	CLEANING	186.0	0
569	569	2013-01-01	25	DAIRY	143.0	0
1789	1789	2013-01-02	1	CLEANING	1060.0	0
1790	1790	2013-01-02	1	DAIRY	579.0	0
1822	1822	2013-01-02	10	CLEANING	1110.0	0
...
3000797	3000797	2017-08-15	7	DAIRY	1279.0	25
3000829	3000829	2017-08-15	8	CLEANING	1198.0	13
3000830	3000830	2017-08-15	8	DAIRY	1330.0	24
3000862	3000862	2017-08-15	9	CLEANING	1439.0	25
3000863	3000863	2017-08-15	9	DAIRY	835.0	19

This query filters rows where the family is “CLEANING” or “DAIRY”, **and** the sales are greater than 0

Note that you don’t need to call the DataFrame name repeatedly, saving keystrokes and making the filter easier to interpret



PRO TIP: QUERY

The `.query()` method lets you use SQL-like syntax to filter DataFrames

- You can specify any number of filtering conditions by using the “and” & “or” keywords
- You can reference variables by using the “@” symbol

```
avg_sales = retail_df.loc[:, "sales"].mean()  
  
avg_sales  
  
357.77574911262707  
  
retail_df.query("family in ['CLEANING', 'DAIRY'] and sales > @avg_sales")  
  
... output truncated
```

	id	date	store_nbr	family	sales	onpromotion
1789	1789	2013-01-02	1	CLEANING	1060.0	0
1790	1790	2013-01-02	1	DAIRY	579.0	0
1822	1822	2013-01-02	10	CLEANING	1110.0	0
1855	1855	2013-01-02	11	CLEANING	3260.0	0
1888	1888	2013-01-02	12	CLEANING	1092.0	0

This query filters rows where the family is “CLEANING” or “DAIRY”, **and** the sales are greater than the avg_sales value (from the variable!)



SORTING DATAFRAMES BY INDICES

You can **sort a DataFrame by its indices** using the `.sort_index()` method

- This sorts rows (`axis=0`) by default, but you can specify `axis=1` to sort the columns

```
condition = retail_df.family.isin(["BEVERAGES", "DELI", "DAIRY"])
sample_df = retail_df[condition].sample(5, random_state=2021)
sample_df
```

This creates a sample DataFrame by filtering rows for the 3 specified product families, and grabbing 5 random rows

	id	date	store_nbr	family	sales	onpromotion
2782266	2782266	2017-04-15	25	BEVERAGES	5516.0	24
2356175	2356175	2016-08-18	2	DAIRY	714.0	7
1691390	1691390	2015-08-10	17	DAIRY	613.0	0
1435443	1435443	2015-03-19	35	DELI	134.0	24
2939747	2939747	2017-07-12	43	DAIRY	628.0	120

```
sample_df.sort_index(ascending=False)
```

This sorts the sample DataFrame in descending order by its row index
(it sorts in ascending order by default)

+	id	date	store_nbr	family	sales	onpromotion	
-	2939747	2939747	2017-07-12	43	DAIRY	628.0	120
-	2782266	2782266	2017-04-15	25	BEVERAGES	5516.0	24
-	2356175	2356175	2016-08-18	2	DAIRY	714.0	7
-	1691390	1691390	2015-08-10	17	DAIRY	613.0	0
-	1435443	1435443	2015-03-19	35	DELI	134.0	24



SORTING DATAFRAMES BY INDICES

You can **sort a DataFrame by its indices** using the `.sort_index()` method

- This sorts rows (`axis=0`) by default, but you can specify `axis=1` to sort the columns

```
condition = retail_df.family.isin(["BEVERAGES", "DELI", "DAIRY"])
sample_df = retail_df[condition].sample(5, random_state=2021)
sample_df
```

	<code>id</code>	<code>date</code>	<code>store_nbr</code>	<code>family</code>	<code>sales</code>	<code>onpromotion</code>
2782266	2782266	2017-04-15	25	BEVERAGES	5516.0	24
2356175	2356175	2016-08-18	2	DAIRY	714.0	7
1691390	1691390	2015-08-10	17	DAIRY	613.0	0
1435443	1435443	2015-03-19	35	DELI	134.0	24
2939747	2939747	2017-07-12	43	DAIRY	628.0	120

```
sample_df.sort_index(axis=1, inplace=True)
sample_df
```

	<code>date</code>	<code>family</code>	<code>id</code>	<code>onpromotion</code>	<code>sales</code>	<code>store_nbr</code>
2782266	2017-04-15	BEVERAGES	2782266	24	5516.0	25
2356175	2016-08-18	DAIRY	2356175	7	714.0	2
1691390	2015-08-10	DAIRY	1691390	0	613.0	17
1435443	2015-03-19	DELI	1435443	24	134.0	35
2939747	2017-07-12	DAIRY	2939747	120	628.0	43



Remember that DataFrame methods **don't sort in place by default**, allowing you to chain multiple methods together

This sorts the sample DataFrame in ascending order by its column index, and modifies the underlying values



SORTING DATAFRAMES BY VALUES

You can **sort a DataFrame by its values** using the `.sort_values()` method

- You can sort by a single column or by multiple columns

```
sample_df.sort_values("store_nbr")
```

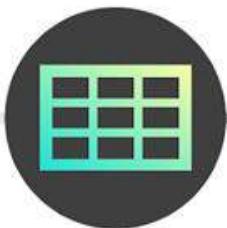
	date	family	id	onpromotion	sales	store_nbr
2356175	2016-08-18	DAIRY	2356175	7	714.0	2
1691390	2015-08-10	DAIRY	1691390	0	613.0	17
2782266	2017-04-15	BEVERAGES	2782266	24	5516.0	25
1435443	2015-03-19	DELI	1435443	24	134.0	35
2939747	2017-07-12	DAIRY	2939747	120	628.0	43

This sorts the sample DataFrame by the values in the `store_nbr` column in ascending order by default

```
sample_df.sort_values(["family", "sales"], ascending=[True, False])
```

	id	date	store_nbr	family	sales	onpromotion
2782266	2782266	2017-04-15	25	BEVERAGES	5516.0	24
2356175	2356175	2016-08-18	2	DAIRY	714.0	7
2939747	2939747	2017-07-12	43	DAIRY	628.0	120
1691390	1691390	2015-08-10	17	DAIRY	613.0	0
1435443	1435443	2015-03-19	35	DELI	134.0	24

This sorts the sample DataFrame by the values in the `family` column in ascending order, then by the values in the `sales` column in descending order within each family



RENAMING COLUMNS

Rename columns in place via assignment using the “columns” property

product_df

	product	price
0	Dairy	2.56
1	Dairy	2.56
2	Dairy	4.55
3	Vegetables	2.74
4	Fruits	5.44

```
product_df.columns = ["product_name", "cost"]  
product_df
```

	product_name	cost
0	Dairy	2.56
1	Dairy	2.56
2	Dairy	4.55
3	Vegetables	2.74
4	Fruits	5.44

Simply assign a list with the new column names using the **columns** property

```
product_df.columns = [col.upper() for col in product_df.columns]  
product_df
```

	PRODUCT_NAME	COST
0	Dairy	2.56
1	Dairy	2.56
2	Dairy	4.55
3	Vegetables	2.74
4	Fruits	5.44

Use a **list comprehension** to clean or standardize column titles using methods like `.upper()`



RENAMING COLUMNS

You can also **rename columns** with the `.rename()` method

`product_df`

	product	price
0	Dairy	2.56
1	Dairy	2.56
2	Dairy	4.55
3	Vegetables	2.74
4	Fruits	5.44

`product_df.rename(columns={'product': 'product_name', 'price': 'cost'})`

	product_name	cost
0	Dairy	2.56
1	Dairy	2.56
2	Dairy	4.55
3	Vegetables	2.74
4	Fruits	5.44

`product_df.rename(columns=lambda x: x.upper())`

	PRODUCT	PRICE
0	Dairy	2.56
1	Dairy	2.56
2	Dairy	4.55
3	Vegetables	2.74
4	Fruits	5.44

Use a dictionary to map the new column names to the old names



Note that the `.rename()` method
doesn't rename in place by default,
so you can chain methods together

Use **lambda functions** to clean
or standardize column titles
using methods like `.upper()`



REORDERING COLUMNS

Reorder columns with the `.reindex()` method when sorting won't suffice

product_df

	product	price	product_id
0	Dairy	2.56	1
1	Dairy	2.56	2
2	Dairy	4.55	3
3	Vegetables	2.74	4
4	Fruits	5.44	5

product_df.reindex(labels=["product_id", "product", "price"], axis=1)

	product_id	product	price
0	1	Dairy	2.56
1	2	Dairy	2.56
2	3	Dairy	4.55
3	4	Vegetables	2.74
4	5	Fruits	5.44



Pass a list of the existing columns in their desired order, and specify axis=1



ARITHMETIC COLUMN CREATION

You can **create columns with arithmetic** by assigning them Series operations

- Simply specify the new column name and assign the operation of interest

baby_books

	id	date	store_nbr	family	sales	onpromotion	
	2390356	2390356	2016-09-06	29	BABY CARE	3.0	0
	2691613	2691613	2017-02-23	31	BABY CARE	1.0	0
	2538925	2538925	2016-11-28	47	BOOKS	6.0	0
	2022637	2022637	2016-02-13	11	BABY CARE	1.0	0
	2585356	2585356	2016-12-24	5	BOOKS	2.0	0

```
baby_books[ "tax_amount" ] = baby_books[ "sales" ] * 0.05
```

```
baby_books[ "total" ] = baby_books[ "sales" ] + baby_books[ "tax_amount" ]
```

baby_books

	id	date	store_nbr	family	sales	onpromotion	tax_amount	total	
	2390356	2390356	2016-09-06	29	BABY CARE	3.0	0	0.15	3.15
	2691613	2691613	2017-02-23	31	BABY CARE	1.0	0	0.05	1.05
	2538925	2538925	2016-11-28	47	BOOKS	6.0	0	0.30	6.30
	2022637	2022637	2016-02-13	11	BABY CARE	1.0	0	0.05	1.05
	2585356	2585356	2016-12-24	5	BOOKS	2.0	0	0.10	2.10

This creates a new **tax_amount** column equal to **sales * 0.05**

This creates a new **total** column equal to **sales + tax_amount**

The new columns are added to the end of the DataFrame by default



BOOLEAN COLUMN CREATION

You can **create Boolean columns** by assigning them a logical test

```
baby_books["taxable_category"] = baby_books["family"] != "BABY CARE"  
baby_books
```

	id	date	store_nbr	family	sales	onpromotion	taxable_category	
	2390356	2390356	2016-09-06	29	BABY CARE	3.0	0	False
	2691613	2691613	2017-02-23	31	BABY CARE	1.0	0	False
	2538925	2538925	2016-11-28	47	BOOKS	6.0	0	True
	2022637	2022637	2016-02-13	11	BABY CARE	1.0	0	False
	2585356	2585356	2016-12-24	5	BOOKS	2.0	0	True

```
baby_books["tax_amount"] = (  
    baby_books["sales"] * 0.05 * (baby_books["family"] != "BABY CARE")  
)
```

	id	date	store_nbr	family	sales	onpromotion	tax_amount	
	2390356	2390356	2016-09-06	29	BABY CARE	3.0	0	0.0
	2691613	2691613	2017-02-23	31	BABY CARE	1.0	0	0.0
	2538925	2538925	2016-11-28	47	BOOKS	6.0	0	0.3
	2022637	2022637	2016-02-13	11	BABY CARE	1.0	0	0.0
	2585356	2585356	2016-12-24	5	BOOKS	2.0	0	0.1

This creates a new **taxable_category** column with Boolean values – True if the family is not "BABY CARE", and False if it is

This creates a new **tax_amount** column by leveraging both Boolean logic & arithmetic:
If the family is not "BABY CARE", then calculate the sales tax (sales * 0.05 * 1), otherwise return zero (sales * 0.05 * 0)



PRO TIP: NUMPY SELECT

NumPy's **select()** function lets you create columns based on multiple conditions

- This is more flexible than NumPy's `where()` function or Pandas' `.where()` method

```
conditions = [
    (baby_books["date"] == "2017-02-23") & (baby_books["family"] == "BABY CARE"),
    (baby_books["date"] == "2016-12-24") & (baby_books["family"] == "BOOKS"),
    (baby_books["date"] == "2016-09-06") & (baby_books["store_nbr"] > 28),
]
choices = ["Winter Clearance", "Christmas Eve", "New Store Special"]
baby_books["Sale_Name"] = np.select(conditions, choices, default="No Sale")
baby_books
```

Specify a set of **conditions** and outcomes (**choices**) for each condition

Then use **np.select** and pass in the conditions, the choices, and an optional default outcome if none of the conditions are met to the new `Sale_Name` column

2390356	2390356	2016-09-06	29	BABY CARE	3.0	0	0.0	3.0	False	New Store Special		
2691613	2691613	2017-02-23	31	BABY CARE	1.0	0	0.0	1.0	False	Winter Clearance	←	The first condition is met, so the first choice is returned
2538925	2538925	2016-11-28	47	BOOKS	6.0	0	0.3	6.3	True	No Sale		
2022637	2022637	2016-02-13	11	BABY CARE	1.0	0	0.0	1.0	False	No Sale		
2585356	2585356	2016-12-24	5	BOOKS	2.0	0	0.1	2.1	True	Christmas Eve		



MAPPING VALUES TO COLUMNS

The `.map()` method **maps values to a column** or an entire DataFrame

- You can pass a dictionary with existing values as the keys, and new values as the values

```
mapping_dict = {"Dairy": "Non-Vegan", "Vegetables": "Vegan", "Fruits": "Vegan"}  
product_df[ "Vegan?"] = product_df[ "product"].map(mapping_dict)  
product_df
```

product_id	product	price	Vegan?
0	1	Dairy	2.56
1	2	Dairy	2.56
2	3	Dairy	4.55
3	4	Vegetables	2.74
4	5	Fruits	5.44

Keys

Values

The dictionary keys will be mapped to the values in the column selected

This creates a new **Vegan?** column by mapping the dictionary keys to the values in the **product** column and returning the corresponding dictionary values in each row



MAPPING VALUES TO COLUMNS

The `.map()` method **maps values to a column** or an entire DataFrame

- You can pass a dictionary with existing values as the keys, and new values as the values
- You can apply lambda functions (*and others!*)

```
product_df["price"] = product_df["price"].map(lambda x: f"${x}")  
product_df
```

This overwrites the `price` column by adding a dollar sign to each of the previous values

product_id	product	price	Vegan?
0	1	Dairy	\$2.56 Non-Vegan
1	2	Dairy	\$2.56 Non-Vegan
2	3	Dairy	\$4.55 Non-Vegan
3	4	Vegetables	\$2.74 Vegan
4	5	Fruits	\$5.44 Vegan



PRO TIP: COLUMN CREATION WITH ASSIGN



How does this code work?

```
sample_df.assign(  
    tax_amount=(sample_df["sales"] * 0.05).round(2).map(lambda x: f"${x}"),  
    on_promotion_flag=np.where(sample_df["onpromotion"] > 0, True, False),  
    year=sample_df["date"].str.slice(0, 4).astype("int"),  
).query("family == 'DAIRY'")
```

	date	family	id	onpromotion	sales	store_nbr
2782266	2017-04-15	BEVERAGES	2782266	24	5516.0	25
2356175	2016-08-18	DAIRY	2356175	7	714.0	2
1691390	2015-08-10	DAIRY	1691390	0	613.0	17
1435443	2015-03-19	DELI	1435443	24	134.0	35
2939747	2017-07-12	DAIRY	2939747	120	628.0	43

Let's create some columns for the `sample_df` DataFrame!





PRO TIP: COLUMN CREATION WITH ASSIGN



How does this code work?

```
sample_df.assign(  
    tax_amount=(sample_df["sales"] * 0.05).round(2).map(lambda x: f"${x}"),  
    on_promotion_flag=np.where(sample_df["onpromotion"] > 0, True, False),  
    year=sample_df["date"].str.slice(0, 4).astype("int"),  
).query("family == 'DAIRY'")
```

	date	family	id	onpromotion	sales	store_nbr	tax_amount
2782266	2017-04-15	BEVERAGES	2782266	24	5516.0	25	\$275.8
2356175	2016-08-18	DAIRY	2356175	7	714.0	2	\$35.7
1691390	2015-08-10	DAIRY	1691390	0	613.0	17	\$30.65
1435443	2015-03-19	DELI	1435443	24	134.0	35	\$6.7
2939747	2017-07-12	DAIRY	2939747	120	628.0	43	\$31.4

First, a column called `tax_amount`:

- Equal to `sales * 0.05`
- Rounded to 2 decimals
- Starting with "\$"



PRO TIP: COLUMN CREATION WITH ASSIGN



How does this code work?

```
sample_df.assign(  
    tax_amount=(sample_df["sales"] * 0.05).round(2).map(lambda x: f"${x}"),  
    on_promotion_flag=np.where(sample_df["onpromotion"] > 0, True, False),  
    year=sample_df["date"].str.slice(0, 4).astype("int"),  
).query("family == 'DAIRY'")
```

	date	family	id	onpromotion	sales	store_nbr	tax_amount	on_promotion_flag
2782266	2017-04-15	BEVERAGES	2782266	24	5516.0	25	\$275.8	True
2356175	2016-08-18	DAIRY	2356175	7	714.0	2	\$35.7	True
1691390	2015-08-10	DAIRY	1691390	0	613.0	17	\$30.65	False
1435443	2015-03-19	DELI	1435443	24	134.0	35	\$6.7	True
2939747	2017-07-12	DAIRY	2939747	120	628.0	43	\$31.4	True

Next, a column called `on_promotion_flag`:

- If `onpromotion > 0` return `True`
- Otherwise, return `False`



PRO TIP: COLUMN CREATION WITH ASSIGN



How does this code work?

```
sample_df.assign(  
    tax_amount=(sample_df["sales"] * 0.05).round(2).map(lambda x: f"${x}"),  
    on_promotion_flag=np.where(sample_df["onpromotion"] > 0, True, False),  
    year=sample_df["date"].str.slice(0, 4).astype("int"),  
).query("family == 'DAIRY'")
```

	date	family	id	onpromotion	sales	store_nbr	tax_amount	on_promotion_flag	year
2782266	2017-04-15	BEVERAGES	2782266	24	5516.0	25	\$275.8	True	2017
2356175	2016-08-18	DAIRY	2356175	7	714.0	2	\$35.7	True	2016
1691390	2015-08-10	DAIRY	1691390	0	613.0	17	\$30.65	False	2015
1435443	2015-03-19	DELI	1435443	24	134.0	35	\$6.7	True	2015
2939747	2017-07-12	DAIRY	2939747	120	628.0	43	\$31.4	True	2017

Next, a last column called **year**:

- Equal to the first 4 characters on the **date** column
- Converted to an integer



PRO TIP: COLUMN CREATION WITH ASSIGN



How does this code work?

```
sample_df.assign(  
    tax_amount=(sample_df["sales"] * 0.05).round(2).map(lambda x: f"${x}"),  
    on_promotion_flag=np.where(sample_df["onpromotion"] > 0, True, False),  
    year=sample_df["date"].str.slice(0, 4).astype("int"),  
    ).query("family == 'DAIRY'")
```

	date	family	id	onpromotion	sales	store_nbr	tax_amount	on_promotion_flag	year
2356175	2016-08-18	DAIRY	2356175		7	714.0	2	\$35.7	True 2016
1691390	2015-08-10	DAIRY	1691390		0	613.0	17	\$30.65	False 2015
2939747	2017-07-12	DAIRY	2939747		120	628.0	43	\$31.4	True 2017



Finally, return the DataFrame and filter it:

- For rows where **family** is “DAIRY”



REVIEW: PANDAS DATA TYPES

Pandas data types mostly expand on their base Python and NumPy equivalents

Numeric:

Data Type	Description	Bit Sizes
bool	Boolean True/False	8
int64 (default)	Whole numbers	8, 16, 32, 64
float64 (default)	Decimal numbers	8, 16, 32, 64
boolean	Nullable Boolean True/False	8
Int64 (default)	Nullable whole numbers	8, 16, 32, 64
Float64 (default)	Nullable decimal numbers	32, 64

*Gray = NumPy data type

*Yellow = Pandas data type

Object / Text:

Data Type	Description
object	Any Python object
string	Only contains strings or text
category	Maps categorical data to a numeric array for efficiency

Time Series:

Data Type	Description
datetime	A single moment in time (January 4, 2015, 2:00:00 PM)
timedelta	The duration between two dates or times (10 days, 3 seconds, etc.)
period	A span of time (a day, a week, etc.)



THE CATEGORICAL DATA TYPE

The Pandas **categorical data type** stores text data with repeated values efficiently

- Python maps each unique category to an integer to save space
- As a rule of thumb, only consider this data type when unique categories < number of rows / 2

```
sample_df.astype({"family": "category"})
```

	id	date	store_nbr	family	sales	onpromotion	family
	2782266	2782266	2017-04-15	25	BEVERAGES	5516.0	24
	2356175	2356175	2016-08-18	2	DAIRY	714.0	7
	1691390	1691390	2015-08-10	17	DAIRY	613.0	0
	1435443	1435443	2015-03-19	35	DELI	134.0	24
	2939747	2939747	2017-07-12	43	DAIRY	628.0	120

These are now stored as integers in the backend



The categorical data type has some quirks during some data manipulation operations that will force it back into an object data type, but it's not something we'll cover in depth in this course



TYPE CONVERSION

Like Series, you can **convert data types** in a DataFrame by using the `.astype()` method and specifying the desired data type (*if compatible*)

```
sample_df["sales_int"] = sample_df["sales"].astype("int")  
sample_df
```

	id	date	store_nbr	family	sales	onpromotion	sales_int	
	2782266	2782266	2017-04-15	25	BEVERAGES	5516.0	24	5516
	2356175	2356175	2016-08-18	2	DAIRY	714.0	7	714
	1691390	1691390	2015-08-10	17	DAIRY	613.0	0	613
	1435443	1435443	2015-03-19	35	DELI	134.0	24	134
	2939747	2939747	2017-07-12	43	DAIRY	628.0	120	628

This creates a new 'sales_int' column by converting 'sales' to integers

```
sample_df = sample_df.astype({"date": "Datetime64", "onpromotion": "float"})  
sample_df
```

	id	date	store_nbr	family	sales	onpromotion	sales_int	
	2782266	2782266	2017-04-15	25	BEVERAGES	5516.0	24.0	5516
	2356175	2356175	2016-08-18	2	DAIRY	714.0	7.0	714
	1691390	1691390	2015-08-10	17	DAIRY	613.0	0.0	613
	1435443	1435443	2015-03-19	35	DELI	134.0	24.0	134
	2939747	2939747	2017-07-12	43	DAIRY	628.0	120.0	628

You can use the `.astype()` method on the entire DataFrame and pass a dictionary with the columns as keys and the desired data type as values



TYPE CONVERSION

Like Series, you can **convert data types** in a DataFrame by using the `.astype()` method and specifying the desired data type (*if compatible*)

product_df

	product_id	product	price	Vegan?
0	1	Dairy	\$2.56	Non-Vegan
1	2	Dairy	\$2.56	Non-Vegan
2	3	Dairy	\$4.55	Non-Vegan
3	4	Vegetables	\$2.74	Vegan
4	5	Fruits	\$5.44	Vegan

product_df.astype({"price": "float"})

```
ValueError: could not convert string to float: '$2.56'
```

.astype() will return a ValueError
if the data type is incompatible

product_df.assign(price=product_df["price"].str.strip("\$").astype("float"))

	product_id	product	price	Vegan?
0	1	Dairy	2.56	Non-Vegan
1	2	Dairy	2.56	Non-Vegan
2	3	Dairy	4.55	Non-Vegan
3	4	Vegetables	2.74	Vegan
4	5	Fruits	5.44	Vegan

But applying cleaning steps
to the data can make it work



STEP 1: DROP COLUMNS

Dropping unnecessary columns is an easy way to free up significant space

- You may not know which columns are important when reading in a dataset for the first time
- If you do, you can limit the columns you read in to begin with (*more on that later!*)

class_data						
	id	start_date	title	class_level	price	students_enrolled
0	0	2022-01-11	Algebra I	8.0	\$299	102
1	1	2022-02-22	Algebra 2	9.0	-	43
2	2	2022-03-03	Geometry	NaN	\$399	16
3	3	2022-04-04	Trigonometry	11.0	\$599	8
4	4	2022-05-05	Calculus	12.0	-	-

Note that the id column
is identical to the index

```
class_data.drop("id", axis=1, inplace=True)  
class_data.memory_usage(deep=True).sum()
```

1427

Note that the memory usage
went down from 1,470 bytes

} By dropping the 'id' column, around 40
bytes were freed (~4% of memory use)



STEP 2: CONVERTING OBJECT DATA TYPES

Try to **convert object data types** to numeric or datetime whenever possible

class_data

	start_date	title	class_level	price	students_enrolled
0	2022-01-11	Algebra I	8.0	\$299	102
1	2022-02-22	Algebra 2	9.0	-	43
2	2022-03-03	Geometry	NaN	\$399	16
3	2022-04-04	Trigonometry	11.0	\$599	8
4	2022-05-05	Calculus	12.0	-	-

Note that the missing values in 'price' and 'students_enrolled' are not NaN values

Use **memory_usage="deep"** with the .info() method to get total memory usage along with the column data types

```
class_data.info(memory_usage="deep")
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 5 entries, 0 to 4
Data columns (total 5 columns):
 #   Column           Non-Null Count  Dtype  
 --- 
 0   start_date       5 non-null      object 
 1   title            5 non-null      object 
 2   class_level      4 non-null      float64
 3   price            5 non-null      object 
 4   students_enrolled 5 non-null     object 
dtypes: float64(1), object(4)
memory usage: 1.4 KB
```

Text data, usually including dates, is read in as an object by default

Numeric data is usually read in as 64-bit (like 'class_level') but errors in the data can cause it to be read in as an object



STEP 2: CONVERTING OBJECT DATA TYPES

Try to **convert object data types** to numeric or datetime whenever possible

```
class_data.memory_usage(deep=True)
```

```
Index          128
start_date    335
title         331
class_level   40
price          299
students_enrolled  294
dtype: int64
```

```
class_data = class_data.astype({"start_date": "Datetime64", "title": "string"})
```

```
class_data.memory_usage(deep=True)
```

```
Index          128
start_date    40
title         331
class_level   40
price          299
students_enrolled  294
dtype: int64
```

Note that by converting 'start_date' to a
datetime data type, around 295 bytes
were freed (~20% of memory use)

On the other hand, nothing changed by
converting 'title' to a string data type



STEP 2: CONVERTING OBJECT DATA TYPES

Try to **convert object data types** to numeric or datetime whenever possible

class_data

	start_date	title	class_level	price	students_enrolled
0	2022-01-11	Algebra I	8.0	\$299	102
1	2022-02-22	Algebra 2	9.0	-	43
2	2022-03-03	Geometry	NaN	\$399	16
3	2022-04-04	Trigonometry	11.0	\$599	8
4	2022-05-05	Calculus	12.0	-	-

```
class_data["students_enrolled"] = class_data["students_enrolled"].replace("-",np.nan).astype("float")
class_data["price"] = class_data["price"].replace("-",np.nan).str.strip("$").astype("float")
```

```
class_data.info(memory_usage="deep")
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 5 entries, 0 to 4
Data columns (total 5 columns):
 #   Column           Non-Null Count   Dtype  
 ---  --  
 0   start_date      5 non-null       datetime64[ns]
 1   title           5 non-null       string  
 2   class_level     4 non-null       float64 
 3   price           3 non-null       float64 
 4   students_enrolled 4 non-null       float64 
dtypes: datetime64[ns](1), float64(3), string(1)
memory usage: 619.0 bytes
```

#	Column	Non-Null Count	Dtype
0	start_date	5 non-null	datetime64[ns]
1	title	5 non-null	string
2	class_level	4 non-null	float64
3	price	3 non-null	float64
4	students_enrolled	4 non-null	float64

This is now only 619 bytes!

Note that additional methods were chained to convert 'price' and 'students_enrolled' to floats:

- "-" was replaced by NaN values on both
- "\$" was stripped on 'price'



STEP 3: DOWNCASE NUMERIC DATA

Integers and floats are cast as 64-bit by default to handle any possible value, but you can **downcase numeric data** to a smaller bit size to save space if possible

- 8-bits = -128 to 127
- 16-bits = -32,768 to 32,767
- 32-bits = -2,147,483,648 to 2,147,483,647
- 64-bits = -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807

class_data					
	start_date	title	class_level	price	students_enrolled
0	2022-01-11	Algebra I	8.0	299.0	102.0
1	2022-02-22	Algebra 2	9.0	NaN	43.0
2	2022-03-03	Geometry	NaN	399.0	16.0
3	2022-04-04	Trigonometry	11.0	599.0	8.0
4	2022-05-05	Calculus	12.0	NaN	NaN

Based on the ranges above, these columns can be downcase as follows:

- 'class_level' to Int8
- 'price' to Int16
- 'students_enrolled' to Int16



STEP 3: DOWNCAST NUMERIC DATA

Integers and floats are cast as 64-bit by default to handle any possible value, but you can **downcast numeric data** to a smaller bit size to save space if possible

```
class_data = class_data.astype(  
    {  
        "class_level": "Int8",  
        "price": "Int16",  
        "students_enrolled": "Int16"  
    }  
  
    )  
  
class_data.info(memory_usage="deep")  
  
<class 'pandas.core.frame.DataFrame'>  
RangeIndex: 5 entries, 0 to 4  
Data columns (total 5 columns):  
 #   Column           Non-Null Count  Dtype     
 ---  --  
 0   start_date      5 non-null       datetime64[ns]  
 1   title            5 non-null       string  
 2   class_level     4 non-null       Int8  
 3   price            3 non-null       Int16  
 4   students_enrolled 4 non-null       Int16  
dtypes: Int16(2), Int8(1), datetime64[ns](1), string(1)  
memory usage: 539.0 bytes
```

This is now only 539 bytes!

class_data					
	start_date	title	class_level	price	students_enrolled
0	2022-01-11	Algebra I	8	299	102
1	2022-02-22	Algebra 2	9	<NA>	43
2	2022-03-03	Geometry	<NA>	399	16
3	2022-04-04	Trigonometry	11	599	8
4	2022-05-05	Calculus	12	<NA>	<NA>

Note that, since these are Pandas' Nullable data types, the NumPy NaN values are now Pandas NA values



STEP 4: USING CATEGORICAL DATA TYPES

Use the **categorical data type** if you have string columns where the number of unique values is less than half of the total number of rows

```
class_data["title"] = class_data["title"].astype("category")  
  
class_data.info(memory_usage="deep")  
  
<class 'pandas.core.frame.DataFrame'>  
RangeIndex: 5 entries, 0 to 4  
Data columns (total 5 columns):  
 #   Column           Non-Null Count  Dtype     
---  --  
 0   start_date      5 non-null       datetime64[ns]  
 1   title            5 non-null       category  
 2   class_level     4 non-null       Int8  
 3   price            3 non-null       Int16  
 4   students_enrolled 4 non-null       Int16  
dtypes: Int16(2), Int8(1), category(1), datetime64[ns](1)  
memory usage: 716.0 bytes
```

This went up to 716 bytes!

class_data					
	start_date	title	class_level	price	students_enrolled
0	2022-01-11	Algebra I		8	299
1	2022-02-22	Algebra 2		9 <NA>	43
2	2022-03-03	Geometry		<NA>	399
3	2022-04-04	Trigonometry		11	599
4	2022-05-05	Calculus		12 <NA>	<NA>

In this case, there are five unique categories in five rows, so this just added overhead
(the demo will show the memory reduction)



RECAP: MEMORY OPTIMIZATION

BEFORE

class_data

	id	start_date	title	class_level	price	students_enrolled
0	0	2022-01-11	Algebra I	8.0	\$299	102
1	1	2022-02-22	Algebra 2	9.0	-	43
2	2	2022-03-03	Geometry	NaN	\$399	16
3	3	2022-04-04	Trigonometry	11.0	\$599	8
4	4	2022-05-05	Calculus	12.0	-	-

class_data.info(memory_usage="deep")

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 5 entries, 0 to 4
Data columns (total 6 columns):
 #   Column           Non-Null Count  Dtype  
--- 
 0   id               5 non-null      int64  
 1   start_date       5 non-null      object  
 2   title            5 non-null      object  
 3   class_level      4 non-null      float64 
 4   price            5 non-null      object  
 5   students_enrolled 5 non-null     object  
dtypes: float64(1), int64(1), object(4)
memory usage: 1.4 KB
```

AFTER

class_data

		start_date	title	class_level	price	students_enrolled
0	2022-01-11	Algebra I	8	299	102	
1	2022-02-22	Algebra 2	9	<NA>	43	
2	2022-03-03	Geometry	<NA>	399	16	
3	2022-04-04	Trigonometry	11	599	8	
4	2022-05-05	Calculus	12	<NA>	<NA>	

class_data.info(memory_usage="deep")

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 5 entries, 0 to 4
Data columns (total 5 columns):
 #   Column           Non-Null Count  Dtype    
--- 
 0   start_date       5 non-null      datetime64[ns]
 1   title            5 non-null      string   
 2   class_level      4 non-null      Int8    
 3   price            3 non-null      Int16   
 4   students_enrolled 4 non-null     Int16  
dtypes: Int16(2), Int8(1), datetime64[ns](1), string(1)
memory usage: 539.0 bytes
```



63% less memory!



A NOTE ON EFFICIENCY

“Premature optimization is the root of all evil” – Don Knuth

- Data analysis is an **iterative process**
 - It's normal to not know the best or most efficient version of a dataset from the beginning
 - Once you understand the data and have an analysis path, then you can optimize
- Efficiency's importance **depends on the use case**
 - If all you need is a quick analysis, fully optimizing your code will only waste time
 - If you're building a pipeline that will run frequently, efficiency & optimization are critical
- Build **efficient habits & workflows**
 - As you work more with Python and Pandas, take time to review your code and note areas for improvement – you'll be able to incorporate better practices next time!

AGGREGATING & RESHAPING

AGGREGATING & RESHAPING



In this section we'll cover **aggregating & reshaping** DataFrames, including grouping columns, performing aggregation calculations, and pivoting & unpivoting data

TOPICS WE'LL COVER:

Grouping Columns

Multi-Index DataFrames

Aggregating Groups

Pivot Tables

Melting DataFrames

GOALS FOR THIS SECTION:

- Group DataFrames by one or more columns and calculate aggregate statistics by group
- Learn to access multi-index DataFrames and reset them to return to a single index
- Create Excel-style PivotTables to summarize data
- Melt “wide” tables of data into a “long” tabular form



AGGREGATING DATAFRAMES

You can **aggregate a DataFrame** column by using aggregation methods (like Series!)

```
retail.loc[:, ['sales', 'onpromotion']].sample(100).sum().round(2)
```

```
sales      33729.57
onpromotion    240.00
dtype: float64
```

```
retail.loc[:, ['sales', 'onpromotion']].sample(100).mean().round(2)
```

```
sales      299.68
onpromotion    2.36
dtype: float64
```

But what if you want multiple aggregate statistics, or summarized statistics by groups?



GROUPING DATAFRAMES

Grouping a DataFrame allows you to aggregate the data at a different level

- For example, transform daily data into monthly, roll up transaction level data by store, etc.

Original DataFrame

	id	date	store_nbr	family	sales	onpromotion
0	0	2013-01-01	1	AUTOMOTIVE	0.0	0
1	1	2013-01-01	1	BABY CARE	0.0	0
2	2	2013-01-01	1	BEAUTY	0.0	0
3	3	2013-01-01	1	BEVERAGES	0.0	0
4	4	2013-01-01	1	BOOKS	0.0	0

Raw sales data by transaction

Grouped by Family

	family	sales
	AUTOMOTIVE	264.0
	BABY CARE	0.0
	BEAUTY	121.0
	BEVERAGES	29817.0
	BOOKS	0.0

Total sales by family





GROUPING DATAFRAMES

To group data, use the `.groupby()` method and specify a column to group by

- The grouped column becomes the index by default

Just specify the columns to group by

```
small_retail.groupby('family')
```

```
<pandas.core.groupby.generic.DataFrameGroupBy object at 0x7f9fe9e30910>
```

This returns a "groupby" object

```
small_retail.groupby('family')['sales'].sum().head()
```

```
family
AUTOMOTIVE      264.0
BABY CARE        0.0
BEAUTY           121.0
BEVERAGES        29817.0
BOOKS             0.0
Name: sales, dtype: float64
```

Using single brackets
returns a Series

To return the groups created, you need to
calculate aggregate statistics:

- Specify the column for the calculations
- Apply an aggregation method



GROUPING DATAFRAMES

To group data, use the `.groupby()` method and specify a column to group by

- The grouped column becomes the index by default

Just specify the columns to group by

```
small_retail.groupby('family')
```

```
<pandas.core.groupby.generic.DataFrameGroupBy object at 0x7f9fe9e30910>
```

This returns a "groupby" object

```
small_retail.groupby('family')[['sales']].sum().head()
```

	sales
family	
AUTOMOTIVE	264.0
BABY CARE	0.0
BEAUTY	121.0
BEVERAGES	29817.0
BOOKS	0.0

Using double brackets
returns a DataFrame

To return the groups created, you need to
calculate aggregate statistics:

- Specify the column for the calculations
- Apply an aggregation method



GROUPING BY MULTIPLE COLUMNS

You can **group by multiple columns** by passing the list of columns into `.groupby()`

- This creates a multi-index object with an index for each column the data was grouped by

```
small_retail.groupby(['family', 'store_nbr'])[['sales']].sum()
```

		sales
family	store_nbr	
AUTOMOTIVE	11	8.000000
	12	9.000000
	13	7.000000
	16	5.000000
	19	5.000000
...		...
SEAFOOD	47	48.239998
	49	78.718000
	50	18.583000
	53	2.000000
	8	57.757000

780 rows × 1 columns

This is a multi-index DataFrame

This returns the sum of sales for each combination of 'family' and 'store_nbr'



GROUPING BY MULTIPLE COLUMNS

You can **group by multiple columns** by passing the list of columns into `.groupby()`

- This creates a multi-index object with an index for each column the data was grouped by
- Specify `as_index=False` to prevent the grouped columns from becoming indices

```
sales_sums = small_retail.groupby(['family', 'store_nbr'],
                                 as_index=False)[['sales']].sum()
```

```
sales_sums
```

	family	store_nbr	sales
0	AUTOMOTIVE	11	8.000000
1	AUTOMOTIVE	12	9.000000
2	AUTOMOTIVE	13	7.000000
3	AUTOMOTIVE	16	5.000000
4	AUTOMOTIVE	19	5.000000
...
775	SEAFOOD	47	48.239998
776	SEAFOOD	49	78.718000
777	SEAFOOD	50	18.583000
778	SEAFOOD	53	2.000000
779	SEAFOOD	8	57.757000

780 rows × 3 columns

This still returns the sum of sales for each combination of 'family' and 'store_nbr', but keeps a numeric index



MULTI-INDEX DATAFRAMES

Multi-index DataFrames are generally created through aggregation operations

- They are stored as a list of tuples, with an item for each layer of the index

sales_sums		
		sales
family	store_nbr	
AUTOMOTIVE	11	8.000000
	12	9.000000
	13	7.000000
	16	5.000000
	19	5.000000
...
SEAFOOD	47	48.239998
	49	78.718000
	50	18.583000
	53	2.000000
	8	57.757000

780 rows × 1 columns



`sales_sums.index`

```
MultiIndex([( 'AUTOMOTIVE', '11'),
            ('AUTOMOTIVE', '12'),
            ('AUTOMOTIVE', '13'),
            ('AUTOMOTIVE', '16'),
            ('AUTOMOTIVE', '19'),
            ('AUTOMOTIVE', '21'),
            ('AUTOMOTIVE', '22'),
            ('AUTOMOTIVE', '26'),
            ('AUTOMOTIVE', '27'),
            ('AUTOMOTIVE', '28'),
            ...
            ('    'SEAFood', '34'),
            ('    'SEAFood', '35'),
            ('    'SEAFood', '39'),
            ('    'SEAFood', '43'),
            ('    'SEAFood', '45'),
            ('    'SEAFood', '47'),
            ('    'SEAFood', '49'),
            ('    'SEAFood', '50'),
            ('    'SEAFood', '53'),
            ('    'SEAFood', '8')],
           names=['family', 'store_nbr'], length=780)
```



ACCESSING MULTI-INDEX DATAFRAMES

The `.loc[]` accessor lets you **access multi-index DataFrames** in different ways:

1. Access rows via the **outer index** only

`sales_sums`

sales	
family	store_nbr
AUTOMOTIVE	11 8.000000
	12 9.000000
	13 7.000000
	16 5.000000
	19 5.000000
...	...
SEAFOOD	47 48.239998
	49 78.718000
	50 18.583000
	53 2.000000
	8 57.757000

780 rows × 1 columns



`sales_sums.loc['AUTOMOTIVE'].head()`

sales	
store_nbr	
11	8.0
12	9.0
13	7.0
16	5.0
19	5.0

All rows for "AUTOMOTIVE"
(note that 'family' is dropped)

`sales_sums.loc["AUTOMOTIVE":"BEAUTY"]`

sales	
family	store_nbr
AUTOMOTIVE	11 8.0
	12 9.0
	13 7.0
	16 5.0
	19 5.0
...	...
BEAUTY	45 10.0
	47 34.0
	48 8.0
	52 0.0
	9 7.0

72 rows × 1 columns

All rows from
"AUTOMOTIVE"
to "BEAUTY"
(inclusive)



ACCESSING MULTI-INDEX DATAFRAMES

The .loc[] accessor lets you **access multi-index DataFrames** in different ways:

2. Access rows via the **outer & inner indices**

sales_sums		
sales		
family	store_nbr	
AUTOMOTIVE	11	8.000000
	12	9.000000
	13	7.000000
	16	5.000000
	19	5.000000

SEAFOOD	47	48.239998
	49	78.718000
	50	18.583000
	53	2.000000
	8	57.757000

780 rows x 1 columns



`sales_sums.loc[('AUTOMOTIVE', '12'), :]`

`sales` 9.0
Name: (AUTOMOTIVE, 12), dtype: float64

All rows for "AUTOMOTIVE" and "12"
(note that 'family' and 'store_nbr' are dropped)

`sales_sums.loc[("AUTOMOTIVE", "11"):(("BEAUTY", "11"), :]`

family	store_nbr
AUTOMOTIVE	11
	12
	13

All rows from "AUTOMOTIVE" and "11"
to "BEAUTY" and "11" (inclusive)

....All rows in Automotive and Baby Care

BEAUTY	1	7.0
	10	1.0
	11	11.0



MODIFYING MULTI-INDEX DATAFRAMES

There are several ways to **modify multi-index DataFrames**:

Reset the index

Moves the index levels back to DataFrame columns

```
sales_sums.reset_index()
```

	family	store_nbr	sales
0	AUTOMOTIVE	11	8.000000
1	AUTOMOTIVE	12	9.000000
2	AUTOMOTIVE	13	7.000000
3	AUTOMOTIVE	16	5.000000
4	AUTOMOTIVE	19	5.000000
...

Swap the index level

Changes the hierarchy for the index levels

```
sales_sums.swaplevel()
```

	store_nbr	family	sales
11	AUTOMOTIVE	8.000000	
12	AUTOMOTIVE	9.000000	
13	AUTOMOTIVE	7.000000	
16	AUTOMOTIVE	5.000000	
19	AUTOMOTIVE	5.000000	
...

Drop an index level

Drops an index level from the DataFrame entirely

```
sales_sums.droplevel('family')
```

	store_nbr	sales
11	8.000000	
12	9.000000	
13	7.000000	
16	5.000000	
19	5.000000	
...



PRO TIP: In most cases it's best to reset the index and avoid multi-index DataFrames – they're not very intuitive!



Be careful! You may lose important information



THE AGG METHOD

The `.agg()` method lets you perform multiple aggregations on a “groupby” object

	date	store_nbr	family	sales	onpromotion
2862475	2017-05-30	25	LIQUOR,WINE,BEER	92.0	2
940501	2014-06-13	48	BABY CARE	0.0	0
1457967	2015-04-01	17	PLAYERS AND ELECTRONICS	0.0	0
1903307	2015-12-07	12	SEAFOOD	3.0	0
196280	2013-04-21	16	PREPARED FOODS	66.0	0



```
small_retail.groupby(['store_nbr', 'family']).agg('sum')
```

store_nbr	family	sales	
		onpromotion	
1	BEAUTY	7.000	1
	BREAD/BAKERY	822.484	9
	CELEBRATION	2.000	0
	CLEANING	682.000	1
	DAIRY	585.000	3
...
9	LADIESWEAR	5.000	0
	LAWN AND GARDEN	8.000	0
	LINGERIE	15.000	0
	LIQUOR,WINE,BEER	69.000	0
	PET SUPPLIES	0.000	0

780 rows × 2 columns

The `.agg()` method will perform the aggregation on all compatible columns, in this case ‘sales’ and ‘onpromotion’, which are numeric



MULTIPLE AGGREGATIONS

You can perform **multiple aggregations** by passing a list of aggregation functions

```
small_retail.groupby(['family', 'store_nbr']).agg(['sum', 'mean'])
```

family	store_nbr	sales		onpromotion	
		sum	mean	sum	mean
AUTOMOTIVE	11	8.000000	8.000000	0	0.0
	12	9.000000	9.000000	0	0.0
	13	7.000000	3.500000	0	0.0
	16	5.000000	5.000000	0	0.0
	19	5.000000	5.000000	0	0.0
SEAFOOD
	47	48.239998	48.239998	0	0.0
	49	78.718000	78.718000	0	0.0
	50	18.583000	18.583000	5	5.0
	53	2.000000	1.000000	0	0.0
	8	57.757000	57.757000	0	0.0

780 rows × 4 columns

This creates two levels in the column index,
one for the original column names, and
another for the aggregations performed



MULTIPLE AGGREGATIONS

You can perform **specific aggregations by column** by passing a dictionary with column names as keys, and lists of aggregation functions as values

```
(small_retail  
    .groupby(['family', 'store_nbr'])  
    .agg({'sales': ['sum', 'mean'],  
          'onpromotion':[ 'min', 'max']})  
)
```

The calculates the sum and mean for 'sales' and the min and max for 'onpromotion'

family	store_nbr	sales		onpromotion	
		sum	mean	min	max
AUTOMOTIVE	11	8.000000	8.000000	0	0
	12	9.000000	9.000000	0	0
	13	7.000000	3.500000	0	0
	16	5.000000	5.000000	0	0
	19	5.000000	5.000000	0	0
...	
SEAFOOD	47	48.239998	48.239998	0	0
	49	78.718000	78.718000	0	0
	50	18.583000	18.583000	5	5
	53	2.000000	1.000000	0	0
	8	57.757000	57.757000	0	0

780 rows × 4 columns



NAMED AGGREGATIONS

You can **name aggregated columns** upon creation to avoid multi-index columns

Specify the new column name and assign it a tuple with the column you want to aggregate and the aggregation to perform

```
(small_retail  
    .groupby(['family', 'store_nbr'])  
    .agg(sales_sum=('sales', 'sum'),  
         sales_avg=('sales', 'mean'),  
         on_promotion_max=('onpromotion', 'max'))  
)
```

A single column index!

family	store_nbr	sales_sum	sales_avg	on_promotion_max
AUTOMOTIVE	11	8.000000	8.000000	0
	12	9.000000	9.000000	0
	13	7.000000	3.500000	0
	16	5.000000	5.000000	0
	19	5.000000	5.000000	0
...				
SEAFOOD	47	48.239998	48.239998	0
	49	78.718000	78.718000	0
	50	18.583000	18.583000	5
	53	2.000000	1.000000	0
	8	57.757000	57.757000	0

780 rows × 3 columns



PRO TIP: TRANSFORM

The `.transform()` method can be used to perform aggregations without reshaping

- This is useful for calculating group-level statistics to perform row-level analysis

```
small_retail.assign(store_sales = (small_retail  
                                 .groupby('store_nbr')[ 'sales' ]  
                                 .transform('sum')))
```

This uses `.assign()` to create a new DataFrame column, and `.transform()` calculates the sum of 'sales' by 'store_nbr' and applies the corresponding value to each row

	date	store_nbr	family	sales	onpromotion	store_sales
2862475	2017-05-30	25	LIQUOR,WINE,BEER	92.000	2	4800.79800
940501	2014-06-13	48	BABY CARE	0.000	0	4058.12603
1457967	2015-04-01	17	PLAYERS AND ELECTRONICS	0.000	0	1099.15101
1903307	2015-12-07	12	SEAFOOD	3.000	0	3288.39100
196280	2013-04-21	16	PREPARED FOODS	66.000	0	4365.89000
...
1875481	2015-11-21	31	PERSONAL CARE	273.000	0	3816.56100
2129111	2016-04-12	48	HOME APPLIANCES	0.000	0	4058.12603
1967389	2016-01-13	10	POULTRY	139.781	0	5272.78100
480840	2013-09-27	5	PRODUCE	7.000	0	11566.84600
23887	2013-01-14	29	POULTRY	0.000	0	4291.98500

The value for rows with store 48 is the same!

1000 rows x 6 columns



PIVOT TABLES

The .pivot_table() method lets you create Excel-style **pivot tables**

```
smaller_retail.pivot_table(index='family',
                           columns='store_nbr',
                           values='sales',
                           aggfunc='sum')
```

store_nbr	1	2	3	4
family				
AUTOMOTIVE	5475.0	9100.0	15647.0	6767.0
BABY CARE	0.0	84.0	672.0	24.0
BEAUTY	4056.0	7936.0	16189.0	6890.0



The screenshot shows the Microsoft Excel ribbon with the 'PivotTable Fields' tab selected. Under the 'Columns' section, 'store_nbr' is listed. Under the 'Rows' section, 'family' is listed. Under the 'Values' section, 'Sum of sales' is listed.

Sum of sales	store_nbr	1	2	3	4
family					
AUTOMOTIVE		5475.0	9100.0	15647.0	6767.0
BABY CARE		0.0	84.0	672.0	24.0
BEAUTY		4056.0	7936.0	16189.0	6890.0



Unlike Excel, Pandas pivot tables **don't have a "filter" argument**, but you can filter your DataFrame before pivoting to return a filtered pivot table



PIVOT TABLE ARGUMENTS

The `.pivot_table()` method has these arguments:

- **index**: returns a row index with distinct values from the specified column
- **columns**: returns a column index with distinct values from the specified column
- **values**: the column, or columns, to perform the aggregations on
- **aggfunc**: defines the aggregation function, or functions, to perform on the “values”
- **margins**: returns row and column totals when True (*False by default*)

```
smaller_retail.pivot_table(index='family',
                           columns='store_nbr',
                           values='sales',
                           aggfunc='sum')
```

store_nbr	1	2	3	4
family				
AUTOMOTIVE	5475.0	9100.0	15647.0	6767.0
BABY CARE	0.0	84.0	672.0	24.0
BEAUTY	4056.0	7936.0	16189.0	6890.0

This returns distinct 'family' values as rows, distinct 'store_nbr' values as columns, and sums the 'sales' for each combination of 'family' and 'str_nbr' as the values.



PIVOT TABLE ARGUMENTS

The `.pivot_table()` method has these arguments:

- **index**: returns a row index with distinct values from the specified column
- **columns**: returns a column index with distinct values from the specified column
- **values**: the column, or columns, to perform the aggregations on
- **aggfunc**: defines the aggregation function, or functions, to perform on the “values”
- **margins**: returns row and column totals when True (*False by default*)

```
smaller_retail.pivot_table(index='family',
                           columns='store_nbr',
                           values='sales',
                           aggfunc='sum',
                           margins=True)
```

store_nbr	1	2	3	4	All
family					
AUTOMOTIVE	5475.0	9100.0	15647.0	6767.0	36989.0
BABY CARE	0.0	84.0	672.0	24.0	780.0
BEAUTY	4056.0	7936.0	16189.0	6890.0	35071.0
All	9531.0	17120.0	32508.0	13681.0	72840.0

Specifying **margins=True** adds row and column totals based on the aggregation (the corner represents the grand total)



MULTIPLE AGGREGATION FUNCTIONS

Multiple aggregation functions can be passed to the “aggfunc” argument

- The new values are added as additional columns

```
smaller_retail.pivot_table(index='family',
                           columns='store_nbr',
                           values='sales',
                           aggfunc=('min', 'max'))
```

The functions are passed as a tuple

store_nbr	max				min			
	1	2	3	4	1	2	3	4
family								
AUTOMOTIVE	19.0	23.0	48.0	22.0	0.0	0.0	0.0	0.0
BABY CARE	0.0	5.0	11.0	3.0	0.0	0.0	0.0	0.0
BEAUTY	12.0	108.0	93.0	19.0	0.0	0.0	0.0	0.0

*There is a column for each store_nbr min and max
(this can create a very wide dataset very quickly)*



MULTIPLE AGGREGATION FUNCTIONS

Multiple aggregation functions can be passed to the “aggfunc” argument

- The new values are added as additional columns

```
smaller_retail.pivot_table(  
    index="family",  
    columns="store_nbr",  
    aggfunc={"sales": ["sum", "mean"], "onpromotion": "max"}),
```

Use a dictionary to apply specific functions to specific columns

store_nbr	onpromotion		sales									
			max	mean								
	1	2	3	4	1	2	3	4	1	2	3	4
family												
AUTOMOTIVE	1	1	1	1	3.251188	5.403800	9.291568	4.018409	5475.0	9100.0	15647.0	6767.0
BABY CARE	0	0	1	0	0.000000	0.049881	0.399050	0.014252	0.0	84.0	672.0	24.0
BEAUTY	2	2	2	2	2.408551	4.712589	9.613420	4.091449	4056.0	7936.0	16189.0	6890.0



PIVOT TABLES VS. GROUPBY

If the column argument isn't specified in a pivot table, it will return a table that's identical to one grouped by the index columns

```
smaller_retail.pivot_table(index=['family', 'store_nbr'],
                           aggfunc={'sum', 'mean'},
                           values=['sales', 'onpromotion'])
```

```
smaller_retail.groupby(['family', 'store_nbr']).agg(
    {"onpromotion": ["mean", "sum"],
     "sales": ["mean", "sum"]})
```

family	store_nbr	onpromotion		sales	
		mean	sum	mean	sum
AUTOMOTIVE	1	0.008314	14.0	3.251188	5475.0
	2	0.007720	13.0	5.403800	9100.0
	3	0.007720	13.0	9.291568	15647.0
	4	0.005938	10.0	4.018409	6767.0
BABY CARE	1	0.000000	0.0	0.000000	0.0
	2	0.000000	0.0	0.049881	84.0
	3	0.000594	1.0	0.399050	672.0
	4	0.000000	0.0	0.014252	24.0
BEAUTY	1	0.134204	226.0	2.408551	4056.0
	2	0.171615	289.0	4.712589	7936.0
	3	0.232185	391.0	9.613420	16189.0
	4	0.159739	269.0	4.091449	6890.0

family	store_nbr	onpromotion		sales	
		mean	sum	mean	sum
AUTOMOTIVE	1	0.008314	14	3.251188	5475.0
	2	0.007720	13	5.403800	9100.0
	3	0.007720	13	9.291568	15647.0
	4	0.005938	10	4.018409	6767.0
BABY CARE	1	0.000000	0	0.000000	0.0
	2	0.000000	0	0.049881	84.0
	3	0.000594	1	0.399050	672.0
	4	0.000000	0	0.014252	24.0
BEAUTY	1	0.134204	226	2.408551	4056.0
	2	0.171615	289	4.712589	7936.0
	3	0.232185	391	9.613420	16189.0
	4	0.159739	269	4.091449	6890.0



PIVOT TABLES VS. GROUPBY

If the column argument isn't specified in a pivot table, it will return a table that's identical to one grouped by the index columns

```
smaller_retail.pivot_table(index=['family', 'store_nbr'],
                           aggfunc={'sum', 'mean'},
                           values=['sales', 'onpromotion'])
```

family	store_nbr	onpromotion		sales	
		mean	sum	mean	sum
AUTOMOTIVE	1	0.008314	14.0	3.251188	5475.0
	2	0.007720	13.0	5.403800	9100.0
	3	0.007720	13.0	9.291568	15647.0
	4	0.005938	10.0	4.018409	6767.0
BABY CARE	1	0.000000	0.0	0.000000	0.0
	2	0.000000	0.0	0.049881	84.0
	3	0.000594	1.0	0.399050	672.0
	4	0.000000	0.0	0.014252	24.0



```
smaller_retail.groupby(['family', 'store_nbr']).agg(
    on_promo_avg="onpromotion", "mean",
    on_promo_sum="onpromotion", "sum",
    sales_avg="sales", "mean"),
    sales_sum="sales", "sum",
)
```

		on_promo_avg	on_promo_sum	sales_avg	sales_sum
family	store_nbr				
AUTOMOTIVE	1	0.008314	14	3.251188	5475.0
	2	0.007720	13	5.403800	9100.0
	3	0.007720	13	9.291568	15647.0
	4	0.005938	10	4.018409	6767.0
BABY CARE	1	0.000000	0	0.000000	0.0
	2	0.000000	0	0.049881	84.0
	3	0.000594	1	0.399050	672.0
	4	0.000000	0	0.014252	24.0
BEAUTY	1	0.134204	226	2.408551	4056.0
	2	0.171615	289	4.712589	7936.0
	3	0.232185	391	9.613420	16189.0
	4	0.159739	269	4.091449	6890.0



PRO TIP: Use groupby if you don't need columns in the pivot, as you can use named aggregations to flatten the column index



PRO TIP: HEATMAPS

You can style a DataFrame based on its values to create a **heatmap**

- Simply chain `.style.background_gradient()` to your DataFrame and add a “cmap” argument

`axis=None` adds a red-yellow-green heatmap to the whole table

```
(smaller_retail.pivot_table(  
    index="family",  
    columns="store_nbr",  
    values="sales",  
    aggfunc="sum"  
).style.background_gradient(cmap="RdYlGn", axis=None))
```

store_nbr	1	2	3	4
family				
AUTOMOTIVE	5475.000000	9100.000000	15647.000000	6767.000000
BABY CARE	0.000000	84.000000	672.000000	24.000000
BEAUTY	4056.000000	7936.000000	16189.000000	6890.000000

`axis=1` adds a red-yellow-green heatmap to each row

```
(  
    smaller_retail.pivot_table(  
        index="family",  
        columns="store_nbr",  
        values="sales",  
        aggfunc="sum"  
    ).style.background_gradient(cmap="RdYlGn", axis=1))
```

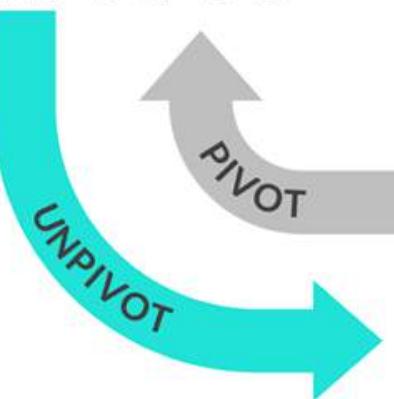
store_nbr	1	2	3	4
family				
AUTOMOTIVE	5475.000000	9100.000000	15647.000000	6767.000000
BABY CARE	0.000000	84.000000	672.000000	24.000000
BEAUTY	4056.000000	7936.000000	16189.000000	6890.000000



MELT

The `.melt()` method will unpivot a DataFrame, or convert columns into rows

country_revenue		2000	2001	2002	2003
	country	151	171	176	184
0	Algeria	151	171	176	184
1	Egypt	204	214	227	241
2	Nigeria	277	301	311	342
3	South Africa	190	211	224	243



	variable	value
0	country	Algeria
1	country	Egypt
2	country	Nigeria
3	country	South Africa
4	2000	151
5	2000	204
6	2000	277
7	2000	190
8	2001	171
9	2001	214
10	2001	301
11	2001	211
12	2002	176
13	2002	227
14	2002	311
15	2002	224
16	2003	184
17	2003	241
18	2003	342
19	2003	243



How does this code work?

- The original column names (`country`, `2000`, etc.) are turned into a single “variable” column
- The values for each original column are placed on a single “value” column next to its corresponding column name



Note that the resulting table isn't perfect, as `.melt()` unpivots a DataFrame around its index, while ideally you'd want to pivot this around the country values



MELT

Use the “id_vars” argument to **specify the column** to unpivot the DataFrame by

country_revenue

	country	2000	2001	2002	2003
0	Algeria	151	171	176	184
1	Egypt	204	214	227	241
2	Nigeria	277	301	311	342
3	South Africa	190	211	224	243

UNPIVOT

country_revenue.melt(id_vars="country")

	country	variable	value
0	Algeria	2000	151
1	Egypt	2000	204
2	Nigeria	2000	277
3	South Africa	2000	190
4	Algeria	2001	171
5	Egypt	2001	214
6	Nigeria	2001	301
7	South Africa	2001	211
8	Algeria	2002	176
9	Egypt	2002	227
10	Nigeria	2002	311
11	South Africa	2002	224
12	Algeria	2003	184
13	Egypt	2003	241
14	Nigeria	2003	342
15	South Africa	2003	243



How does this code work?

- The “id_vars” column (country) is kept in the DataFrame
- The rest of the DataFrame columns are “melted” around the countries in matching variable/value pairs



MELT

You can also select the columns to melt and name the “variable” & “value” columns

country_revenue		2000	2001	2002	2003
	country	2000	2001	2002	2003
0	Algeria	151	171	176	184
1	Egypt	204	214	227	241
2	Nigeria	277	301	311	342
3	South Africa	190	211	224	243

UNPIVOT

```
country_revenue.melt(  
    id_vars="country",  
    value_vars=["2001", "2002", "2003"],  
    var_name="year",  
    value_name="GDP",  
)
```

	country	year	GDP
0	Algeria	2001	171
1	Egypt	2001	214
2	Nigeria	2001	301
3	South Africa	2001	211
4	Algeria	2002	176
5	Egypt	2002	227
6	Nigeria	2002	311
7	South Africa	2002	224
8	Algeria	2003	184
9	Egypt	2003	241
10	Nigeria	2003	342
11	South Africa	2003	243



How does this code work?

- **id_vars** melts the DataFrame around the “country” column
- **value_vars** selects 2001, 2002, and 2003 as the columns to melt (omitting 2000)
- **var_name & value_name** set “year” and “GDP” as column names instead of “variable” and “value”

COMBINING DATAFRAMES



WHY MULTIPLE DATAFRAMES?

There are many scenarios where working with **multiple DataFrames** is necessary:

- Relational Databases save a lot of space by not repeating redundant data (*normalization*)
- Data may come from multiple sources (*like complementing company numbers with external data*)
- Multiple files can be used to store the same data split by different time periods or groups

item_sales.head()

	id	date	store_nbr	family	sales	onpromotion
0	0	2013-01-01	1	AUTOMOTIVE	0.0	0
1	1	2013-01-01	1	BABY CARE	0.0	0
2	2	2013-01-01	1	BEAUTY	0.0	0
3	3	2013-01-01	1	BEVERAGES	0.0	0
4	4	2013-01-01	1	BOOKS	0.0	0

store_transactions.head()

	date	store_nbr	transactions
0	2013-01-01	25	770
1	2013-01-02	1	2111
2	2013-01-02	2	2358
3	2013-01-02	3	3487
4	2013-01-02	4	1922

These two tables come from the same database, so they share the "date" and "store_nbr" columns

If you want to calculate transactions and sales by store, you need to join the tables by their related columns



APPENDING & JOINING

There are two ways to combine DataFrames: **appending** & **joining**

- **Appending** stacks the rows from multiple DataFrames with the same column structure
- **Joining** adds related columns from one DataFrame to another, based on common values

date	store	sales
2022-05-01	1	341
2022-05-01	2	291
2022-05-01	3	493
2022-05-01	4	428
2022-05-01	5	152

date	store	sales
2022-06-01	1	67
2022-06-01	2	144
2022-06-01	3	226
2022-06-01	4	397
2022-06-01	5	163

date	store	sales	store	region
2022-05-01	1	341	1	North
2022-05-01	2	291	2	North
2022-05-01	3	493	3	East
2022-05-01	4	428	4	East
2022-05-01	5	152	5	West

date	store	sales
2022-05-01	1	341
2022-05-01	2	291
2022-05-01	3	493
2022-05-01	4	428
2022-05-01	5	152
2022-06-01	1	67
2022-06-01	2	144
2022-06-01	3	226
2022-06-01	4	397
2022-06-01	5	163

Appending these two tables with the same columns added the rows from one to the other

date	store	sales	region
2022-05-01	1	341	North
2022-05-01	2	291	North
2022-05-01	3	493	East
2022-05-01	4	428	East
2022-05-01	5	152	West

Joining these two tables added the region column based on the matching store values



APPENDING DATAFRAMES

You can **append DataFrames** with the concat() function

- The columns for the DataFrames must be identical
- pd.concat([df_1, df_2]) will stack the rows from df_2 at the bottom of df_1

```
first_5_rows.head()
```

	date	id	store_nbr	family	sales	onpromotion
0	2013-01-01	0	1	AUTOMOTIVE	0.0	0
1	2013-01-01	1	1	BABY CARE	0.0	0
2	2013-01-01	2	1	BEAUTY	0.0	0
3	2013-01-01	3	1	BEVERAGES	0.0	0
4	2013-01-01	4	1	BOOKS	0.0	0

```
next_5_rows.head()
```

	date	id	store_nbr	family	sales	onpromotion
5	2013-01-01	5	1	BREAD/BAKERY	0.0	0
6	2013-01-01	6	1	CELEBRATION	0.0	0
7	2013-01-01	7	1	CLEANING	0.0	0
8	2013-01-01	8	1	DAIRY	0.0	0
9	2013-01-01	9	1	DELI	0.0	0

```
pd.concat([df1, df2, df3, ...])
```

```
pd.concat([first_5_rows, next_5_rows])
```

	date	id	store_nbr	family	sales	onpromotion
0	2013-01-01	0	1	AUTOMOTIVE	0.0	0
1	2013-01-01	1	1	BABY CARE	0.0	0
2	2013-01-01	2	1	BEAUTY	0.0	0
3	2013-01-01	3	1	BEVERAGES	0.0	0
4	2013-01-01	4	1	BOOKS	0.0	0
5	2013-01-01	5	1	BREAD/BAKERY	0.0	0
6	2013-01-01	6	1	CELEBRATION	0.0	0
7	2013-01-01	7	1	CLEANING	0.0	0
8	2013-01-01	8	1	DAIRY	0.0	0
9	2013-01-01	9	1	DELI	0.0	0





JOINING DATAFRAMES

You can **join DataFrames** with the `merge()` function

- The DataFrames must share at least one column to match the values between them

```
left_df.merge(right_df,  
              how,  
              left_on,  
              right_on)
```

The diagram illustrates the `merge()` function call with several annotations:

- A bracket on the left side of the code block groups the first two arguments: `left_df` and `right_df`. An arrow points from this bracket to the text: *"Left" DataFrame to join with "Right"*.
- An arrow points from the argument `how` to the text: *Type of join*.
- An arrow points from the argument `left_on` to the text: *Column(s) on "Left" DataFrame to join by*.
- An arrow points from the argument `right_on` to the text: *Column(s) on "Right" DataFrame to join by*.
- An arrow points from the `right_df` argument to the text: *"Right" DataFrame to join with "Left"*.



JOINING DATAFRAMES

You can **join DataFrames** with the `merge()` function

- The DataFrames must share at least one column to match the values between them

EXAMPLE

Adding the “transactions” by date and store number to the family-level sales table

`item_sales.head()`

	<code>id</code>	<code>date</code>	<code>store_nbr</code>	<code>family</code>	<code>sales</code>	<code>onpromotion</code>
0	0	2013-01-01	1	AUTOMOTIVE	0.0	0
1	1	2013-01-01	1	BABY CARE	0.0	0
2	2	2013-01-01	1	BEAUTY	0.0	0
3	3	2013-01-01	1	BEVERAGES	0.0	0
4	4	2013-01-01	1	BOOKS	0.0	0

`store_transactions.head()`

	<code>date</code>	<code>store_nbr</code>	<code>transactions</code>
0	2013-01-01	25	770
1	2013-01-02	1	2111
2	2013-01-02	2	2358
3	2013-01-02	3	3487
4	2013-01-02	4	1922

What would be the `merge()` arguments to join these two DataFrames?

- The **left DataFrame** is “`item_sales`”
- The **right DataFrame** is “`store_transactions`”
- The **left columns to join by** are “`date`” and “`store_nbr`”
- The **right columns to join by** are “`date`” and “`store_nbr`”
- What about the **join type**?



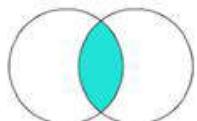
Joining these tables only on “`date`” or “`store_nbr`” **will lead to incorrect matches** if either “`date`” or “`store_nbr`” has more than 1 unique value



TYPES OF JOINS

There are 4 primary **types of joins**: Inner, Left, Right, and Outer

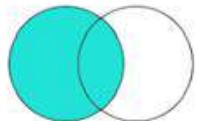
- Inner and Left joins are the most common, while Right is rarely used



Inner

Returns records that exist in BOTH tables, and excludes unmatched records from either table

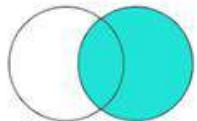
how="inner"



Left

Returns ALL records from the LEFT table, and any matching records from the RIGHT table

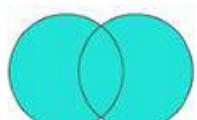
how="left"



Right

Returns ALL records from the RIGHT table, and any matching records from the LEFT table

how="right"



Outer

Returns ALL records from BOTH tables, including non-matching records

how="outer"



TYPES OF JOINS

Left Table

date	store	offer	sales
2022-05-01	1	1	341
2022-05-01	2		291
2022-05-01	3	1	493
2022-05-01	4	2	428
2022-05-01	5		152
2022-06-01	1	3	67
2022-06-01	2		144
2022-06-01	3	1	226
2022-06-01	4		397
2022-06-01	5	4	163

n=10

Right Table

offer	discount
1	5%
2	10%
3	15%
4	20%
5	50%

n=5

Left Join

date	store	offer	sales	discount
2022-05-01	1	1	341	5%
2022-05-01	2		291	
2022-05-01	3	1	493	5%
2022-05-01	4	2	428	10%
2022-05-01	5		152	
2022-06-01	1	3	67	15%
2022-06-01	2		144	
2022-06-01	3	1	226	5%
2022-06-01	4		397	
2022-06-01	5	4	163	20%

n=10

Inner Join

date	store	offer	sales	discount
2022-05-01	1	1	341	5%
2022-05-01	3	1	493	5%
2022-05-01	4	2	428	10%
2022-06-01	1	3	67	15%
2022-06-01	3	1	226	5%
2022-06-01	5	4	163	20%

n=6

Outer Join

date	store	offer	sales	discount
2022-05-01	1	1	341	5%
2022-05-01	2		291	
2022-05-01	3	1	493	5%
2022-05-01	4	2	428	10%
2022-05-01	5		152	
2022-06-01	1	3	67	15%
2022-06-01	2		144	
2022-06-01	3	1	226	5%
2022-06-01	4		397	
2022-06-01	5	4	163	20%
				50%

n=11



EXAMPLE: INNER JOIN

Left Table

item_sales					
	id	date	store_nbr	family	sales
				onpromotion	
0	1945944	2016-01-01	1	AUTOMOTIVE	0.000
1	1945945	2016-01-01	1	BABY CARE	0.000
2	1945946	2016-01-01	1	BEAUTY	0.000
3	1945947	2016-01-01	1	BEVERAGES	0.000
4	1945948	2016-01-01	1	BOOKS	0.000
...
1054939	3000883	2017-08-15	9	POULTRY	438.133
1054940	3000884	2017-08-15	9	PREPARED FOODS	154.553
1054941	3000885	2017-08-15	9	PRODUCE	2419.729
1054942	3000886	2017-08-15	9	SCHOOL AND OFFICE SUPPLIES	121.000
1054943	3000887	2017-08-15	9	SEAFOOD	16.000
1054944 rows x 6 columns					

Right Table

store_transactions		
	date	store_nbr
		transactions
0	2013-01-01	25
1	2013-01-02	1
2	2013-01-02	2
3	2013-01-02	3
4	2013-01-02	4
...
83483	2017-08-15	50
83484	2017-08-15	51
83485	2017-08-15	52
83486	2017-08-15	53
83487	2017-08-15	54
83488 rows x 3 columns		

Inner Join

```
item_sales.merge(  
    store_transactions,  
    how="inner",  
    left_on=["store_nbr", "date"],  
    right_on=["store_nbr", "date"],  
)
```

	id	date	store_nbr	family	sales	onpromotion	transactions
0	1947957	2016-01-02	16	AUTOMOTIVE	8.000	0	373
1	1947958	2016-01-02	16	BABY CARE	0.000	0	373
2	1947959	2016-01-02	16	BEAUTY	0.000	0	373
3	1947960	2016-01-02	16	BEVERAGES	1533.000	0	373
4	1947961	2016-01-02	16	BOOKS	0.000	0	373
...
1026163	3000883	2017-08-15	9	POULTRY	438.133	0	2155
1026164	3000884	2017-08-15	9	PREPARED FOODS	154.553	1	2155
1026165	3000885	2017-08-15	9	PRODUCE	2419.729	148	2155
1026166	3000886	2017-08-15	9	SCHOOL AND OFFICE SUPPLIES	121.000	8	2155
1026167	3000887	2017-08-15	9	SEAFOOD	16.000	0	2155
1026168 rows x 7 columns							

An inner join between these two DataFrames discards all rows where "store_nbr" and "date" don't match

Because '2016-01-01' was NOT in transactions, it is NOT included in the joined table, which leads to a reduced row count



EXAMPLE: LEFT JOIN

Left Table

item_sales					
	id	date	store_nbr	family	sales onpromotion
0	1945944	2016-01-01	1	AUTOMOTIVE	0.000 0
1	1945945	2016-01-01	1	BABY CARE	0.000 0
2	1945946	2016-01-01	1	BEAUTY	0.000 0
3	1945947	2016-01-01	1	BEVERAGES	0.000 0
4	1945948	2016-01-01	1	BOOKS	0.000 0
...
1054939	3000683	2017-08-15	9	POULTRY	438.133 0
1054940	3000684	2017-08-15	9	PREPARED FOODS	154.553 1
1054941	3000685	2017-08-15	9	PRODUCE	2419.729 148
1054942	3000686	2017-08-15	9	SCHOOL AND OFFICE SUPPLIES	121.000 8
1054943	3000687	2017-08-15	9	SEAFOOD	16.000 0

Right Table

store_transactions		
	date	store_nbr transactions
0	2013-01-01	25 770
1	2013-01-02	1 2111
2	2013-01-02	2 2358
3	2013-01-02	3 3487
4	2013-01-02	4 1922
...
83483	2017-08-15	50 2804
83484	2017-08-15	51 1573
83485	2017-08-15	52 2255
83486	2017-08-15	53 932
83487	2017-08-15	54 802

Left Join

item_sales.merge(store_transactions, how="left", left_on=["store_nbr", "date"], right_on=["store_nbr", "date"],)					
	id	date	store_nbr	family	sales onpromotion transactions
0	1945944	2016-01-01	1	AUTOMOTIVE	0.000 0 NaN
1	1945945	2016-01-01	1	BABY CARE	0.000 0 NaN
2	1945946	2016-01-01	1	BEAUTY	0.000 0 NaN
3	1945947	2016-01-01	1	BEVERAGES	0.000 0 NaN
4	1945948	2016-01-01	1	BOOKS	0.000 0 NaN
...
1054939	3000683	2017-08-15	9	POULTRY	438.133 0 2155.0
1054940	3000684	2017-08-15	9	PREPARED FOODS	154.553 1 2155.0
1054941	3000685	2017-08-15	9	PRODUCE	2419.729 148 2155.0
1054942	3000686	2017-08-15	9	SCHOOL AND OFFICE SUPPLIES	121.000 8 2155.0
1054943	3000687	2017-08-15	9	SEAFOOD	16.000 0 2155.0

A left join will preserve all rows in the left table, and simply includes NaN values in "transactions" when no match is found

 **PRO TIP:** If you're performing a join between two tables for the first time, start with a left join to grasp the potential data loss from an inner join and decide which works best for your analysis



THE JOIN METHOD

The `.join()` method can join two DataFrames based on their index

item_sales_short

	date	store_nbr	family	sales
0	2013-01-01	1	PRODUCE	10
1	2013-01-01	2	PRODUCE	20
2	2013-01-02	1	PRODUCE	30
3	2013-01-02	2	PRODUCE	40
4	2013-01-03	1	PRODUCE	50

transactions_short

	date	store_nbr	transactions
0	2012-12-31	1	100
1	2013-01-01	1	200
2	2013-01-01	2	300
3	2013-01-02	1	400
4	2013-01-02	2	500

item_sales_short.join(transactions_short, rsuffix="2")

	date	store_nbr	family	sales	date2	store_nbr2	transactions
0	2013-01-01	1	PRODUCE	10	2012-12-31	1	100
1	2013-01-01	2	PRODUCE	20	2013-01-01	1	200
2	2013-01-02	1	PRODUCE	30	2013-01-01	2	300
3	2013-01-02	2	PRODUCE	40	2013-01-02	1	400
4	2013-01-03	1	PRODUCE	50	2013-01-02	2	500



The "transactions" column was joined based on the index here, not on matches for "date" and "store_nbr"

(`rsuffix="2"` lets Pandas know not to include the first two columns from the right DataFrame – they are duplicates)





THE JOIN METHOD

You can also join based on indices with `.merge()`

- Specify `left_index=True` and `right_index=True`

`item_sales_short`

	date	store_nbr	family	sales
0	2013-01-01	1	PRODUCE	10
1	2013-01-01	2	PRODUCE	20
2	2013-01-02	1	PRODUCE	30
3	2013-01-02	2	PRODUCE	40
4	2013-01-03	1	PRODUCE	50

`item_sales_short.merge(transactions_short, left_index=True, right_index=True)`

	date_x	store_nbr_x	family	sales	date_y	store_nbr_y	transactions
0	2013-01-01	1	PRODUCE	10	2012-12-31	1	100
1	2013-01-01	2	PRODUCE	20	2013-01-01	1	200
2	2013-01-02	1	PRODUCE	30	2013-01-01	2	300
3	2013-01-02	2	PRODUCE	40	2013-01-02	1	400
4	2013-01-03	1	PRODUCE	50	2013-01-02	2	500

`transactions_short`

	date	store_nbr	transactions
0	2012-12-31	1	100
1	2013-01-01	1	200
2	2013-01-01	2	300
3	2013-01-02	1	400
4	2013-01-02	2	500



PRO TIP: You will see examples of joining tables with the `join` method, but it's rare to have perfect alignment of DataFrame indices in practice; since `merge()` can join on indices as well as columns, it is the only method you need to know

CLEANING DATA



DATA CLEANING OVERVIEW

The goal of **data cleaning** is to get raw data into a format that's ready for analysis

This includes:

- Converting columns to the correct **data types** for analysis
- Handling **data issues** that could impact the results of your analysis
- **Creating new columns** from existing columns that are useful for analysis

The order in which you complete these data cleaning tasks will vary by dataset, but this is a good starting point



Even though there are automated tools available, doing some **manual data cleaning** provides a good opportunity to start understanding and getting a good feel for your data



DATA TYPES

When using Pandas to read data, columns are automatically assigned a **data type**

- Use the `.dtypes` attribute to view the data type for each DataFrame column
- Note that sometimes numeric columns (`int`, `float`) and date & time columns (`datetime`) aren't recognized properly by Pandas and get read in as text columns (`object`)

```
df = pd.read_csv('customers.csv')  
df.head()
```

	Name	City	State	Income	Birthdate
0	Susan Rodriguez	Maplewood	NJ	\$45,000	1970-09-12
1	Joseph Martinez	Portland	OR	\$120,000	1960-12-10
2	Joseph Martinez	Portland	OR	\$120,000	1960-12-10
3	Wayne Nielson	Dahlgren	VA	\$75,000	1968-12-05
4	Beverly Nixon	Long Island	NY	\$45,000,000	1962-03-05



Default data types

df.dtypes

Name	object
City	object
State	object
Income	object
Birthdate	object
	dtype: object

These need to be **converted** to be analyzed!

Data Type	Description
bool	Boolean, True/False
int64	Integers
float64	Floating point, decimals
object	Text or mixed values
datetime64	Date and time values



PRO TIP: Use the `.info()` method as an alternative to show additional information along with the data types



CONVERTING TO DATETIME

Use `pd.to_datetime()` to convert object columns into datetime columns

```
df.Birthdate  
0    1970-09-12  
1    1960-12-10  
2    1960-12-10  
3    1968-12-05  
4    1962-03-05  
5    1993-09-23  
6    1996-03-12  
7    1994-01-13  
8    1967-08-27  
9      NaN  
10     NaN  
11     NaN  
12    1968-05-19  
Name: Birthdate, dtype: object
```



```
# convert to a datetime column  
df.Birthdate = pd.to_datetime(df.Birthdate)  
df.Birthdate  
0    1970-09-12  
1    1960-12-10  
2    1960-12-10  
3    1968-12-05  
4    1962-03-05  
5    1993-09-23  
6    1996-03-12  
7    1994-01-13  
8    1967-08-27  
9      NaT  
10     NaT  
11     NaT  
12    1968-05-19  
Name: Birthdate, dtype: datetime64[ns]
```

Note that the missing values
are now NaT instead of NaN
(more on missing data later!)



PRO TIP: Pandas does a pretty good job at detecting the datetime values from an object if the text is in a standard format (like "YYYY-MM-DD"), but you can also manually specify the format using the `format` argument within the `pd.to_datetime()` function: `pd.to_datetime(dt_col, format='%Y-%M-%D')`



CONVERTING TO NUMERIC

Use `pd.to_numeric()` to convert object columns into numeric columns

- To remove non-numeric characters (\$, %, etc.), use `str.replace()`

```
df.Income
```

	Income
0	\$45,000
1	\$120,000
2	\$120,000
3	\$75,000
4	\$45,000,000
5	\$50,000
6	\$105,000
7	\$62,000
8	\$60,000
9	NaN
10	\$80,000
11	NaN
12	\$110,000

Name: Income, dtype: object

```
# remove all dollar signs and commas  
clean_income = df.Income.str.replace('$','').str.replace(',','')
```

	Income
0	45000
1	120000
2	120000
3	75000
4	45000000
5	50000
6	105000
7	62000
8	60000
9	NaN
10	80000
11	NaN
12	110000

```
# convert to a numeric column  
df.Income = pd.to_numeric(clean_income)
```

```
df.dtypes
```

Name	Dtype
Name	object
City	object
State	object
Income	float64
Birthdate	datetime64[ns]

Currency data is read in as an object by Python due to the dollar sign (\$) and comma separator (,)



An alternative is to use `Series.astype()` to convert to more specific data types like 'int', 'float', 'object' and 'bool', but `pd.to_numeric()` can handle missing values (NaN), while `Series.astype()` cannot



DATA ISSUES OVERVIEW

Data issues need to be identified and corrected upfront in order to not impact or skew the results of your analysis

Common “messy” data issues include:

Name	City	State	Income	Birthdate
Susan Rodriguez	Maplewood	NJ	\$45,000	1970-09-12
Joseph Martinez	Portland	OR	\$120,000	1960-12-10
Joseph Martinez	Portland	OR	\$120,000	1960-12-10
Wayne Nielson	Dahlgren	VA	\$75,000	1968-12-05
Beverly Nixon	Long Island	NY	\$45,000,000	1962-03-05
Veronica Comerford	Tuskegee	AL	\$50,000	1993-09-23
Ivan Layton	Albany	New York	\$105,000	1996-03-12
Laura Hailey	Ephraim	Utah	\$62,000	1994-01-13
John Depaul	Baton Rouge	LA	\$60,000	1967-08-27
James Weiss	Houston	TX		
Addie Stevenson	Jacksonville		\$80,000	
Daniel Long				
Queen Watson	Los Angeles	CA	\$110,000	1968-05-19

Duplicates ←

Inconsistent text & typos ←

→ **Outliers**

→ **Missing data**



MISSING DATA

There are various ways to represent **missing data** in Python

- **np.NaN** – NumPy's NaN is the most common representation (*values are stored as floats*)
- **pd.NA** – Pandas' NA is a newer missing data type (*values can be stored as integers*)
- **None** – Base Python's default missing data type (*doesn't allow numerical calculations*)

```
df = pd.read_csv("customers.csv")
df.tail()
```

	Name	City	State	Income	Age
8	John Depaul	Baton Rouge	LA	60000.0	55.0
9	James Weiss	Houston	TX	NaN	NaN
10	Addie Stevenson	Jacksonville	NaN	80000.0	NaN
11	Daniel Long	NaN	NaN	NaN	NaN
12	Queen Watson	Los Angeles	CA	110000.0	54.0

Missing values are treated as **np.NaN** when data is read into Pandas



IDENTIFYING MISSING DATA

The easiest way to identify missing data is with the `.isna()` method

- You can also use `.info()` or `.value_counts(dropna=False)`

```
df.isna()
```

	Name	City	State	Income	Age
0	False	False	False	False	False
1	False	False	False	False	False
2	False	False	False	False	False
3	False	False	False	False	False
4	False	False	False	False	False
5	False	False	False	False	False
6	False	False	False	False	False
7	False	False	False	False	False
8	False	False	False	False	False
9	False	False	False	True	True
10	False	False	True	False	True
11	False	True	True	True	True
12	False	False	False	False	False

```
df.isna().sum()
```

```
Name      0  
City      1  
State     2  
Income    2  
Age       3  
dtype: int64
```

Use `sum()` to return the missing values by column

```
df[df.isna().any(axis=1)]
```

	Name	City	State	Income	Age
9	James Weiss	Houston	TX	NaN	NaN
10	Addie Stevenson	Jacksonville	NaN	80000.0	NaN
11	Daniel Long		NaN	NaN	NaN

Or use `any(axis=1)` to select the rows with missing values

This returns `True` for any missing values



HANDLING MISSING DATA

There are multiple ways to **handle missing data**:

- Keep the missing data as is
- Remove an entire row or column with missing data
- Impute missing numerical data with a 0 or a substitute like the average, mode, etc.
- Resolve the missing data based on your domain expertise

```
df[df.isna().any(axis=1)]
```

	Name	City	State	Income	Age
9	James Weiss	Houston	TX	NaN	NaN
10	Addie Stevenson	Jacksonville	Nan	80000.0	Nan
11	Daniel Long	NaN	Nan	NaN	NaN

We have no data on this customer →

Missing age values are likely ok to keep

This is likely Jacksonville, FL

Can we calculate an approximate income?



There is no right or wrong way to deal with missing data, which is why it's important to **be thoughtful and deliberate** in how you handle it



KEEPING MISSING DATA

You can still perform calculations if you choose to **keep missing data**

```
df
```

	Name	City	State	Income	Age
0	Susan Rodriguez	Maplewood	NJ	450000.0	52.0
1	Joseph Martinez	Portland	OR	1200000.0	62.0
2	Joseph Martinez	Portland	OR	1200000.0	62.0
3	Wayne Nielson	Dahlgren	VA	75000.0	54.0
4	Beverly Nixon	Long Island	NY	45000000.0	61.0
5	Veronica Comerford	Tuskegee	AL	50000.0	29.0
6	Ivan Layton	Albany	New York	105000.0	27.0
7	Laura Hailey	Ephraim	Utah	62000.0	29.0
8	John Depaul	Baton Rouge	LA	60000.0	55.0
9	James Weiss	Houston	TX	NaN	NaN
10	Addie Stevenson	Jacksonville	NaN	80000.0	NaN
11	Daniel Long	NaN	NaN	NaN	NaN
12	Queen Watson	Los Angeles	CA	110000.0	54.0

```
df["Age"].mean()
```

48.5

```
df["Age"].count()
```

10

These ignore the missing values





REMOVING MISSING DATA

The `.dropna()` method removes rows with missing data

df

	Name	City	State	Income	Age
0	Susan Rodriguez	Maplewood	NJ	45000.0	52.0
1	Joseph Martinez	Portland	OR	120000.0	62.0
2	Joseph Martinez	Portland	OR	120000.0	62.0
3	Wayne Nielson	Dahlgren	VA	75000.0	54.0
4	Beverly Nixon	Long Island	NY	45000000.0	61.0
5	Veronica Comerford	Tuskegee	AL	50000.0	29.0
6	Ivan Layton	Albany	New York	105000.0	27.0
7	Laura Hailey	Ephraim	Utah	62000.0	29.0
8	John Depaul	Baton Rouge	LA	60000.0	55.0
9	James Weiss	Houston	TX	NaN	NaN
10	Addie Stevenson	Jacksonville	NaN	80000.0	NaN
11	Daniel Long	NaN	NaN	NaN	NaN
12	Queen Watson	Los Angeles	CA	110000.0	54.0

Our original data set with all of its rows, including all missing values



REMOVING MISSING DATA

The `.dropna()` method removes rows with missing data

Removes any rows with NaN values

```
df.dropna()
```



	Name	City	State	Income	Age
0	Susan Rodriguez	Maplewood	NJ	45000.0	52.0
1	Joseph Martinez	Portland	OR	120000.0	62.0
2	Joseph Martinez	Portland	OR	120000.0	62.0
3	Wayne Nielson	Dahlgren	VA	75000.0	54.0
4	Beverly Nixon	Long Island	NY	45000000.0	61.0
5	Veronica Comerford	Tuskegee	AL	50000.0	29.0
6	Ivan Layton	Albany	New York	105000.0	27.0
7	Laura Hailey	Ephraim	Utah	62000.0	29.0
8	John Depaul	Baton Rouge	LA	60000.0	55.0
12	Queen Watson	Los Angeles	CA	110000.0	54.0

Note that the row index is now skipping values, but you can reset the index with `df.dropna().reset_index()`



REMOVING MISSING DATA

The `.dropna()` method removes rows with missing data

Removes any rows with NaN values

```
df.dropna()
```

Removes rows that only have NaN values

```
df.dropna(how="all")
```



	Name	City	State	Income	Age
0	Susan Rodriguez	Maplewood	NJ	45000.0	52.0
1	Joseph Martinez	Portland	OR	120000.0	62.0
2	Joseph Martinez	Portland	OR	120000.0	62.0
3	Wayne Nielson	Dahlgren	VA	75000.0	54.0
4	Beverly Nixon	Long Island	NY	45000000.0	61.0
5	Veronica Comerford	Tuskegee	AL	50000.0	29.0
6	Ivan Layton	Albany	New York	105000.0	27.0
7	Laura Hailey	Ephraim	Utah	62000.0	29.0
8	John Depaul	Baton Rouge	LA	60000.0	55.0
9	James Weiss	Houston	TX	NaN	NaN
10	Addie Stevenson	Jacksonville	NaN	80000.0	NaN
11	Daniel Long	NaN	NaN	NaN	NaN
12	Queen Watson	Los Angeles	CA	110000.0	54.0



REMOVING MISSING DATA

The `.dropna()` method removes rows with missing data

Removes any rows with NaN values

```
df.dropna()
```

Removes rows that only have NaN values

```
df.dropna(how="all")
```

Removes rows that don't have at least "n" values

```
df.dropna(thresh=2)
```



	Name	City	State	Income	Age
0	Susan Rodriguez	Maplewood	NJ	45000.0	52.0
1	Joseph Martinez	Portland	OR	120000.0	62.0
2	Joseph Martinez	Portland	OR	120000.0	62.0
3	Wayne Nielson	Dahlgren	VA	75000.0	54.0
4	Beverly Nixon	Long Island	NY	45000000.0	61.0
5	Veronica Comerford	Tuskegee	AL	50000.0	29.0
6	Ivan Layton	Albany	New York	105000.0	27.0
7	Laura Hailey	Ephraim	Utah	62000.0	29.0
8	John Depaul	Baton Rouge	LA	60000.0	55.0
9	James Weiss	Houston	TX	NaN	NaN
10	Addie Stevenson	Jacksonville	NaN	80000.0	NaN
12	Queen Watson	Los Angeles	CA	110000.0	54.0



REMOVING MISSING DATA

The `.dropna()` method removes rows with missing data

Removes any rows with NaN values

```
df.dropna()
```

Removes rows that only have NaN values

```
df.dropna(how="all")
```

Removes rows that don't have at least "n" values

```
df.dropna(thresh=2)
```

Removes rows with NaN values in a specified column

```
df.dropna(subset=["City"])
```

	Name	City	State	Income	Age
0	Susan Rodriguez	Maplewood	NJ	45000.0	52.0
1	Joseph Martinez	Portland	OR	120000.0	62.0
2	Joseph Martinez	Portland	OR	120000.0	62.0
3	Wayne Nielson	Dahlgren	VA	75000.0	54.0
4	Beverly Nixon	Long Island	NY	45000000.0	61.0
5	Veronica Comerford	Tuskegee	AL	50000.0	29.0
6	Ivan Layton	Albany	New York	105000.0	27.0
7	Laura Hailey	Ephraim	Utah	62000.0	29.0
8	John Depaul	Baton Rouge	LA	60000.0	55.0
9	James Weiss	Houston	TX	NaN	NaN
10	Addie Stevenson	Jacksonville	NaN	80000.0	NaN
12	Queen Watson	Los Angeles	CA	110000.0	54.0





PRO TIP: KEEPING NON-MISSING DATA

You can use `.notna()` to keep non-missing data instead

```
df[df["City"].notna()]
```



Note that using `.dropna()` or `.notna()` to remove rows with missing data **does not make permanent changes** to the DataFrame, so you need to save the output to a new DataFrame (or the same one) or set the argument `inplace=True`

	Name	City	State	Income	Age
0	Susan Rodriguez	Maplewood	NJ	45000.0	52.0
1	Joseph Martinez	Portland	OR	120000.0	62.0
2	Joseph Martinez	Portland	OR	120000.0	62.0
3	Wayne Nielson	Dahlgren	VA	75000.0	54.0
4	Beverly Nixon	Long Island	NY	45000000.0	61.0
5	Veronica Comerford	Tuskegee	AL	50000.0	29.0
6	Ivan Layton	Albany	New York	105000.0	27.0
7	Laura Hailey	Ephraim	Utah	62000.0	29.0
8	John Depaul	Baton Rouge	LA	60000.0	55.0
9	James Weiss	Houston	TX	NaN	NaN
10	Addie Stevenson	Jacksonville	NaN	80000.0	NaN
12	Queen Watson	Los Angeles	CA	110000.0	54.0



IMPUTING MISSING DATA

The `.fillna()` method imputes missing data with an appropriate value

- There are many ways to impute data in a column (zero, mean, mode, etc.), so take a moment to decide the best one – if you’re unsure, reach out to a subject matter expert

```
df[["Income"]]
```



```
df["Income"] = df["Income"].fillna(df["Income"].median())
df
```

	Income
0	45000.0
1	120000.0
2	120000.0
3	75000.0
4	45000000.0
5	50000.0
6	105000.0
7	62000.0
8	60000.0
9	NaN
10	80000.0
11	110000.0

← What value can
be used here?

	Name	City	State	Income	Age
0	Susan Rodriguez	Maplewood	NJ	45000.0	52.0
1	Joseph Martinez	Portland	OR	120000.0	62.0
2	Joseph Martinez	Portland	OR	120000.0	62.0
3	Wayne Nielson	Dahlgren	VA	75000.0	54.0
4	Beverly Nixon	Long Island	NY	45000000.0	61.0
5	Veronica Comerford	Tuskegee	AL	50000.0	29.0
6	Ivan Layton	Albany	New York	105000.0	27.0
7	Laura Hailey	Ephraim	Utah	62000.0	29.0
8	John Depaul	Baton Rouge	LA	60000.0	55.0
9	James Weiss	Houston	TX	80000.0	NaN
10	Addie Stevenson	Jacksonville	NaN	80000.0	NaN
11	Queen Watson	Los Angeles	CA	110000.0	54.0

Using the median removes
the impact of the outlier



RESOLVING MISSING DATA

An alternative to imputing is to **resolve missing data** with domain expertise

- You can use the `.loc[]` accessor to update a specific value

df

	Name	City	State	Income	Age
0	Susan Rodriguez	Maplewood	NJ	45000.0	52.0
1	Joseph Martinez	Portland	OR	120000.0	62.0
2	Joseph Martinez	Portland	OR	120000.0	62.0
3	Wayne Nielson	Dahlgren	VA	75000.0	54.0
4	Beverly Nixon	Long Island	NY	45000000.0	61.0
5	Veronica Comerford	Tuskegee	AL	50000.0	29.0
6	Ivan Layton	Albany	New York	105000.0	27.0
7	Laura Hailey	Ephraim	Utah	62000.0	29.0
8	John Depaul	Baton Rouge	LA	60000.0	55.0
9	James Weiss	Houston	TX	80000.0	NaN
10	Addie Stevenson	Jacksonville	NaN	80000.0	NaN
11	Queen Watson	Los Angeles	CA	110000.0	54.0



```
df.loc[10, "State"] = "FL"  
df
```

	Name	City	State	Income	Age
0	Susan Rodriguez	Maplewood	NJ	45000.0	52.0
1	Joseph Martinez	Portland	OR	120000.0	62.0
2	Joseph Martinez	Portland	OR	120000.0	62.0
3	Wayne Nielson	Dahlgren	VA	75000.0	54.0
4	Beverly Nixon	Long Island	NY	45000000.0	61.0
5	Veronica Comerford	Tuskegee	AL	50000.0	29.0
6	Ivan Layton	Albany	New York	105000.0	27.0
7	Laura Hailey	Ephraim	Utah	62000.0	29.0
8	John Depaul	Baton Rouge	LA	60000.0	55.0
9	James Weiss	Houston	TX	80000.0	NaN
10	Addie Stevenson	Jacksonville	FL	80000.0	NaN
11	Queen Watson	Los Angeles	CA	110000.0	54.0



INCONSISTENT TEXT & TYPOS

Inconsistent text & typos in a data set are represented by values that are either:

- Incorrect by a few digits or characters
- Inconsistent with the rest of a column

```
df.head(8)
```

	Name	City	State	Income	Age
0	Susan Rodriguez	Maplewood	NJ	45000.0	52.0
1	Joseph Martinez	Portland	OR	120000.0	62.0
2	Joseph Martinez	Portland	OR	120000.0	62.0
3	Wayne Nielson	Dahlgren	VA	75000.0	54.0
4	Beverly Nixon	Long Island	NY	45000000.0	61.0
5	Veronica Comerford	Tuskegee	AL	50000.0	29.0
6	Ivan Layton	Albany	New York	105000.0	27.0
7	Laura Hailey	Ephraim	Utah	62000.0	29.0



Finding inconsistent text and typos within a large data set in Python is **not a straightforward approach**, as there is no function that will automatically identify these situations

These are full state names, while the rest of the column has abbreviations



IDENTIFYING INCONSISTENT TEXT & TYPOS

While there is no specific method to **identify inconsistent text & typos**, you can take the following two approaches to check a column depending on its data type:

Categorical data

Look at the unique values in the column

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

OR	2
NJ	1
VA	1
NY	1
AL	1
New York	1
Utah	1
LA	1
TX	1
FL	1
CA	1

Name: State, dtype: int64

These represent
the same thing!

Numerical data

Look at the descriptive stats of the column

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

count	10.000000
mean	48.500000
std	14.370108
min	27.000000
25%	34.750000
50%	54.000000
75%	59.500000
max	62.000000

Name: Age, dtype: float64

This looks like a
realistic age range



HANDLING INCONSISTENT TEXT & TYPOS

You can fix **inconsistent text & typos** by using:

- 1 `.loc[]` to update a value at a particular location
- 2 `np.where()` to update values in a column based on a conditional statement
- 3 `.map()` to map a set of values to another set of values
- 4 **String methods** like `str.lower()`, `str.strip()` & `str.replace()` to clean text data



We've already covered the **loc[] accessor** to resolve missing data using domain expertise



UPDATE VALUES BASED ON A LOGICAL CONDITION

Use `np.where()` to update values based on a logical condition

```
np.where(condition, if_true, if_false)
```

Calls the
NumPy library

A logical expression that
evaluates to True or False

Value to return when
the expression is True

Value to return when
the expression is False



This is **different from the Pandas where method**, which has similar functionality, but different syntax

The NumPy function is used more often than the Pandas method because `np.where` is vectorized, meaning it executes faster



UPDATE VALUES BASED ON A LOGICAL CONDITION

Use `np.where()` to update values based on a logical condition

df

	Name	City	State	Income	Birthdate
0	Susan Rodriguez	Maplewood	NJ	45000.0	1970-09-12
1	Joseph Martinez	Portland	OR	120000.0	1960-12-10
2	Joseph Martinez	Portland	OR	120000.0	1960-12-10
3	Wayne Nielson	Dahlgren	VA	75000.0	1968-12-05
4	Beverly Nixon	Long Island	NY	45000000.0	1962-03-05
5	Veronica Comerford	Tuskegee	AL	50000.0	1993-09-23
6	Ivan Layton	Albany	New York	105000.0	1996-03-12
7	Laura Hailey	Ephraim	Utah	62000.0	1994-01-13

If a State value is equal to 'Utah', then replace it with 'UT'. Otherwise, keep the value that was already in the State column

```
import numpy as np  
  
df.State = np.where(df.State == 'Utah', 'UT', df.State)  
df
```

	Name	City	State	Income	Birthdate
0	Susan Rodriguez	Maplewood	NJ	45000.0	1970-09-12
1	Joseph Martinez	Portland	OR	120000.0	1960-12-10
2	Joseph Martinez	Portland	OR	120000.0	1960-12-10
3	Wayne Nielson	Dahlgren	VA	75000.0	1968-12-05
4	Beverly Nixon	Long Island	NY	45000000.0	1962-03-05
5	Veronica Comerford	Tuskegee	AL	50000.0	1993-09-23
6	Ivan Layton	Albany	New York	105000.0	1996-03-12
7	Laura Hailey	Ephraim	UT	62000.0	1994-01-13



MAP VALUES

Use `.map()` to map values from one set of values to another set of values

```
# map multiple state values to one abbreviation  
state_mapping = {'AL': 'AL',  
                 'Alabama': 'AL',  
                 'NY': 'NY',  
                 'New York': 'NY'}
```

You can pass a dictionary with existing values as the keys, and new values as the values

```
# use the map method to create a new column  
df['State_Clean'] = df.State.map(state_mapping)  
df
```

	Name	City	State	Income	Birthdate	State_Clean
4	Beverly Nixon	Long Island	NY	45000000.0	1962-03-05	NY
5	Veronica Comerford	Tuskegee	AL	50000.0	1993-09-23	AL
6	Ivan Layton	Albany	New York	105000.0	1996-03-12	NY

These states were mapped to these state abbreviations



This is similar to creating a **lookup table** in Excel and using VLOOKUP to search for a value in a column of the table and retrieve a corresponding value from another column



PRO TIP: CLEANING TEXT

String methods are commonly used to **clean text** and standardize it for analysis

run_times				
	Name	Run Time	Warm Up Time	Location
0	Alexis	9.2343	3.5	"school"
1	Alexis	10.3842	3.5	School
2	Alexis	8.1209	3 min	"the gym"
3	David	7.2123	2.2	"school"
4	David	6.8342	2	"gym"



```
# strip away the leading and trailing characters  
run_times.Location.str.strip('\"'")
```

```
0    school  
1    School  
2    the gym  
3    school  
4    gym  
Name: Location, dtype: object
```

```
# make everything lowercase  
run_times.Location.str.strip('\"'").str.lower()
```

```
0    school  
1    school  
2    the gym  
3    school  
4    gym  
Name: Location, dtype: object
```

```
# replace or remove characters  
run_times.Location.str.strip('\"'").str.lower().str.replace('the ', '')
```

```
0    school  
1    school  
2    gym  
3    school  
4    gym  
Name: Location, dtype: object
```



There is a lot more you can do to clean text data, which will be covered in the **Natural Language Processing** course



DUPLICATE DATA

Duplicate data represents the presence of one or more redundant rows that contain the same information as another, and can therefore be removed

	Name	City	State	Income	Age
0	Susan Rodriguez	Maplewood	NJ	45000.0	52.0
1	Joseph Martinez	Portland	OR	120000.0	62.0
2	Joseph Martinez	Portland	OR	120000.0	62.0
3	Wayne Nielson	Dahlgren	VA	75000.0	54.0
4	Beverly Nixon	Long Island	NY	45000000.0	61.0
5	Veronica Comerford	Tuskegee	AL	50000.0	29.0
6	Ivan Layton	Albany	NY	105000.0	27.0
7	Laura Hailey	Ephraim	UT	62000.0	29.0
8	John Depaul	Baton Rouge	LA	60000.0	55.0
9	James Weiss	Houston	TX	80000.0	NaN
10	Addie Stevenson	Jacksonville	FL	80000.0	NaN
11	Queen Watson	Los Angeles	CA	110000.0	54.0

This is duplicate data
on the same customer

These are the same values,
but they're not considered
duplicate data



IDENTIFYING DUPLICATE DATA

Use `.duplicated()` to identify duplicate rows of data

	Name	City	State	Income	Age
0	Susan Rodriguez	Maplewood	NJ	45000.0	52.0
1	Joseph Martinez	Portland	OR	120000.0	62.0
2	Joseph Martinez	Portland	OR	120000.0	62.0
3	Wayne Nielson	Dahlgren	VA	75000.0	54.0
4	Beverly Nixon	Long Island	NY	45000000.0	61.0
5	Veronica Comerford	Tuskegee	AL	50000.0	29.0
6	Ivan Layton	Albany	NY	105000.0	27.0
7	Laura Hailey	Ephraim	UT	62000.0	29.0
8	John Depaul	Baton Rouge	LA	60000.0	55.0
9	James Weiss	Houston	TX	80000.0	NaN
10	Addie Stevenson	Jacksonville	FL	80000.0	NaN
11	Queen Watson	Los Angeles	CA	110000.0	54.0



`df.duplicated()`



`df.duplicated().sum()`

```
0  False  
1  False  
2  True  
3  False  
4  False  
5  False  
6  False  
7  False  
8  False  
9  False  
10 False  
11 False  
dtype: bool
```

This returns True for every row that is a duplicate of a previous row

`df[df.duplicated(keep=False)]`

	Name	City	State	Income	Age
1	Joseph Martinez	Portland	OR	120000.0	62.0
2	Joseph Martinez	Portland	OR	120000.0	62.0

You can use `keep=False` to return all the duplicate rows



REMOVING DUPLICATE DATA

Use `.drop_duplicates()` to remove duplicate rows of data

```
df.drop_duplicates()
```

You may need to reset the index →

	Name	City	State	Income	Age
0	Susan Rodriguez	Maplewood	NJ	45000.0	52.0
1	Joseph Martinez	Portland	OR	120000.0	62.0
3	Wayne Nielson	Dahlgren	VA	75000.0	54.0
4	Beverly Nixon	Long Island	NY	45000000.0	61.0
5	Veronica Comerford	Tuskegee	AL	50000.0	29.0
6	Ivan Layton	Albany	NY	105000.0	27.0
7	Laura Hailey	Ephraim	UT	62000.0	29.0
8	John Depaul	Baton Rouge	LA	60000.0	55.0
9	James Weiss	Houston	TX	80000.0	NaN
10	Addie Stevenson	Jacksonville	FL	80000.0	NaN
11	Queen Watson	Los Angeles	CA	110000.0	54.0



OUTLIERS

An **outlier** is a value in a data set that is much bigger or smaller than the others

df

	Name	City	State	Income	Age
0	Susan Rodriguez	Maplewood	NJ	45000.0	52.0
1	Joseph Martinez	Portland	OR	120000.0	62.0
2	Wayne Nielson	Dahlgren	VA	75000.0	54.0
3	Beverly Nixon	Long Island	NY	45000000.0	61.0
4	Veronica Comerford	Tuskegee	AL	50000.0	29.0
5	Ivan Layton	Albany	NY	105000.0	27.0
6	Laura Hailey	Ephraim	UT	62000.0	29.0
7	John Depaul	Baton Rouge	LA	60000.0	55.0
8	James Weiss	Houston	TX	80000.0	NaN
9	Addie Stevenson	Jacksonville	FL	80000.0	NaN
10	Queen Watson	Los Angeles	CA	110000.0	54.0

Average income (**including** outlier) = **\$4.1M**

Average income (**excluding** outlier) = **\$82K**



If outliers are not identified and dealt with, they can have a **notable impact** on calculations and models



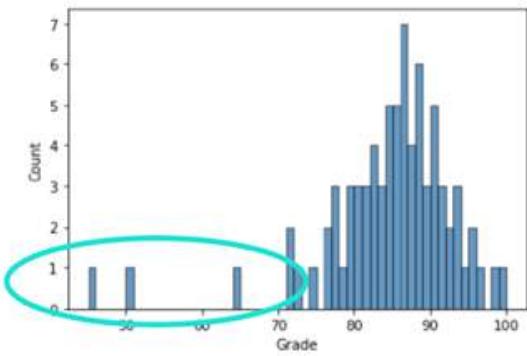
IDENTIFYING OUTLIERS

You can **identify outliers** in different ways using plots and statistics

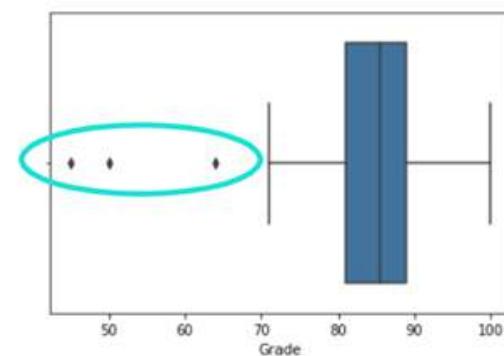
EXAMPLE

Identifying outliers in student grades from a college class

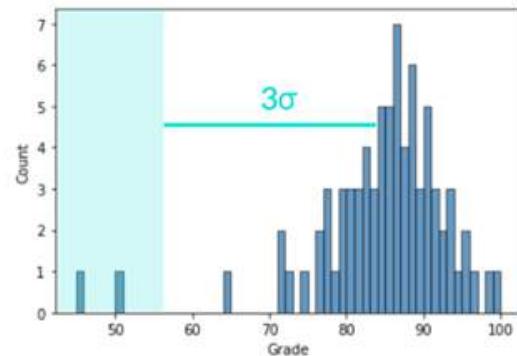
Histogram



Box Plot



Standard Deviation



It's also important to define what you'll consider an outlier in each scenario



Should we flag 3 or 2 outliers?

Use your domain expertise!



HISTOGRAMS

Histograms are used to visualize the distribution (or shape) of a numerical column

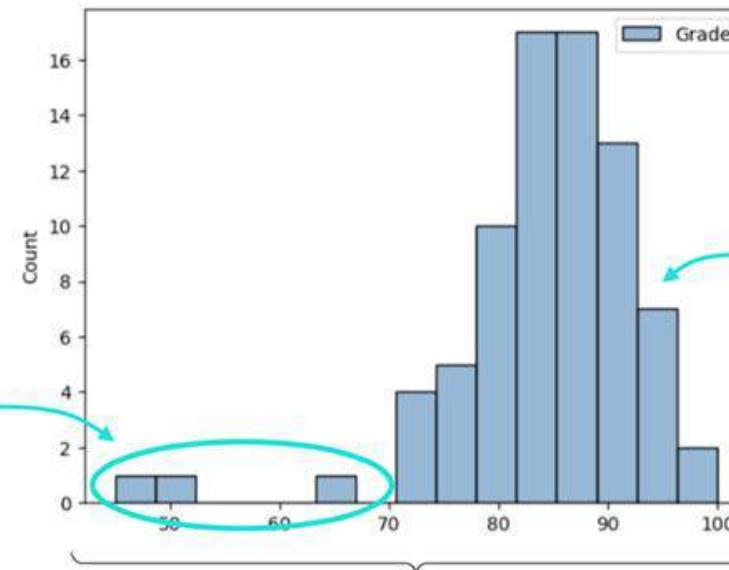
- They help identify outliers by showing which values fall outside of the normal range

```
df = pd.read_csv("student_grades.csv")  
df
```

	Student	Grade
0	Emma	86
1	Olivia	86
2	Noah	86
3	Sophia	87
4	Liam	90
...
73	Zoey	91
74	Aaron	85
75	Charles	93
76	Connor	91
77	Riley	87

```
import seaborn as sns  
sns.histplot(df);
```

You can create them with the **seaborn** library



These are the potential outliers

This is the range of values in the data set

The height of each bar is how often the value occurs

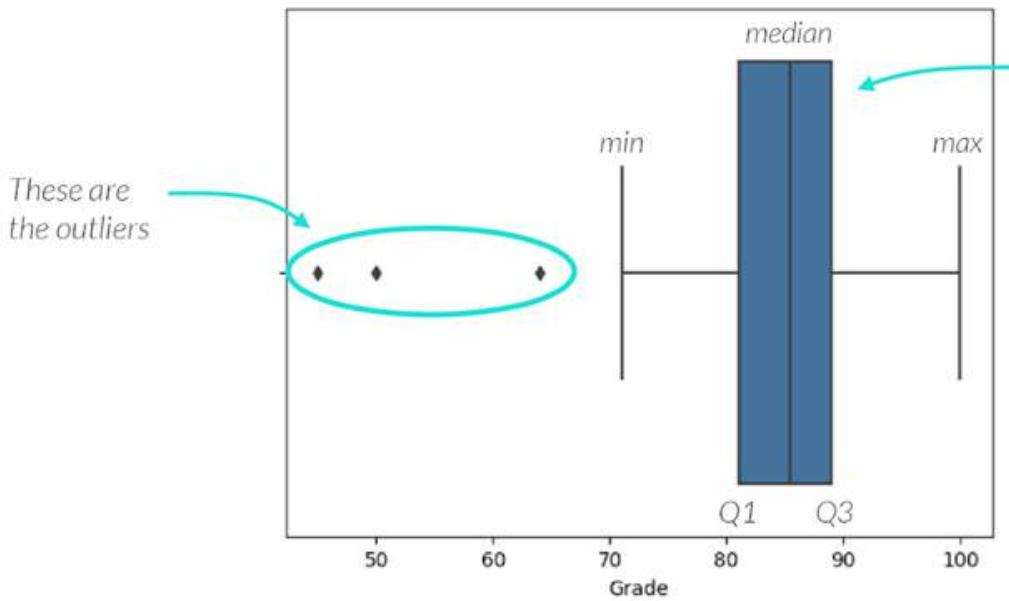


BOXPLOTS

Boxplots are used to visualize the descriptive statistics of a numerical column

- They automatically plot outliers as dots outside of the min/max data range

```
sns.boxplot(x=df.Grade);
```



These are
the outliers

The width of the "box" is the **interquartile range** (IQR), which is the middle 50% of the data

Any value farther away than $1.5 \times \text{IQR}$ from each side of the box is considered an outlier



STANDARD DEVIATION

The **standard deviation** is a measure of the spread of a data set from the mean



$$\text{Standard Deviation} = 8.2$$



$$\text{Standard Deviation} = 5.8$$



STANDARD DEVIATION

The **standard deviation** is a measure of the spread of a data set from the mean

Values at least 3 standard deviations away from the mean are considered outliers

- This is meant for normally distributed, or bell shaped, data
- The threshold of 3 standard deviations can be changed to 2 or 4+ depending on the data

```
import numpy as np  
  
mean = np.mean(df.Grade)  
sd = np.std(df.Grade)  
  
mean, sd
```

```
(84.08987341772152, 8.723725033779411)
```

```
[grade for grade in df.Grade if (grade < mean - 3*sd) or (grade > mean + 3*sd)]
```

```
[50, 45]
```

 This returns a list of values, or outliers, at least 3 standard deviations away from the mean



HANDLING OUTLIERS

Like with missing data, there are multiple ways to **handle outliers**:

- Keep outliers
- Remove an entire row or column with outliers
- Impute outliers with NaN or a substitute like the average, mode, max, etc.
- Resolve outliers based on your domain expertise

```
df.head()
```

	Name	City	State	Income	Age
0	Susan Rodriguez	Maplewood	NJ	45000.0	52.0
1	Joseph Martinez	Portland	OR	120000.0	62.0
2	Wayne Nielson	Dahlgren	VA	75000.0	54.0
3	Beverly Nixon	Long Island	NY	45000000.0	61.0
4	Veronica Comerford	Tuskegee	AL	50000.0	29.0



How would you handle this?



CREATING NEW COLUMNS

After cleaning data types & issues, you may still not have the exact data that you need, so you can **create new columns** from existing data to aid your analysis

- Numeric columns – calculating percentages, applying conditional calculations, etc.
- Datetime columns – extracting datetime components, applying datetime calculations, etc.
- Text columns – extracting text, splitting into multiple columns, finding patterns, etc.

Name	Cost	Date	Notes
Alexis	\$0	4/15/23	Coach: great job!
Alexis	\$0	4/22/23	Coach: keep it up
Alexis	\$25	5/10/23	PT: add strength training
David	\$20	5/1/23	Trainer: longer warm up
David	\$20	5/10/23	Trainer: pace yourself

Add 8% tax
Extract the month

Split into two columns

Name	Cost + Tax	Month	Person	Note
Alexis	\$0	April	Coach	great job!
Alexis	\$0	April	Coach	keep it up
Alexis	\$27.00	May	PT	add strength training
David	\$21.60	May	Trainer	longer warm up
David	\$21.60	May	Trainer	pace yourself

Data is ready for further analysis!



CALCULATING PERCENTAGES

To **calculate a percentage**, you can set up two columns with the numerator and denominator values and then divide them (you can also multiply by 100 if desired)

EXAMPLE

Finding the percentage of total spend for each item

shopping_list

	Category	Item	Price
0	Fruit	Apple	1.29
1	Fruit	Banana	0.40
2	Fruit	Grapes	3.99
3	Vegetable	Carrots	1.50
4	Vegetable	Celery	1.99

This will be the **numerator**



CALCULATING PERCENTAGES

To **calculate a percentage**, you can set up two columns with the numerator and denominator values and then divide them (you can also multiply by 100 if desired)

EXAMPLE

Finding the percentage of total spend for each item

```
shopping_list
```

```
# calculate the total amount spent
shopping_list['Total Spend'] = shopping_list['Price'].sum()
shopping_list
```

	Category	Item	Price	Total Spend
0	Fruit	Apple	1.29	9.17
1	Fruit	Banana	0.40	9.17
2	Fruit	Grapes	3.99	9.17
3	Vegetable	Carrots	1.50	9.17
4	Vegetable	Celery	1.99	9.17

sum = 9.17

This will be the denominator



CALCULATING PERCENTAGES

To **calculate a percentage**, you can set up two columns with the numerator and denominator values and then divide them (you can also multiply by 100 if desired)

EXAMPLE

Finding the percentage of total spend for each item

```
shopping_list
```

```
# calculate the total amount spent
shopping_list['Total Spend'] = shopping_list['Price'].sum()
shopping_list
```

```
# calculate the percent spent
shopping_list['Percent Spend'] = shopping_list['Price'] / total_spend * 100
shopping_list
```

	Category	Item	Price	Total Spend	Percent Spend
0	Fruit	Apple	1.29	9.17	14.067612
1	Fruit	Banana	0.40	9.17	4.362050
2	Fruit	Grapes	3.99	9.17	43.511450
3	Vegetable	Carrots	1.50	9.17	16.357688
4	Vegetable	Celery	1.99	9.17	21.701200

These add up to 100%



CALCULATING BASED ON A CONDITION

Use `np.where()` to create a new column based on a logical condition

run_times

	Name	Run Time	Warm Up Time	Location	Run Date	Race Date	Rain	Fee
0	Alexis	9.2343	3.5	school	2023-04-15	2023-06-01	False	0.0
1	Alexis	10.3842	3.5	school	2023-04-22	2023-06-01	True	0.0
2	Alexis	8.1209	3	gym	2023-05-10	2023-06-01	False	2.5
3	David	7.2123	2.2	school	2023-05-01	2023-06-15	False	0.0
4	David	6.8342	2	gym	2023-05-10	2023-06-15	False	2.5

```
# update all gym fees to include tax
run_times['Fee with Tax'] = np.where(run_times.Location == 'gym', run_times.Fee * 1.08, run_times.Fee)
run_times
```

	Name	Run Time	Warm Up Time	Location	Run Date	Race Date	Rain	Fee	Fee with Tax
0	Alexis	9.2343	3.5	school	2023-04-15	2023-06-01	False	0.0	0.0
1	Alexis	10.3842	3.5	school	2023-04-22	2023-06-01	True	0.0	0.0
2	Alexis	8.1209	3	gym	2023-05-10	2023-06-01	False	2.5	2.7
3	David	7.2123	2.2	school	2023-05-01	2023-06-15	False	0.0	0.0
4	David	6.8342	2	gym	2023-05-10	2023-06-15	False	2.5	2.7

If Location is equal to 'gym', then increase the Fee by 8% in the Fee with Tax column
Otherwise, set it to the existing Fee



EXTRACTING DATETIME COMPONENTS

Use **dt.component** to extract a component from a datetime value (day, month, etc.)

run_times

	Run Date	Race Date
0	2023-04-15 12:00:00	2023-06-01
1	2023-04-22 12:30:00	2023-06-01
2	2023-05-10 15:00:00	2023-06-01
3	2023-05-01 15:15:00	2023-06-15
4	2023-05-10 16:30:00	2023-06-15

extract the day from the date
run_times['Run Date'].dt.day

```
0    15  
1    22  
2    10  
3     1  
4    10  
Name: Run Date, dtype: int64
```

extract the day of the week
run_times['Run Date'].dt.dayofweek

```
0    5  
1    5  
2    2  
3    0  
4    2  
Name: Run Date, dtype: int64
```

extract the time
run_times['Run Date'].dt.time

```
0    12:00:00  
1    12:30:00  
2    15:00:00  
3    15:15:00  
4    16:30:00  
Name: Run Date, dtype: object
```

Component

Output

dt.date	Date (without time component)
dt.year	Year
dt.month	Numeric month (1-12)
dt.day	Day of the month
dt.dayofweek	Numeric weekday (Mon=0, Sun=6)
dt.time	Time (without date component)
dt.hour	Hour (0-23)
dt.minute	Minute (0-59)
dt.second	Second (0-59)



DATETIME CALCULATIONS

Datetime calculations between columns can be done using basic arithmetic

- Use `pd.to_timedelta()` to add or subtract a particular timeframe

```
# calculate the difference between two dates  
run_times['Race Date'] - run_times['Run Date']
```

```
0    46 days 12:00:00  
1    39 days 11:30:00  
2    21 days 09:00:00  
3    44 days 08:45:00  
4    35 days 07:30:00  
dtype: timedelta64[ns] ← Note that the data type changed
```

```
# add two weeks to the race date  
run_times['Race Date'] + pd.to_timedelta(2, unit='W')
```

```
0    2023-06-15  
1    2023-06-15  
2    2023-06-15  
3    2023-06-29  
4    2023-06-29  
Name: Race Date, dtype: datetime64[ns]
```

Time delta units:

- D = day
- W = week
- H = hour
- T = minute
- S = second



EXTRACTING TEXT

You can use `.str[start:end]` to extract characters from a text field

- Note that the position of each character in a string is 0-indexed, and the “end” is non-inclusive

run_notes

notes

0	Day 1 - Starting off: Congrats on starting you...
1	Day 2 - Progressing: On your second day, you m...
2	Day 3 - Finding a rhythm: By day 3, you may ha...
3	Day 4 - Pushing yourself: On day 4, you may ha...
4	Day 5 - Consistency: On your final day, you sh...



first 6 characters
run_notes.notes.str[:6]

0	Day 1
1	Day 2
2	Day 3
3	Day 4
4	Day 5

Name: notes, dtype: object

The blank starts from 0 by default

last 11 characters
run_notes.notes.str[-11:]

0	endurance.
1	Keep it up!
2	your runs.
3	endurance.
4	Keep it up!

Name: notes, dtype: object

The negative grabs the “start” from the end of the list, and the blank “end” goes to the end of the text string



SPLITTING INTO MULTIPLE COLUMNS

Use `str.split()` to split a column by a delimiter into multiple columns

run_notes

notes

- 0 Day 1 - Starting off: Congrats on starting you...
- 1 Day 2 - Progressing: On your second day, you m...
- 2 Day 3 - Finding a rhythm: By day 3, you may ha...
- 3 Day 4 - Pushing yourself: On day 4, you may ha...
- 4 Day 5 - Consistency: On your final day, you sh...



```
# split on the '-' character  
two_fields = run_notes.notes.str.split('-')  
two_fields
```

```
0 [Day 1 , Starting off: Congrats on starting y...  
1 [Day 2 , Progressing: On your second day, you...  
2 [Day 3 , Finding a rhythm: By day 3, you may ...  
3 [Day 4 , Pushing yourself: On day 4, you may ...  
4 [Day 5 , Consistency: On your final day, you ...  
Name: notes, dtype: object
```

Splitting text
returns a list

```
# turn list series into dataframe  
pd.DataFrame(two_fields.to_list(),  
             columns=['Day', 'Notes'])
```

	Day	Notes
0	Day 1	Starting off: Congrats on starting your runni...
1	Day 2	Progressing: On your second day, you may have...
2	Day 3	Finding a rhythm: By day 3, you may have foun...
3	Day 4	Pushing yourself: On day 4, you may have felt...
4	Day 5	Consistency: On your final day, you should fe...

This is now a
DataFrame with
two columns



FINDING PATTERNS

Use `str.contains()` to find words or patterns within a text field

```
# create a new column that says whether or not
# the note contains the term 'final'
run_notes['Contains final'] = run_notes.notes.str.contains('final')
run_notes
```

	notes	Contains final
0	Day 1 - Starting off: Congrats on starting you...	False
1	Day 2 - Progressing: On your second day, you m...	False
2	Day 3 - Finding a rhythm: By day 3, you may ha...	False
3	Day 4 - Pushing yourself: On day 4, you may ha...	False
4	Day 5 - Consistency: On your final day, you sh...	True

```
# contains great or congrats
run_notes.notes.str.contains('great|congrats', regex=True)
```

```
0    False
1    False
2     True
3     True
4    False
Name: notes, dtype: bool
```



Regex stands for **regular expression**, which is a way of finding patterns within text (more on this topic will be covered in the Natural Language Processing course)

EXPLORATORY DATA ANALYSIS

EXPLORATORY DATA ANALYSIS



In this section we'll cover **exploratory data analysis** (EDA), which includes a variety of techniques used to better understand a dataset and discover hidden patterns & insights

TOPICS WE'LL COVER:

[EDA Overview](#)

[Exploring Data](#)

[Visualizing Data](#)

[EDA Tips](#)

GOALS FOR THIS SECTION:

- Learn Python techniques for exploring a new data set, like filtering, sorting, grouping, and visualizing
- Practice finding patterns and drawing insights from data by exploring it from multiple angles
- Understand that while EDA focuses on technical aspects, the goal is to get a good feel for the data



EXPLORATORY DATA ANALYSIS

The **exploratory data analysis** (EDA) phase gives data scientists a chance to:

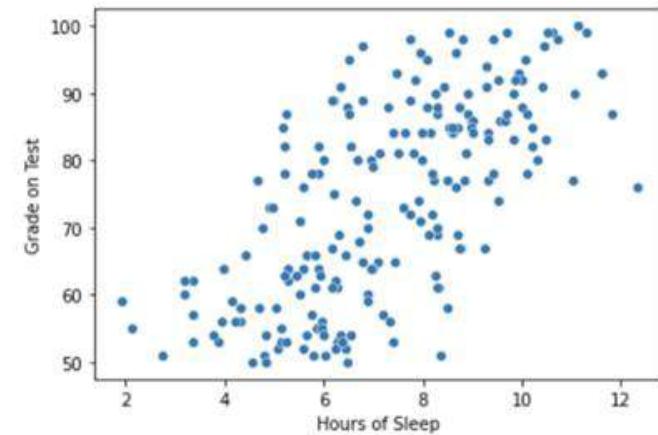
- Get a better sense of the data by viewing it from multiple angles
- Discover patterns, relationships, and insights from the data

From data...

	Hours of Sleep	Hours Studied	Grade on Test
0	10.602667	4.586892	99
1	8.172997	4.405120	72
2	6.430132	-0.519630	52
3	7.963793	5.004348	80
4	8.279421	2.984489	87

EDA

... to insights



Students with
more sleep got
higher grades!



COMMON EDA TECHNIQUES

These are some **common EDA techniques** used by data scientists:

Exploring data

*Viewing & summarizing
data from multiple angles*

Examples:

- Filtering
- Sorting
- Grouping

Visualizing data

*Using charts to identify
trends & patterns*

Examples:

- Histograms
- Scatterplots
- Pair plots



There is **no particular order** that these tasks need to be completed in;
you are free to mix and match them depending on your data set



FILTERING

You can filter a DataFrame by **passing a logical test** into the loc[] accessor

- Apply multiple filters by using the “&” and “|” operators (AND/OR)

tea

	type	name	temp
0	green	sencha	180
1	black	chai	210
2	black	ceylon	200
3	herbal	mint	212
4	herbal	ginger	212
5	herbal	chamomile	200
6	herbal	tumeric	175
7	oolong	green oolong	190



tea.loc[tea.type == 'herbal']

	type	name	temp
3	herbal	mint	212
4	herbal	ginger	212
5	herbal	chamomile	200
6	herbal	tumeric	175

This returns rows where
“type” is equal to “herbal”

```
mask = (tea.type == 'herbal') | (tea.temp >= 200)  
tea.loc[mask]
```

	type	name	temp
3	herbal	mint	212
4	herbal	ginger	212
5	herbal	chamomile	200
6	herbal	tumeric	175

This returns rows where
“type” is equal to “herbal”
OR “temp” is greater than
or equal to 200

← Use a **Boolean mask** to
apply multiple filters with
complex logic



SORTING

You can sort a DataFrame by using the `.sort_values()` method

- This sorts in ascending order by default

tea



`tea.sort_values('name')`



`tea.sort_values('name', ascending=False)`

	type	name	temp
0	green	sencha	180
1	black	chai	210
2	black	ceylon	200
3	herbal	mint	212
4	herbal	ginger	212
5	herbal	chamomile	200
6	herbal	tumeric	175
7	oolong	green oolong	190

	type	name	temp
2	black	ceylon	200
1	black	chai	210
5	herbal	chamomile	200
4	herbal	ginger	212
7	oolong	green oolong	190
3	herbal	mint	212
0	green	sencha	180
6	herbal	tumeric	175

	type	name	temp
6	herbal	tumeric	175
0	green	sencha	180
3	herbal	mint	212
7	oolong	green oolong	190
4	herbal	ginger	212
5	herbal	chamomile	200
1	black	chai	210
2	black	ceylon	200



GROUPING

The “**split-apply-combine**” approach is used to group data in a DataFrame and apply calculations to each group

EXAMPLE

Finding the average temperature for each type of tea

	type	name	temp
0	green	sencha	180
1	black	chai	210
2	black	ceylon	200
3	herbal	mint	212
4	herbal	ginger	212
5	herbal	chamomile	200
6	herbal	tumeric	175
7	oolong	green oolong	190

Split the data by “type”

0	green	sencha	180	180
1	black	chai	210	210
2	black	ceylon	200	200
3	herbal	mint	212	212
4	herbal	ginger	212	212

2	black	ceylon	200	205
3	herbal	mint	212	212
4	herbal	ginger	212	212
5	herbal	chamomile	200	200
6	herbal	tumeric	175	175

7	oolong	green oolong	190	190
---	--------	--------------	-----	-----

Apply a calculation (average) on the “temp” in each group

	type	temp
0	black	205.00
1	green	180.00
2	herbal	199.75
3	oolong	190.00

Combine the “type” and average “temp” for each group into a final table



GROUPING

You can group a DataFrame by using the **.groupby()** method

```
df.groupby(col) [col].aggregation()
```

The DataFrame
to group

The column(s) to group by
(unique values determine
the **rows** in the output)

The column(s) to apply
the calculation to
(these become the new
columns in the output)

The calculation(s)
to apply for each group
(these become the new
values in the output)

Examples:

- `mean()`
- `sum()`
- `min()`
- `max()`
- `count()`
- `nunique()`



GROUPING

You can group a DataFrame by using the `.groupby()` method

```
tea
```

	type	name	temp
0	green	sencha	180
1	black	chai	210
2	black	ceylon	200
3	herbal	mint	212
4	herbal	ginger	212
5	herbal	chamomile	200
6	herbal	tumeric	175
7	oolong	green oolong	190



```
tea.groupby('type')['temp'].mean()
```

```
type  
black      205.00  
green     180.00  
herbal    199.75  
oolong    190.00  
Name: temp, dtype: float64
```

This returns the average
"temp" by "type"

```
tea.groupby('type')['temp'].mean().reset_index()
```

```
type   temp  
0   black  205.00  
1   green  180.00  
2   herbal 199.75  
3   oolong 190.00
```

Use `reset_index()` to
return a DataFrame





MULTIPLE AGGREGATIONS

Chain the `.agg()` method to `.groupby()` to apply multiple aggregations to each group

```
tea.groupby('type')['temp'].agg(['min', 'max', 'count', 'nunique']).reset_index()
```

	type	min	max	count	nunique
0	black	200	210	2	2
1	green	180	180	1	1
2	herbal	175	212	4	3
3	oolong	190	190	1	1

This returns the minimum & maximum temperatures, the number of teas, and the unique temperatures for each tea type

You can also write the code this way!

```
(tea.groupby('type')['temp']
    .agg(['min', 'max', 'count', 'nunique'])
    .reset_index())
```



PRO TIP: When chaining multiple methods together, wrap the code in parenthesis so you can place each method on a separate line
(this makes reading the code easier!)



PRO TIP: HEAD & TAIL

In addition to aggregating grouped data, you can also use the `.head()` and `.tail()` methods to return the first or last “n” records for each group

sales			
	Name	Date	Sales
0	Anna	3/7/23	9
1	Anna	3/10/23	28
2	Anna	3/13/23	14
3	Anna	3/16/23	8
4	Anna	3/19/23	27
5	Bob	4/1/23	33
6	Bob	4/2/23	9
7	Bob	4/3/23	20
8	Bob	4/4/23	31

Return the first row of each group

sales.groupby('Name').head(1)			
	Name	Date	Sales
0	Anna	3/7/23	9
5	Bob	4/1/23	33

Return the last 3 rows of each group

sales.groupby('Name').tail(3)			
	Name	Date	Sales
2	Anna	3/13/23	14
3	Anna	3/16/23	8
4	Anna	3/19/23	27
6	Bob	4/2/23	9
7	Bob	4/3/23	20
8	Bob	4/4/23	31



VISUALIZING DATA

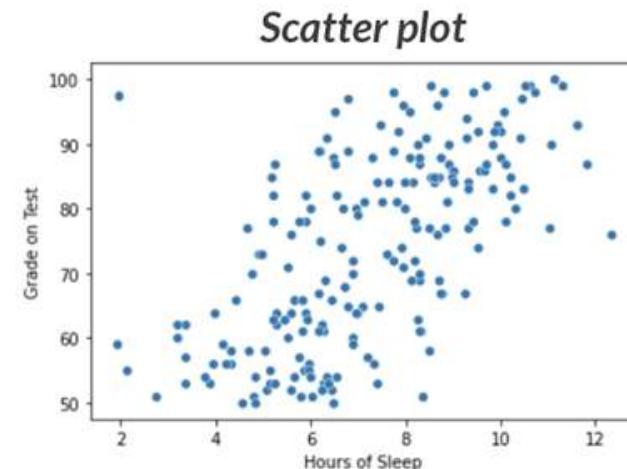
Visualizing data as part of the exploratory data analysis process lets you:

- More easily identify patterns and trends in the data
- Quickly spot anomalies to further investigate

```
student_data[['Hours of Sleep', 'Grade on Test']]
```

	Hours of Sleep	Grade on Test
0	10.602667	99
1	8.172997	72
2	6.430132	52
3	7.963793	80
4	8.279421	87
...
195	4.155300	59
196	4.306093	58
197	4.686733	58
198	9.407950	98
199	5.989777	54

200 rows × 2 columns



Students with more sleep got higher grades



One student barely slept but tested very well



Data visualization is also used later in the data science process to communicate insights to stakeholders



DATA VISUALIZATION IN PANDAS

You can use the `.plot()` method to create quick and simple visualizations directly from a Pandas DataFrame

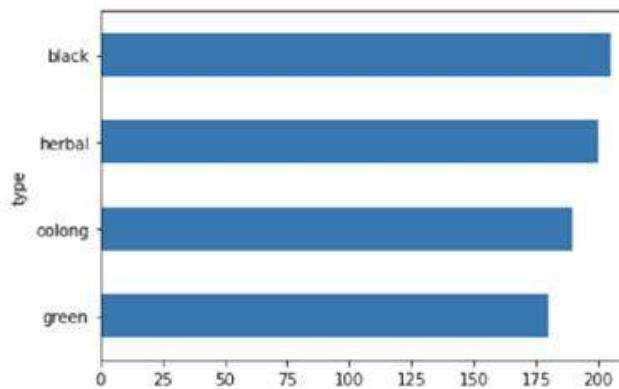
```
# tea temperatures by tea type  
tea_temps
```

```
type  
green    180.00  
oolong   190.00  
herbal   199.75  
black    205.00  
Name: temp, dtype: float64
```

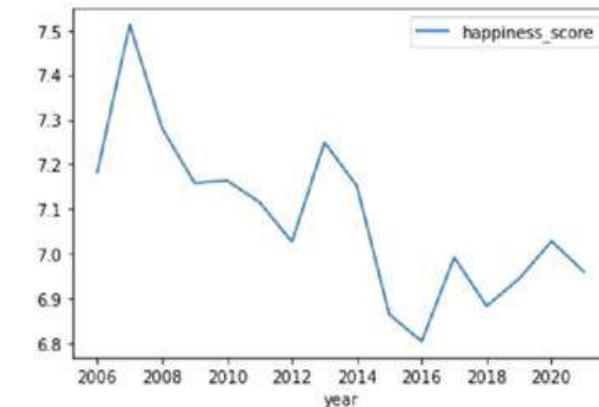
```
# happiness scores for the US  
us_happiness.head()
```

	country_name	year	happiness_score
1967	United States	2006	7.181794
1968	United States	2007	7.512688
1969	United States	2008	7.280386
1970	United States	2009	7.158032
1971	United States	2010	7.163616

```
# create a bar chart from a dataframe  
tea_temps.plot.barh();
```



```
# create a line chart from a dataframe  
us_happiness.plot.line(x='year', y='happiness_score');
```





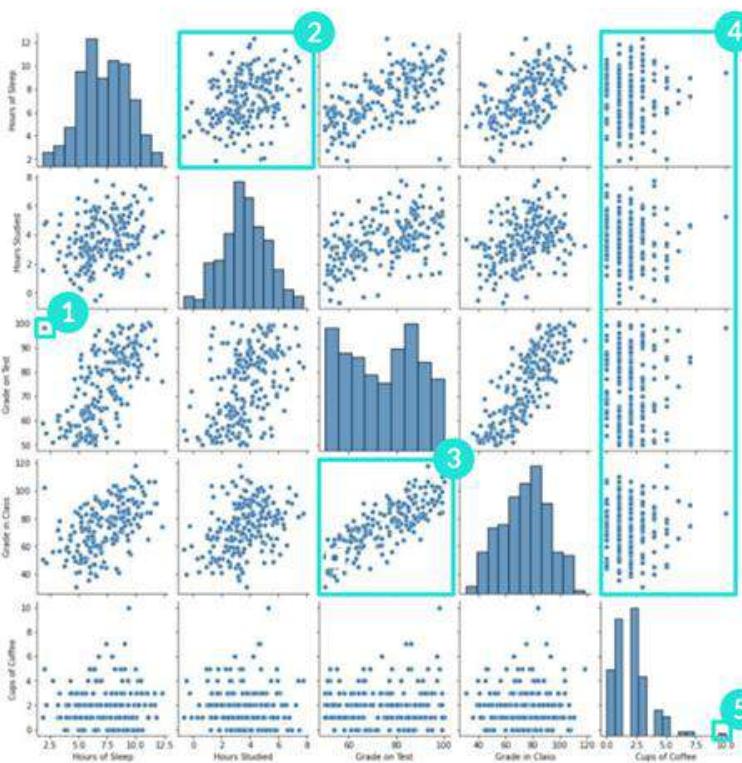
PRO TIP: PAIR PLOTS

Use `sns.pairplot()` to create a pair plot that shows all the scatterplots and histograms that can be made using the numeric variables in a DataFrame

```
import seaborn as sns  
sns.pairplot(student_data);
```



PRO TIP: Create a pair plot as your first visual to identify general patterns that you can dig into later individually



- 1 **Outlier** – This student barely studied and still aced the test (look into them)
- 2 **Relationship** – The hours spent studying and sleeping don't seem related at all (this is a surprising insight)
- 3 **Relationship** – The test grade is highly correlated with the class grade (can ignore one of the two fields for the analysis)
- 4 **Data Type** – The cups of coffee field only has integers (keep this in mind)
- 5 **Outlier** – This student drinks a lot of coffee (check on them)



DISTRIBUTIONS

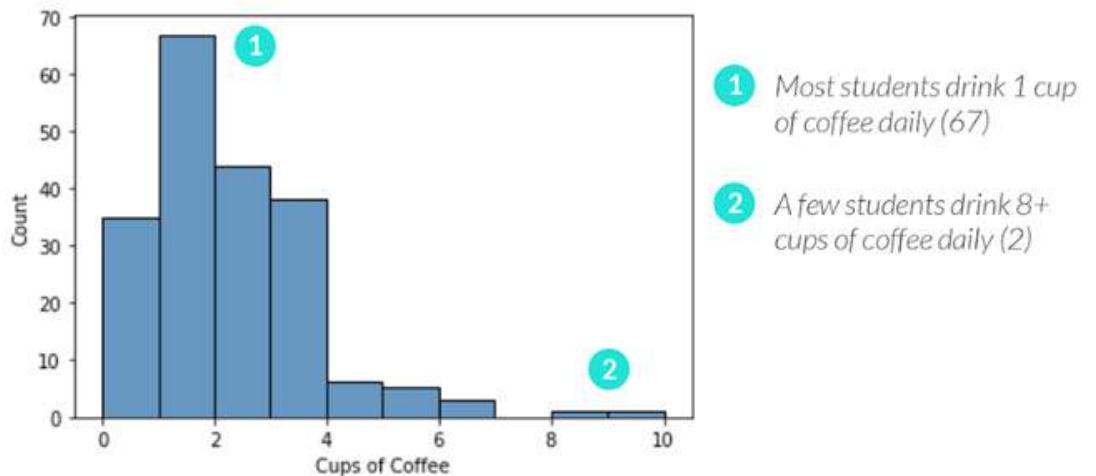
A **distribution** shows all the possible values in a column and how often each occurs. It can be shown in two ways:

Frequency table

0	35
1	67
2	44
3	38
4	6
5	5
6	3
8	1
10	1

Name: Cups of Coffee

Histogram



PRO TIP: Distributions can be used to find inconsistencies and outliers



FREQUENCY TABLES

You can create a frequency table with the `.value_counts()` method

```
student_data['Cups of Coffee']
```

```
0      5  
1      3  
2      4  
3      1  
4      0  
..  
195     2  
196     1  
197     1  
198    10  
199     2  
  
Name: Cups of Coffee, Length: 200, dtype: int64
```



```
(student_data['Cups of Coffee'].value_counts()  
 .sort_index())
```

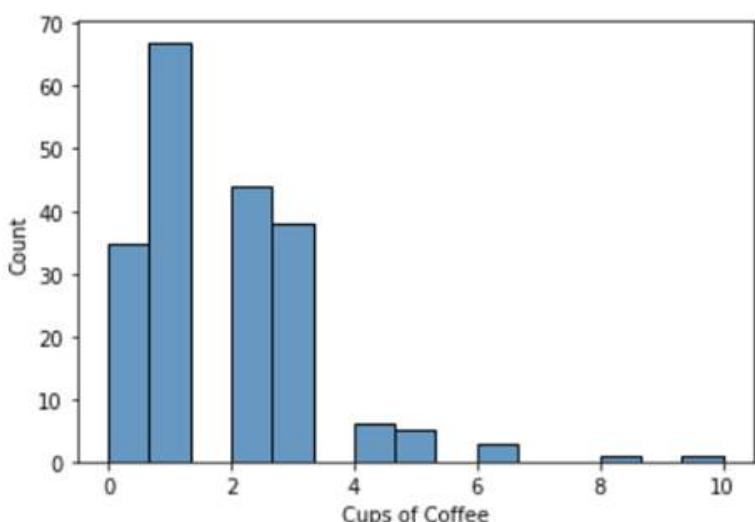
```
0      35  
1      67  
2      44  
3      38  
4       6  
5       5  
6       3  
8       1  
10      1  
  
Name: Cups of Coffee, dtype: int64
```



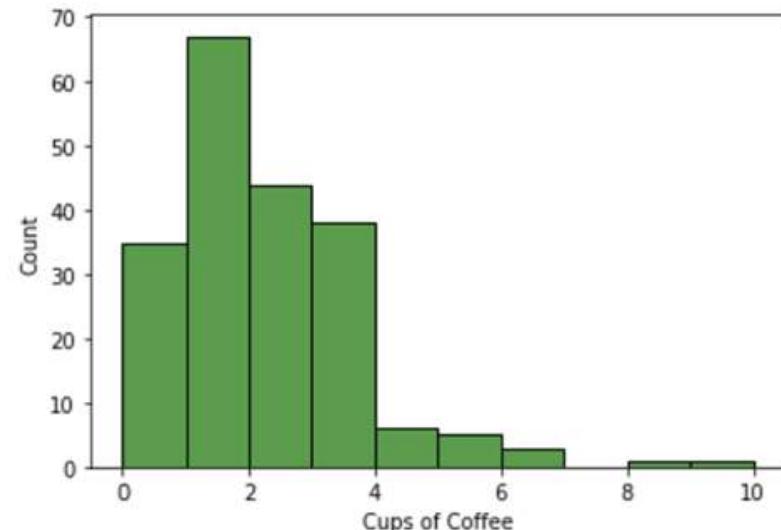
HISTOGRAMS

You can create a histogram by passing a DataFrame column to **sns.histplot()**

```
sns.histplot(student_data['Cups of Coffee'])  
<AxesSubplot:xlabel='Cups of Coffee', ylabel='Count'>
```



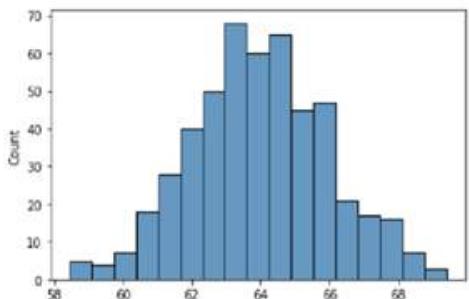
```
sns.histplot(student_data['Cups of Coffee'],  
            bins=10, color='green');
```





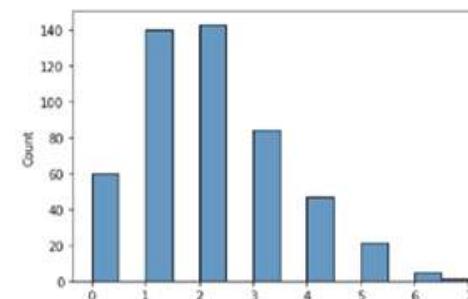
COMMON DISTRIBUTIONS

These are some **common distributions** you'll encounter as a data scientist



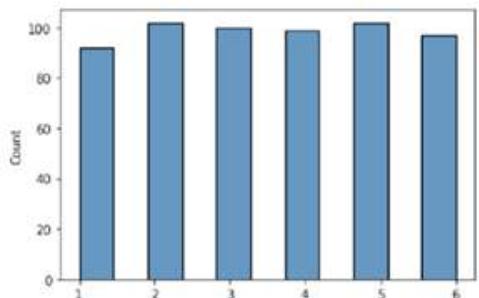
Normal distribution

If you collect the height of 500 women, most will be ~64" with fewer being much shorter or taller than that



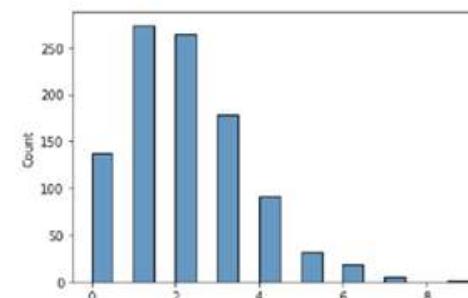
Binomial distribution

Knowing that 10% of all ad views result in a click, these are the clicks you'll get if you show an ad to 20 customers



Uniform distribution

If you roll a die 500 times, the chances of getting any of the 6 numbers is the same



Poisson distribution

Knowing that you typically get 2 cancellations each day, these are the number of cancellations you will see on any given day



While it's not necessary to memorize the formulas for each distribution, being able to **recognize the shapes and data types** for these distributions will be helpful for future data science modeling and analysis



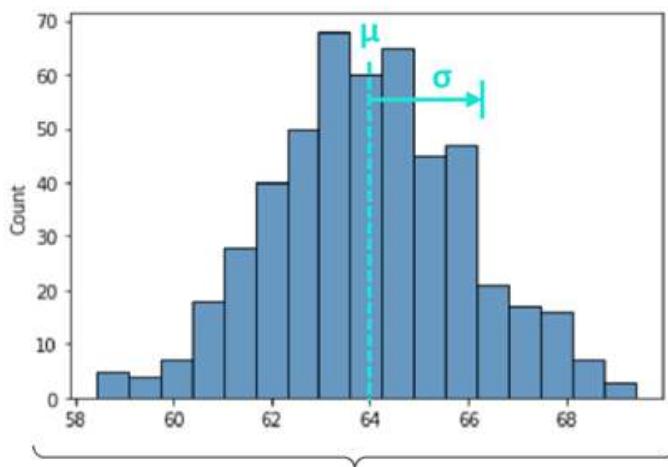
NORMAL DISTRIBUTION

Many numeric variables naturally follow a **normal distribution**, or “bell curve”

- Normal distributions are described by two values: the **mean (μ) & standard deviation (σ)**
- The standard deviation measures, on average, how far each value lies from the mean

EXAMPLE

Women's Heights (in)



A sample of 500 women shows a mean height of 5'4" (64 inches) and a standard deviation of 2.2 inches



The **empirical rule** outlines where most values fall in a normal distribution:

- 68% fall within **1σ** from the mean
- 95% fall within **2σ** from the mean
- 99.7% fall within **3σ** from the mean

(this is why data points over 3σ away from the mean are considered outliers)

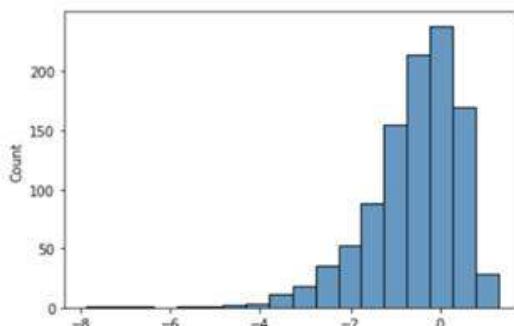


SKEW

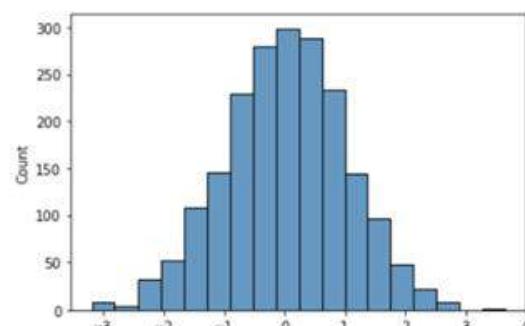
The **skew** represents the asymmetry of a normal distribution around its mean

- For example, data on household income is typically right skewed

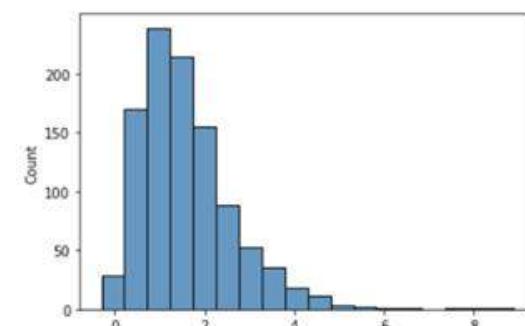
Left skew



Normal Distribution



Right skew



There are **techniques that can deal with skewed data**, such as taking the log of the data set, to turn it into normally distributed data; more on this will be discussed in the prep for modeling section



SCATTERPLOTS

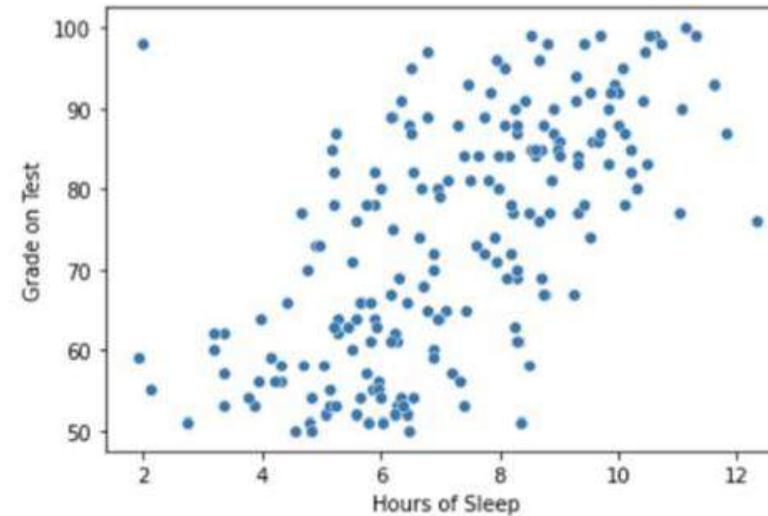
Scatterplots are used to visualize the relationship between numerical variables

- `sns.scatterplot(data=df, x="x axis column", y="y axis column")`

```
student_data.head(3)
```

	Hours of Sleep	Hours Studied	Grade on Test
0	10.602667	4.586892	99
1	8.172997	4.405120	72
2	6.430132	-0.519630	52

```
sns.scatterplot(data=student_data,  
                 x='Hours of Sleep',  
                 y='Grade on Test');
```

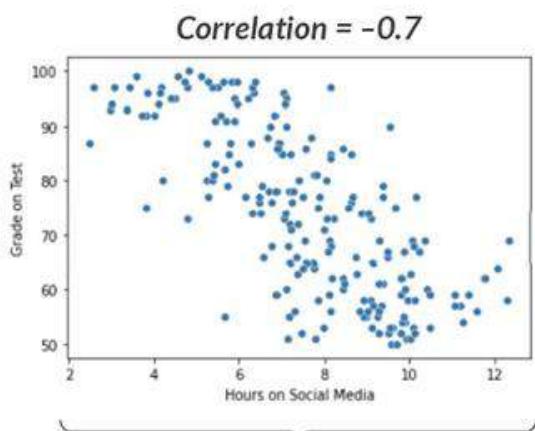




CORRELATION

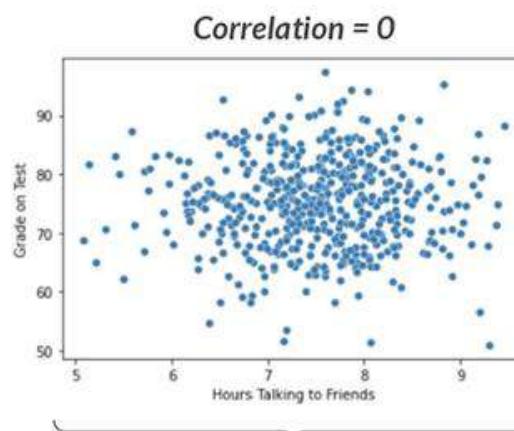
A **correlation** describes the relationship between two numerical columns (-1 to 1)

- **-1** is a perfect negative correlation, **0** is no correlation, and **1** is a perfect positive correlation



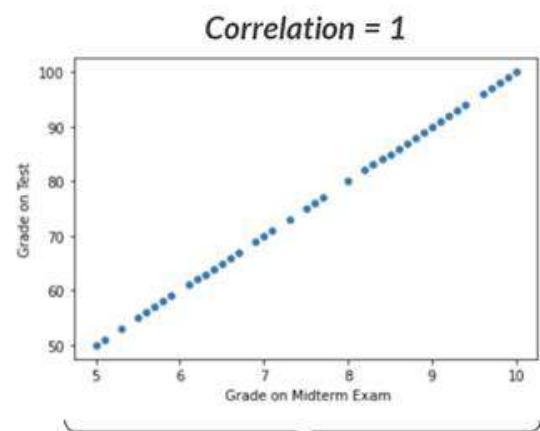
This is a **negative correlation**

(Students who spent more time on social media got worse grades)



This has **no correlation**

(No relationship exists between hours talking with friends and test grades)



This is a **perfect positive correlation**

(This is likely an error and the grades are exact copies of each other)



Correlation does not imply causation! Just because two variables are related does not necessarily mean that changes to one cause changes to the other



CORRELATION

Use the `.corr()` method to calculate the correlation coefficient between each pair of numerical variables in a DataFrame

- A correlation under 0.5 is weak, between 0.5 and 0.8 is moderate, and over 0.8 is strong

```
student_data.corr()
```

	Hours of Sleep	Hours Studied	Grade on Test	Grade in Class	Cups of Coffee
Hours of Sleep	1.000000	0.265404	0.637864	0.502397	-0.079643
Hours Studied	0.265404	1.000000	0.479153	0.374456	-0.020209
Grade on Test	0.637864	0.479153	1.000000	0.815721	-0.038804
Grade in Class	0.502397	0.374456	0.815721	1.000000	-0.035288
Cups of Coffee	-0.079643	-0.020209	-0.038804	-0.035288	1.000000

1 The correlation between a variable with itself will always be 1 – you can ignore these values across the diagonal

2 While this looks like a negative correlation, this value is very close to zero, meaning that hours studied and cups of coffee are uncorrelated

3 The only strong correlation in this table is that the grade on the test is **positively correlated** with the grade in the class (keep this in mind for future modeling or insights)



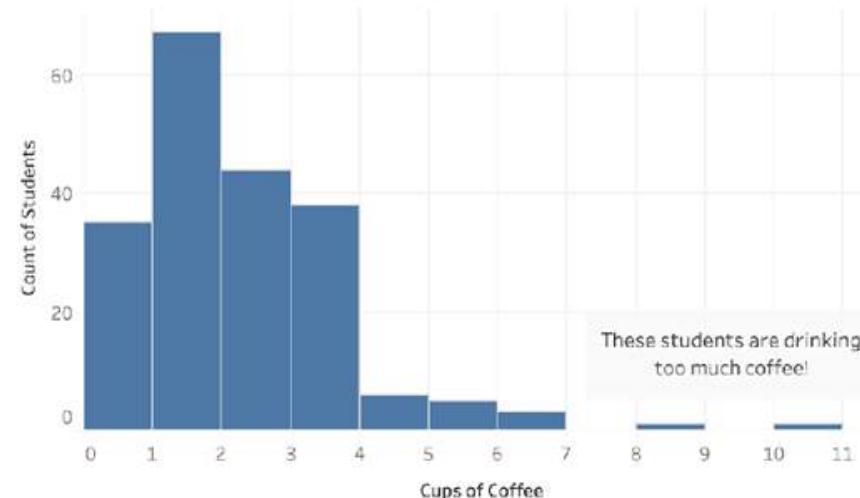
DATA VISUALIZATION IN PRACTICE

Data visualizations can be **difficult to tweak and polish in Python**, so feel free to export the data as a CSV file and import it into another tool (*Excel, Tableau, etc.*)

```
student_data.to_csv('student_data.csv')
```



Cups of Coffee Consumed Daily



PRO TIP: Before sharing a visual, take a moment to think about what you want your audience to take away from it
If possible, modify, remove or highlight specific parts emphasize your points



EDA TIPS

Before diving into EDA:

- Remind yourself of the original question(s) that you're trying to answer

As you go through EDA:

- Keep a running list of observations and questions for both yourself and your client
- Apply the techniques in any order that makes the most sense for your data
- You may have to go back to earlier steps and gather more data or further clean your data as you discover more about your data during EDA

You know you've completed your initial EDA once you've:

- Investigated any initial idea or question that comes to mind about the data and gathered some meaningful insights (*you can always come back to EDA after modeling!*)



Working with a large data set?

Look at a subset, apply EDA techniques, then extrapolate to the whole data set



Already answered your question?

Sometimes EDA is all you need, and you may not need to apply any algorithms

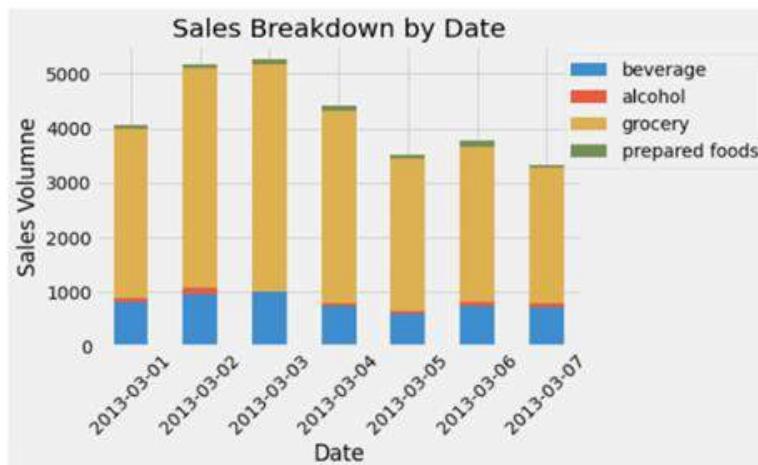
DATA VISUALIZATION



THE MATPLOTLIB API

Pandas uses the **Matplotlib API** to create charts and visualize data

- This is an integration with the main Matplotlib library



This course only covers Pandas' limited data visualization capabilities, but other full Python libraries like **Matplotlib**, **Seaborn**, and **Plotly** offer more chart types, options and capabilities



THE PLOT METHOD

You can visualize a DataFrame by using the `.plot()` method

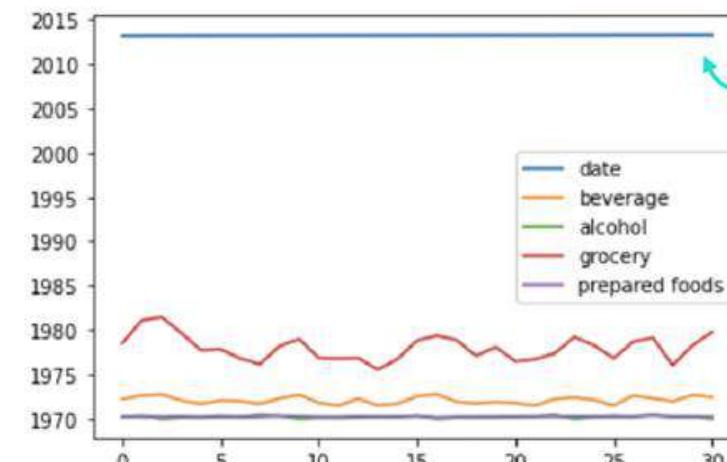
- This creates a **line chart** by default, using the row index as the x-axis and plotting each numerical column as a separate series on the y-axis

```
sales_df.head()
```

	date	beverage	alcohol	grocery	prepared foods
0	2013-03-01	809.0	75.0	3105.0	81.0
1	2013-03-02	950.0	117.0	4044.0	75.0
2	2013-03-03	1000.0	0.0	4186.0	80.0
3	2013-03-04	750.0	50.0	3525.0	90.0
4	2013-03-05	595.0	41.0	2815.0	60.0

```
sales_df.plot()
```

<AxesSubplot:>



Note that 'date' was plotted as a numeric series

The index is used as
the x-axis by default



THE PLOT METHOD

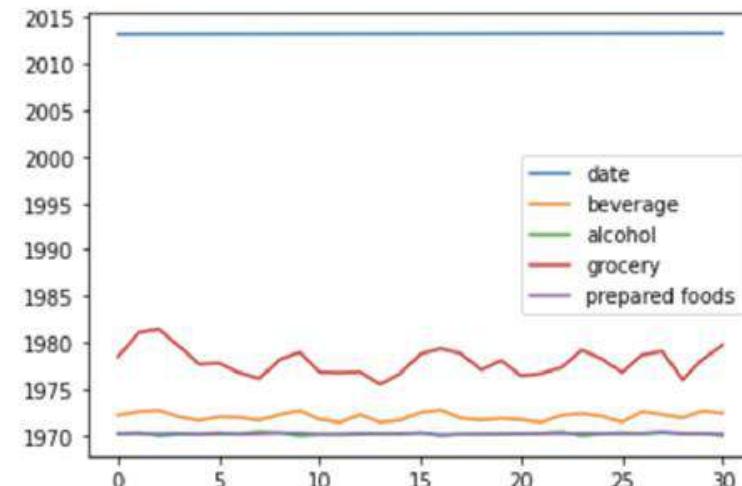
You can visualize a DataFrame by using the `.plot()` method

- This creates a **line chart** by default, using the row index as the x-axis and plotting each numerical column as a separate series on the y-axis

```
sales_df.head()
```

	date	beverage	alcohol	grocery	prepared foods
0	2013-03-01	809.0	75.0	3105.0	81.0
1	2013-03-02	950.0	117.0	4044.0	75.0
2	2013-03-03	1000.0	0.0	4186.0	80.0
3	2013-03-04	750.0	50.0	3525.0	90.0
4	2013-03-05	595.0	41.0	2815.0	60.0

```
sales_df.plot();
```



PRO TIP: Add ";" at the end of the code to remove "<AxesSubplot:>" when using Jupyter Notebooks

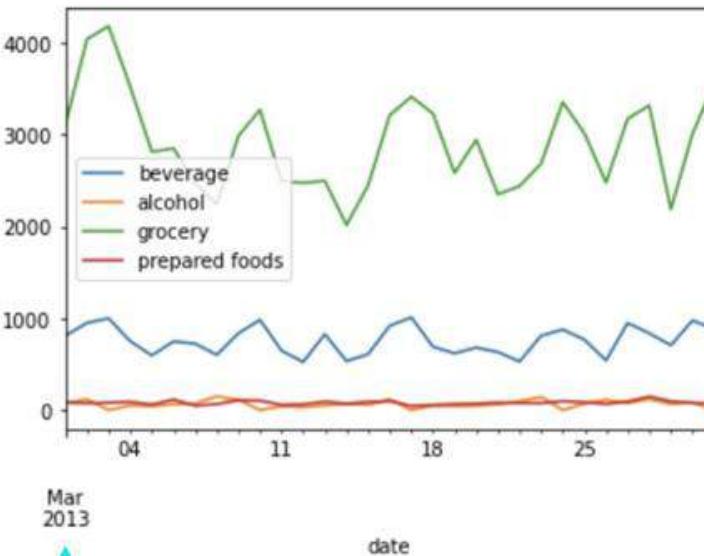


THE PLOT METHOD

You can **change the x-axis** by setting a different index or using the "x" argument

This sets 'date' as the index before plotting the chart

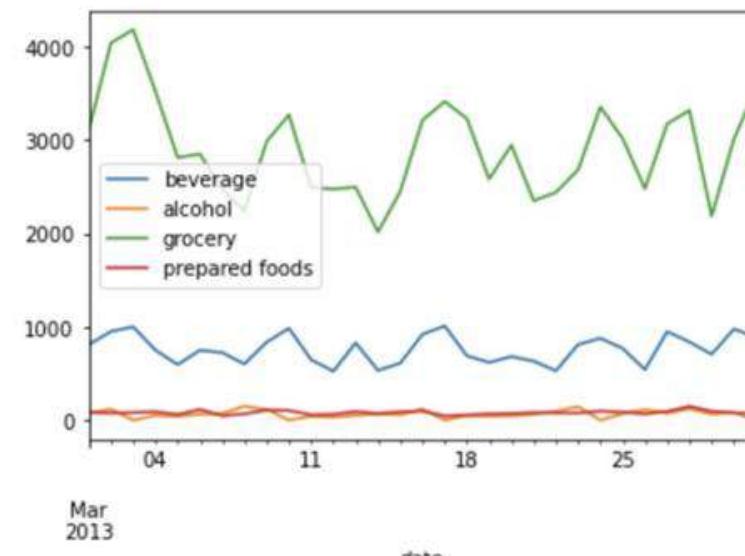
```
sales_df.set_index("date").plot()  
<AxesSubplot:xlabel='date'>
```



It automatically splits
the date components!

This sets 'date' as the x-axis in the .plot() parameters

```
sales_df.plot(x="date")  
<AxesSubplot:xlabel='date'>
```



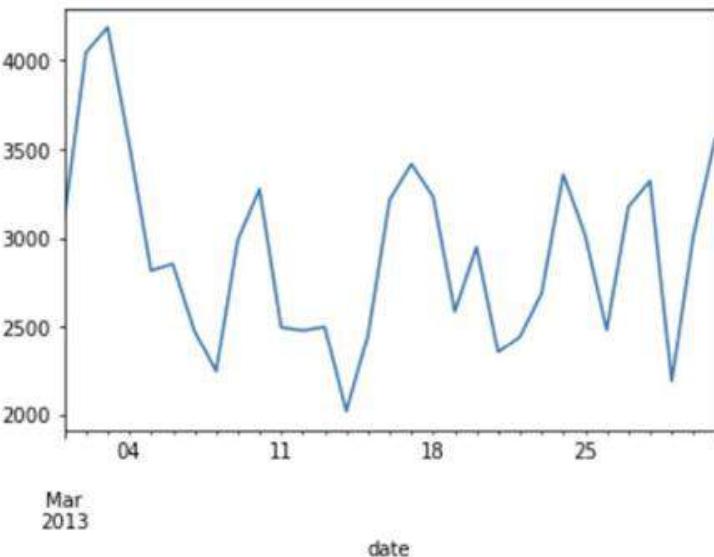


THE PLOT METHOD

You can **select series** to plot with the `.loc[]` accessor or using the "y" argument

This filters the 'grocery' column before plotting the chart

```
sales_df.set_index("date").loc[:, "grocery"].plot()  
<AxesSubplot:xlabel='date'>
```



This plots 'grocery' values along the y-axis

```
sales_df.plot(x="date", y="grocery")  
<AxesSubplot:xlabel='date'>
```

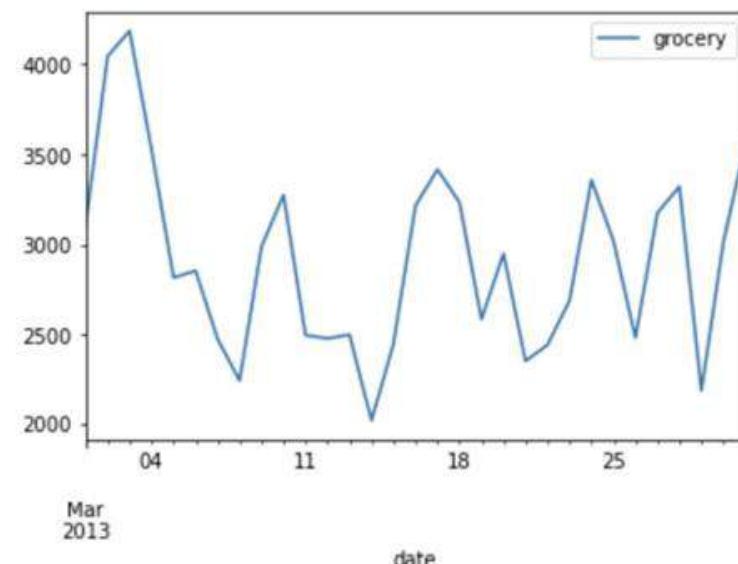




CHART FORMATTING

You can modify the **chart formatting** by using `.plot()` method arguments

- **title = "title"** – title to use for the chart
- **xlabel = "title"** – name to use for the x-axis
- **ylabel = "title"** – name to use for the y-axis
- **color = "color" or "#hexacode"** – color(s) to use for the data series
- **cmap = "color palette"** – preset color palette to apply to the chart
- **style = "symbol"** – style for the line (*dashed, dotted, etc.*)
- **legend = True/False** – adds or removes the legend from the chart
- **rot = degrees** – degrees to rotate the x-axis labels for readability
- **figsize = (width, height)** – size for the chart in inches
- **grid = True/False** – adds gridlines to the chart when True
- **subplots = True/False** – creates a separate chart for each series when True
 - **sharex = True/False** – each series in the subplot shares the x-axis when True
 - **sharey = True/False** – each series in the subplot shares the y-axis when True
 - **layout = (rows, columns)** – the rows and columns to break the subplots into

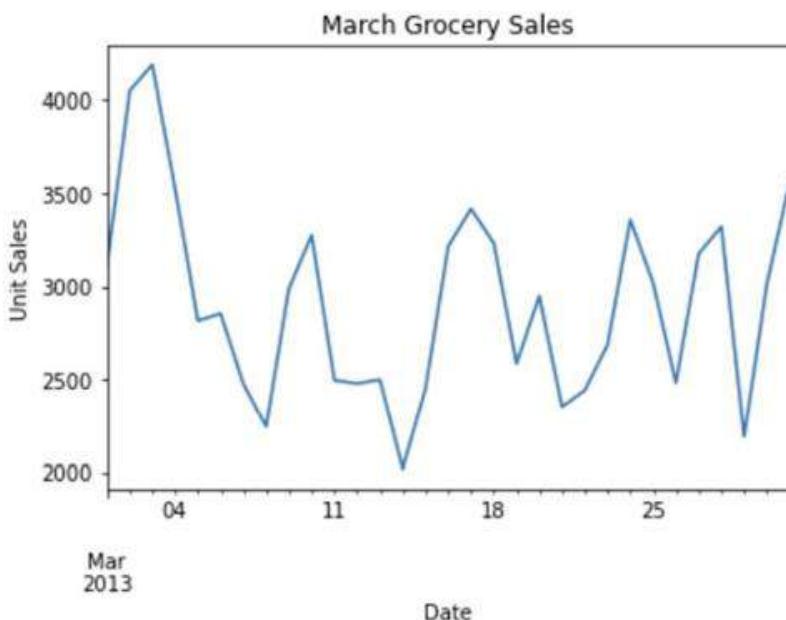


CHART TITLES

You can add a custom **chart title** and **axis labels** to increase understanding

```
sales_df.set_index("date").loc[:, "grocery"].plot(  
    title="March Grocery Sales",  
    xlabel="Date",  
    ylabel="Unit Sales"  
);
```

Break each .plot() argument into separate rows to make the code easier to read



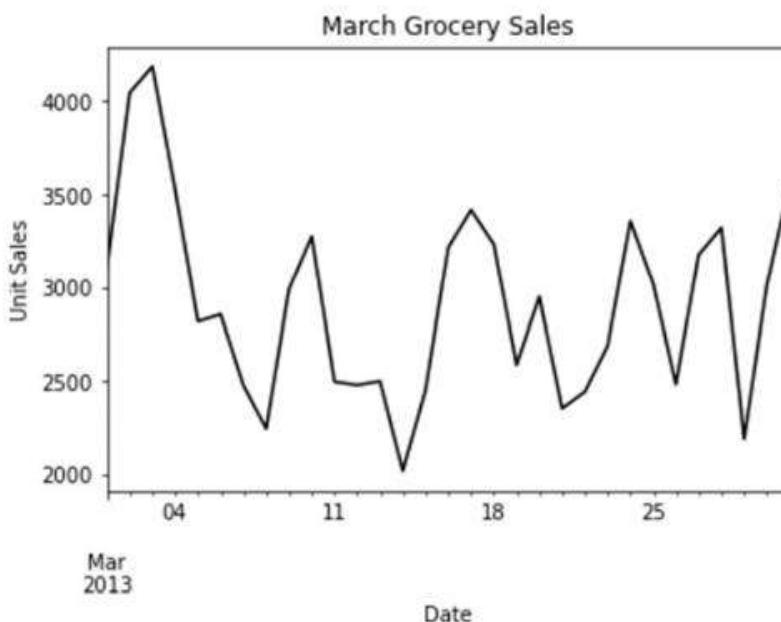


SERIES COLORS

You can modify the **series colors** by using common color names or hex codes

```
sales_df.set_index("date")["grocery"].plot(  
    title="March Grocery Sales",  
    xlabel="Date",  
    ylabel="Unit Sales",  
    color="Black"  
)
```

Note that Python understands basic colors like "Black", "Red", "Green", etc.

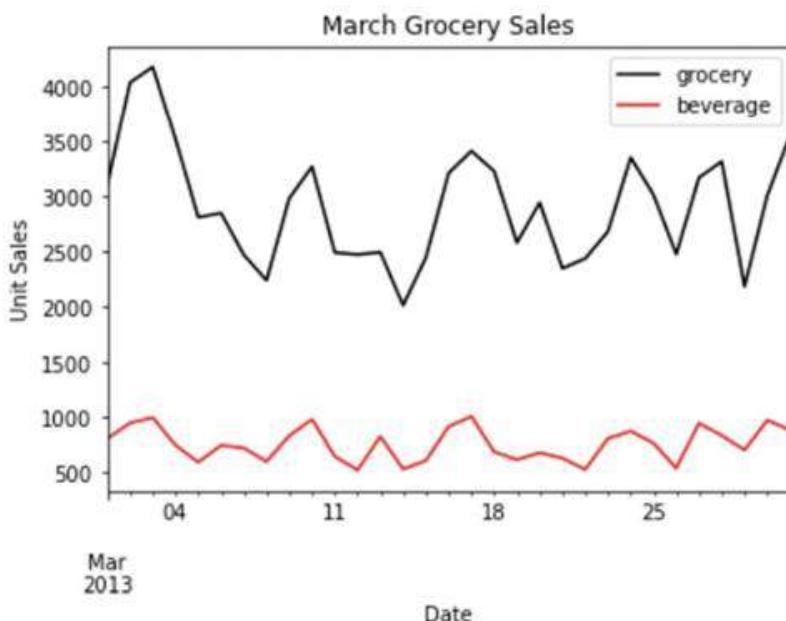




SERIES COLORS

You can modify the **series colors** by using common color names or hex codes

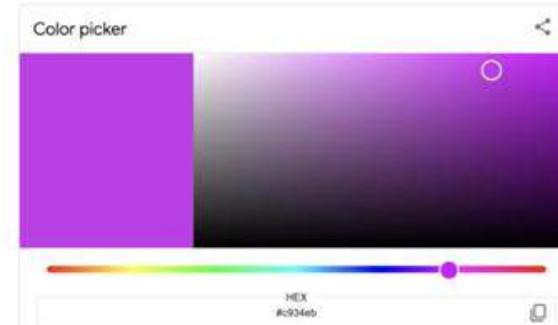
```
sales_df.set_index("date").loc[:, ["grocery", "beverage"]].plot(  
    title="March Grocery Sales",  
    xlabel="Date",  
    ylabel="Unit Sales",  
    color=["#000000", "#FF0000"], ← Hexadecimal codes work too!  
);
```



If you have more than one series,
you can pass the colors in a list



PRO TIP: Sites like Google have
helpful hexadecimal color pickers





COLOR PALETTES

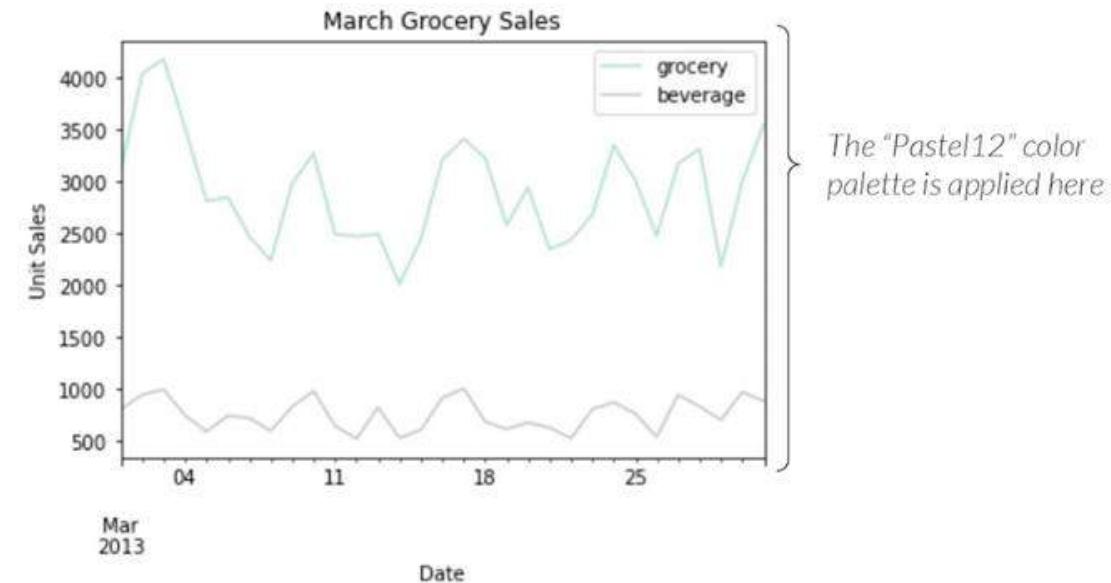
You can also modify the entire **color palette** for the series in the chart

Default Color Palette:



Series colors are applied in this sequential order

```
sales_df.set_index("date").loc[:, ["grocery", "beverage"]].plot(  
    title="March Grocery Sales",  
    xlabel="Date",  
    ylabel="Unit Sales",  
    cmap="Pastel2",  
)
```





LINE STYLE

Line charts are solid by default, but you can change the **line style** if needed

```
sales_df.set_index("date").loc[:, ["grocery", "beverage"]].plot(  
    title="March Grocery Sales",  
    xlabel="Date",  
    ylabel="Unit Sales",  
    color=["Black", "Red"],  
    style=[ "--", "-."],  
);
```



This sets the first series as a dashed line and the second as dot-dashed

Symbol	Line Style
-	Solid (default)
--	Dashed
-.	Dash-dot
..	Dotted



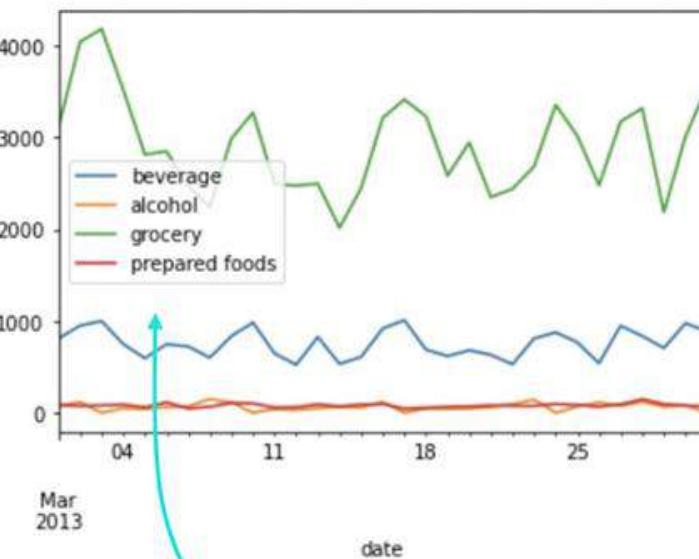
CHART LEGEND

The “legend” .plot() argument lets you **add or remove the legend**

- This can be useful in some scenarios, but in others you’ll want to *reposition* the legend

```
sales_df.set_index("date").plot()
```

```
<AxesSubplot:xlabel='date'>
```



Pandas tries to place
it in the “best” spot

```
sales_df.set_index("date").plot(legend=False)
```

```
<AxesSubplot:xlabel='date'>
```

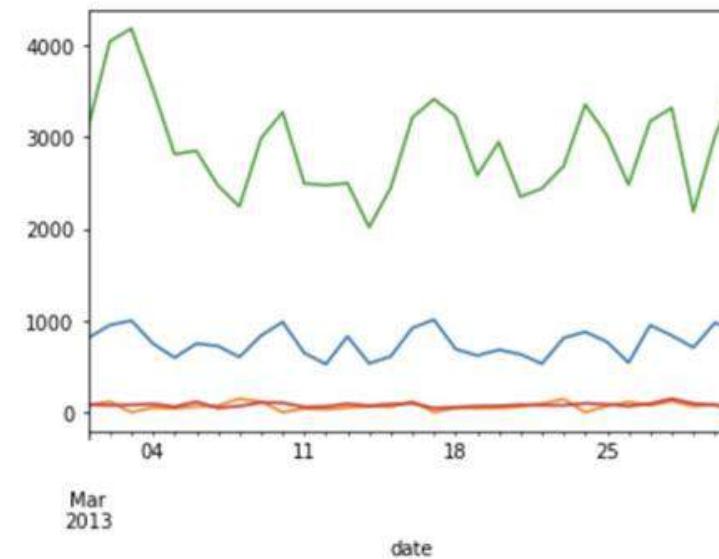




CHART LEGEND

You can **reposition the legend** by chaining the `.legend()` method to `.plot()` and specifying its location with the “`loc`” or “`bbox_to_anchor`” arguments

Location Options

best (default)

upper right

upper left

upper center

lower right

lower left

lower center

center right

center left

center

```
sales_df.set_index("date").loc[:, ["grocery", "beverage"]].plot(  
    title="March Grocery Sales",  
    xlabel="Date",  
    ylabel="Unit Sales",  
    color=["Black", "Red"],  
    style=[ "--", "-."],  
    ).legend(loc="center left");
```

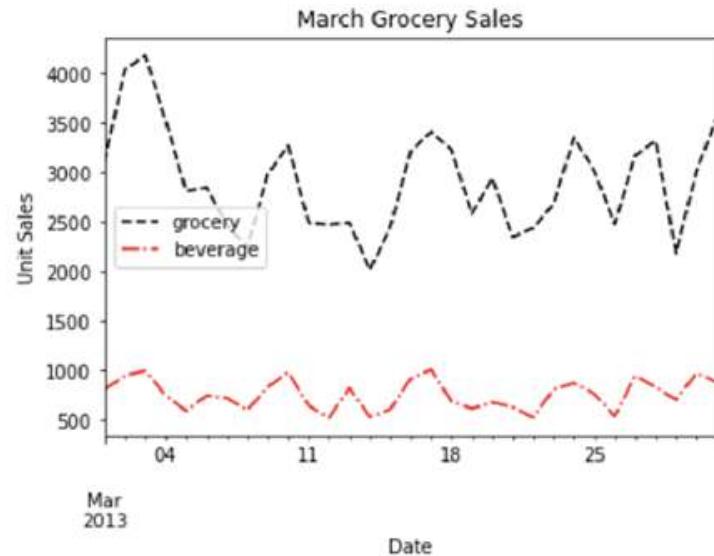




CHART LEGEND

You can **reposition the legend** by chaining the `.legend()` method to `.plot()` and specifying its location with the “loc” or “bbox_to_anchor” arguments

Location Options

best (default)

upper right

upper left

upper center

lower right

lower left

lower center

center right

center left

center

```
sales_df.set_index("date").loc[:, ["grocery", "beverage"]].plot(  
    title="March Grocery Sales",  
    xlabel="Date",  
    ylabel="Unit Sales",  
    color=["Black", "Red"],  
    style=[ "--", "-."],  
    ).legend(bbox_to_anchor=(1.3, 1));
```



“bbox_to_anchor” lets you specify
the coordinates for the legend
(it takes some trial and error!)



GRIDLINES

You can add **gridlines** to charts for visual reference

```
sales_df.set_index("date").loc[:, ["grocery", "beverage"]].plot(  
    title="March Grocery Sales",  
    xlabel="Date",  
    ylabel="Unit Sales",  
    color=["Black", "Red"],  
    style=[ "--", "-."],  
    grid=True,  
).legend(bbox_to_anchor=(1.3, 1))
```



This adds gridlines on the y-axis for line charts, but other chart types get x and y gridlines



PRO TIP: CHART STYLES

Matplotlib & Seaborn have premade **style templates** that can be applied to charts

- Once a style is set, it will automatically be applied to all charts

```
import seaborn as sns
sns.set_style("darkgrid") ← The style is set in advance

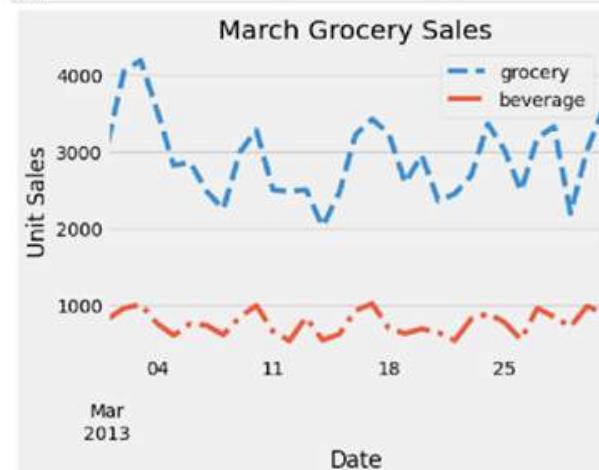
sales_df.set_index("date").loc[:, ["grocery", "beverage"]].plot(
    title="March Grocery Sales",
    xlabel="Date",
    ylabel="Unit Sales",
);
```



This is Seaborn's **darkgrid** style, which has better font sizing, and a background color

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

sales_df.set_index("date").loc[:, ["grocery", "beverage"]].plot(
    title="March Grocery Sales",
    xlabel="Date",
    ylabel="Unit Sales",
    # color=["Black", "Red"],
    style=[ "--", "-."], ← You can still customize each chart
);
```



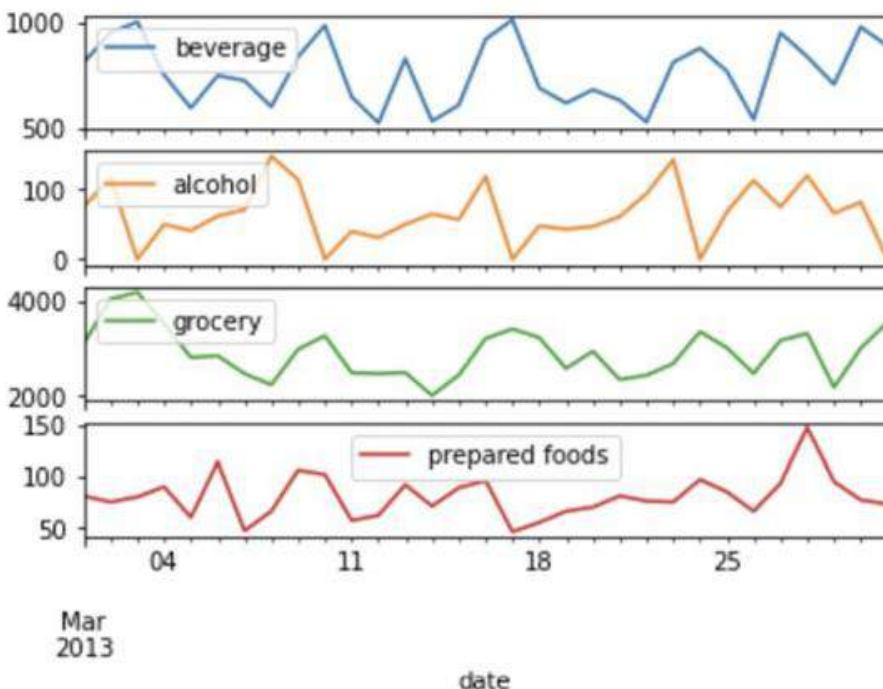
This is Matplotlib's **fivethirtyeight** style, but with dashed lines instead of solid



SUBPLOTS

You can leverage **subplots** to create a separate chart for each series

```
sales_df.set_index("date").plot(subplots=True);
```



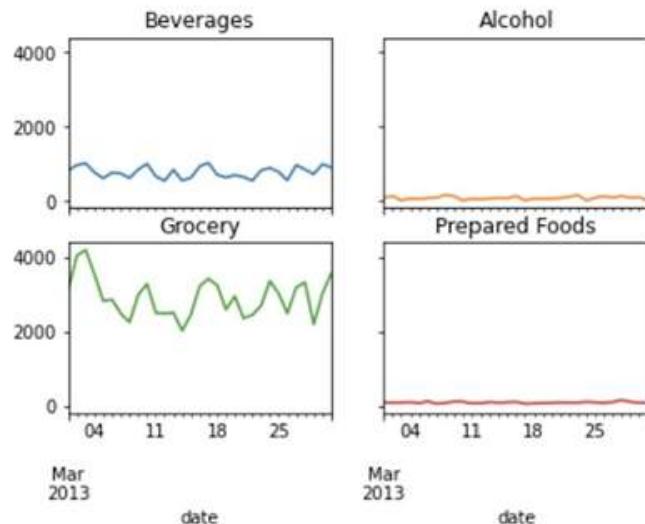
The subplots are stacked in a single column by default



SUBPLOTS

You can leverage **subplots** to create a separate chart for each series

```
sales_df.set_index("date").plot(  
    subplots=True,  
    layout=(2, 2),  
    sharey=True,  
    legend=False,  
    title=["Beverages", "Alcohol", "Grocery", "Prepared Foods"],  
)
```



There are several subplot options you can use:

- **layout** lets you specify the rows and columns for the subplots
- **sharey** & **sharex** let you set consistent axes across the subplots
- **title** can map a list of specified titles to each subplot

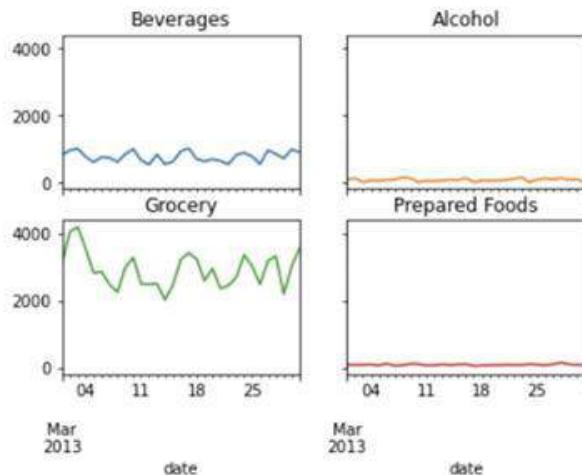


CHART SIZE

You can adjust the **size** of the plot or subplots (in inches) using “`figsize`”

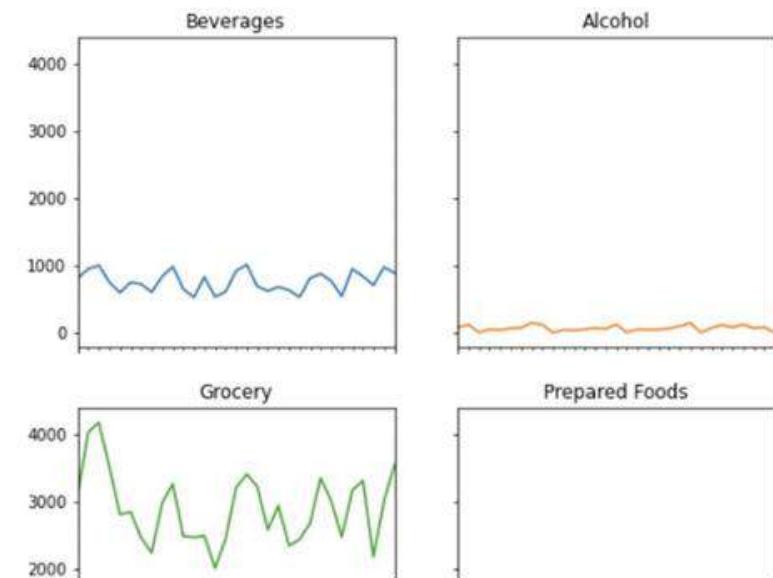
The default size is 6.4×4.8 inches

```
sales_df.set_index("date").plot(  
    subplots=True,  
    layout=(2, 2),  
    sharey=True,  
    legend=False,  
    title=["Beverages", "Alcohol", "Grocery", "Prepared Foods"],  
)
```



This has been resized to 8×8 inches

```
sales_df.set_index("date").plot(  
    figsize=(8, 8),  
    subplots=True,  
    layout=(2, 2),  
    sharey=True,  
    legend=False,  
    title=["Beverages", "Alcohol", "Grocery", "Prepared Foods"],  
)
```





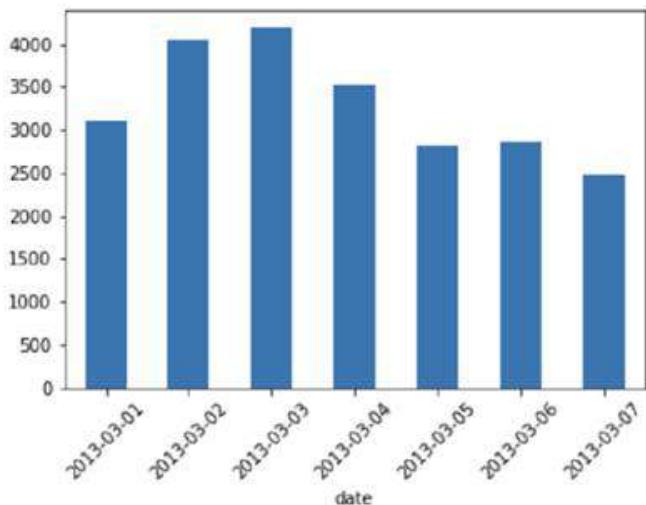
CHANGING CHART TYPES

You can **change chart types** with the “kind” argument or the attribute for each chart

This selects the chart type within the .plot() parameters

```
(sales_df.set_index(sales_df["date"].dt.date).iloc[:7, 3]
 .plot(kind='bar',
       y="grocery",
       rot=45))
```

<AxesSubplot:xlabel='date'>

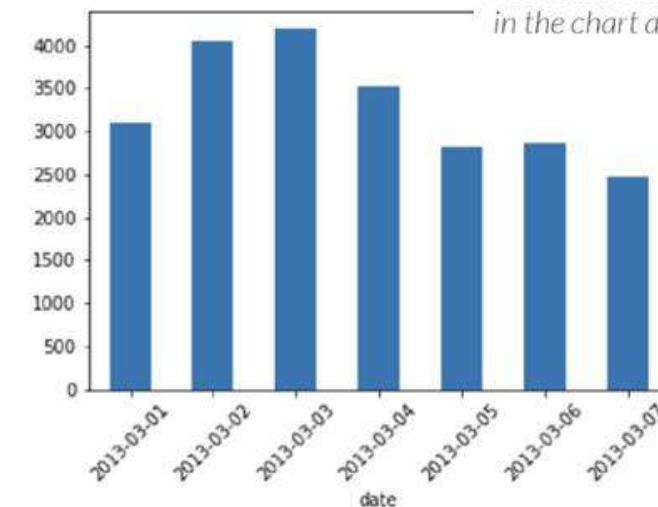


This chains the chart type as an attribute to .plot()

```
(sales_df.set_index(sales_df["date"].dt.date).iloc[:7, 3]
 .plot.bar(y="grocery",
           rot=45))
```

<AxesSubplot:xlabel='date'>

*Note that the arguments live
in the chart attribute now*



PREFERRED

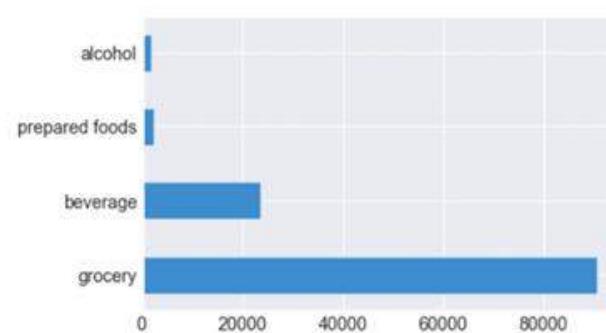
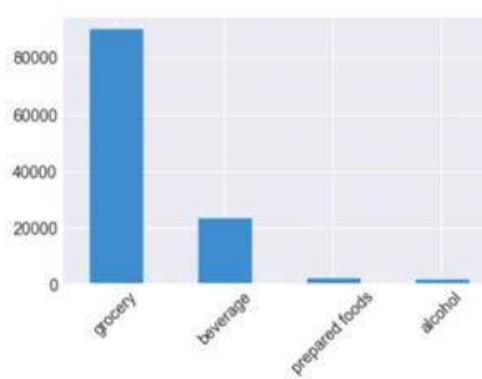


COMMON CHART TYPES

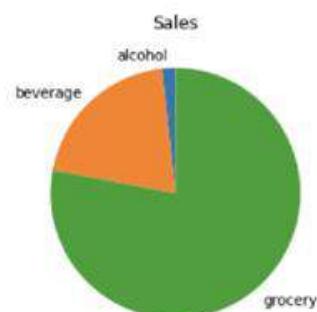
LINE CHART (*default*)



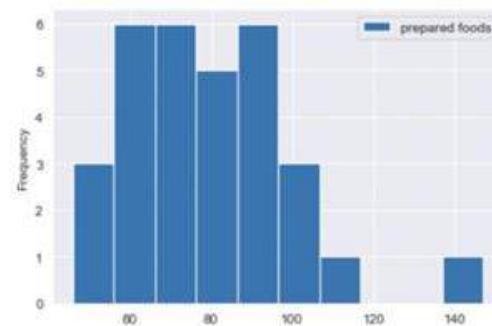
BAR CHART (bar or barh)



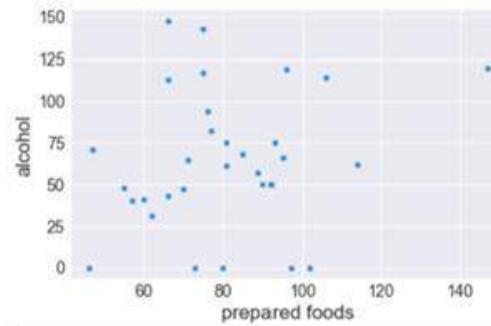
PIE CHART (pie)



HISTOGRAM (hist)



SCATTERPLOT (scatter)



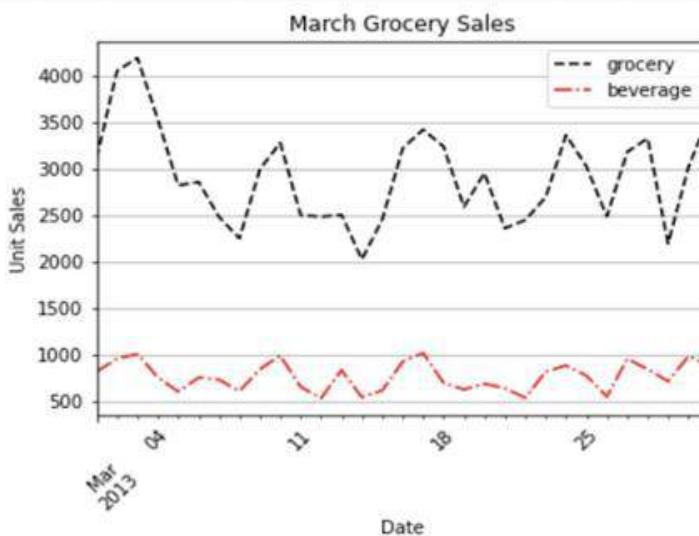


LINE CHARTS

Line charts are used for showing trends and changes over time

```
sales_df.set_index("date").loc[:, ["grocery", "beverage"]].plot(  
    title="March Grocery Sales",  
    xlabel="Date",  
    ylabel="Unit Sales",  
    color=["Black", "Red"],  
    style=[ "--", "-." ],  
    grid=True,  
    rot=45,  
);
```

They are the default chart type when using .plot()



The x-axis should be continuous!



BAR CHARTS

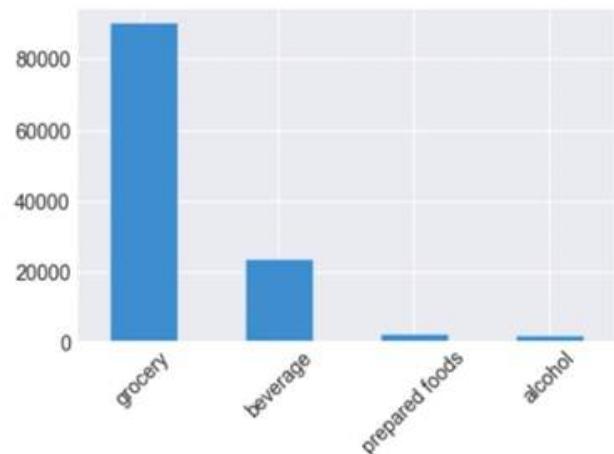
Bar charts are used for making comparisons with categorical data

```
sales_summary = (  
    sales_df[["beverage", "alcohol", "grocery", "prepared foods"]]  
    .sum()  
    .sort_values(ascending=False)  
)
```

```
sales_summary  
grocery      90429.0  
beverage     23567.0  
prepared foods 2490.0  
alcohol       2000.0  
dtype: float64
```

You'll typically aggregate the data by category first

```
sales_summary.plot.bar(rot=45)
```



"bar" will return a vertical bar chart (or column chart), and "rot" rotates the labels



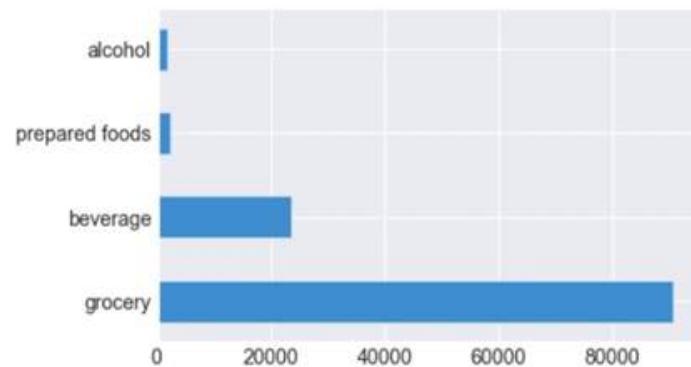
BAR CHARTS

Bar charts are used for making comparisons with categorical data

```
sales_summary = (  
    sales_df[["beverage", "alcohol", "grocery", "prepared foods"]]  
    .sum()  
    .sort_values(ascending=False)  
)
```

```
sales_summary  
grocery      90429.0  
beverage     23567.0  
prepared foods 2490.0  
alcohol       2000.0  
dtype: float64
```

```
sales_summary.plot.barh()
```



You'll typically aggregate the data by category first

"barh" will return a horizontal bar chart



Note that horizontal bar charts are sorted in reverse order



GROUPED BAR CHARTS

Plotting multiple series with a bar plot will create a **grouped bar chart**

store_sales_by_family

family	BEVERAGES	GROCERY I	LIQUOR,WINE,BEER	PREPARED FOODS
store_nbr				
1	783.5	1469.5	79.5	45.5000
2	1740.5	3672.0	52.5	100.7355
3	5180.0	8312.5	382.0	386.4820
4	1247.5	3209.5	116.0	116.7000

store_sales_by_family.plot.bar()



The bars are grouped by
the DataFrame index

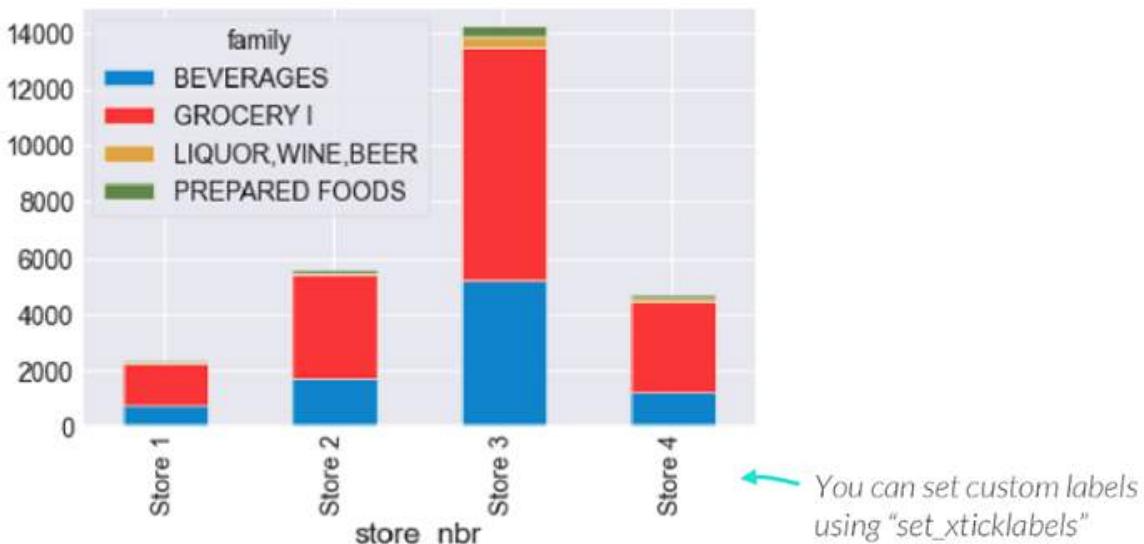


STACKED BAR CHARTS

Specify “`stacked=True`” when plotting multiple series to create a **stacked bar chart**

- This still lets you compare the categories, but also shows the composition of each category

```
(store_sales_by_family.plot.bar(stacked=True).set_xticklabels(  
    labels=["Store 1", "Store 2", "Store 3", "Store 4"]  
))
```



← You can set custom labels
using “`set_xticklabels`”



100% STACKED BAR CHARTS

Calculate percent of total values by index to create a **100% stacked bar chart**

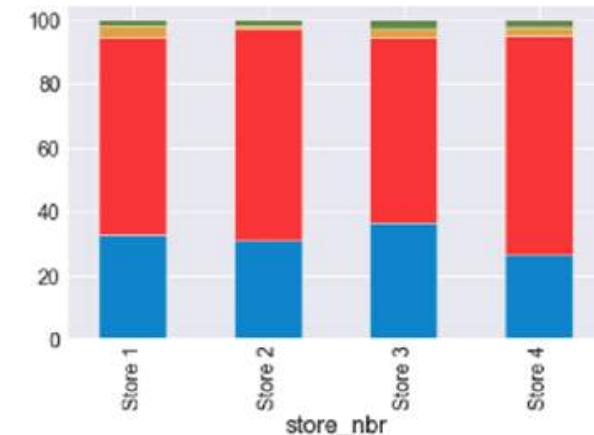
- This emphasizes the difference in composition between categories (*instead of absolute values*)

```
stacked_data = store_sales_by_family.apply(lambda x: x * 100 / sum(x), axis=1)  
stacked_data
```

family	BEVERAGES	GROCERY I	LIQUOR,WINE,BEER	PREPARED FOODS
store_nbr				
1	32.947855	61.795627	3.343146	1.913373
2	31.271698	65.975108	0.943272	1.809923
3	36.322884	58.288412	2.678637	2.710066
4	26.600849	68.437213	2.473506	2.488432

The values in each row add up to 100

```
(stacked_data.plot.bar(stacked=True, legend=False).set_xticklabels(  
    labels=["Store 1", "Store 2", "Store 3", "Store 4"]  
))
```

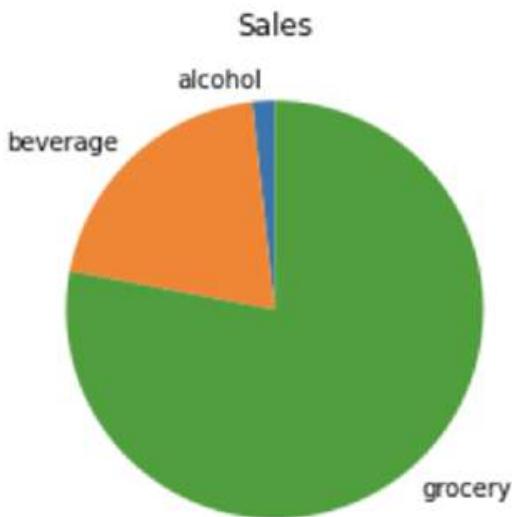




PIE CHARTS

Pie charts are used for showing composition with categorical data

```
(sales_df.loc[:, ["beverage", "alcohol", "grocery"]]  
    .sum()  
    .sort_values()  
    .plot.pie(title="Sales", ylabel="", startangle=90)  
)
```



PRO TIP: Pie charts get a bad rep, but they can be effective as long as:

- # of categories ≤ 5
- Slices are sorted
- First slice starts at 0°

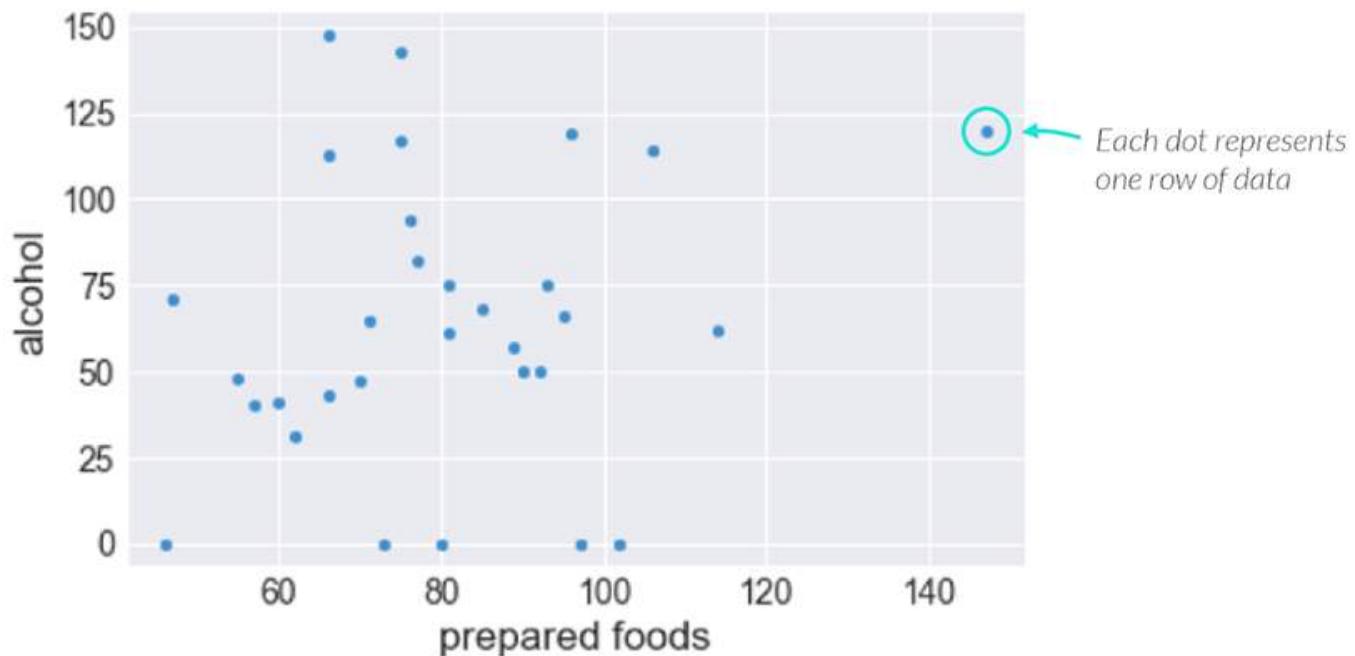


SCATTERPLOTS

Scatterplots are used for showing the relationship between numerical series

```
sales_df.plot.scatter(x="prepared foods", y="alcohol");
```

Just specify the x and y values!





SCATTERPLOTS

You can classify the dots by category by using color

```
sales_df.plot.scatter(  
    x="prepared foods",  
    y="alcohol",  
    title="Alcohol vs. Prepared Food Sales",  
    xlabel="Prepared Food Sales",  
    ylabel="Alcohol Sales",  
    c=sales_df["date"].dt.dayofweek,  
    colormap="Set2",  
)
```



This sets a different color
based on "dayofweek"

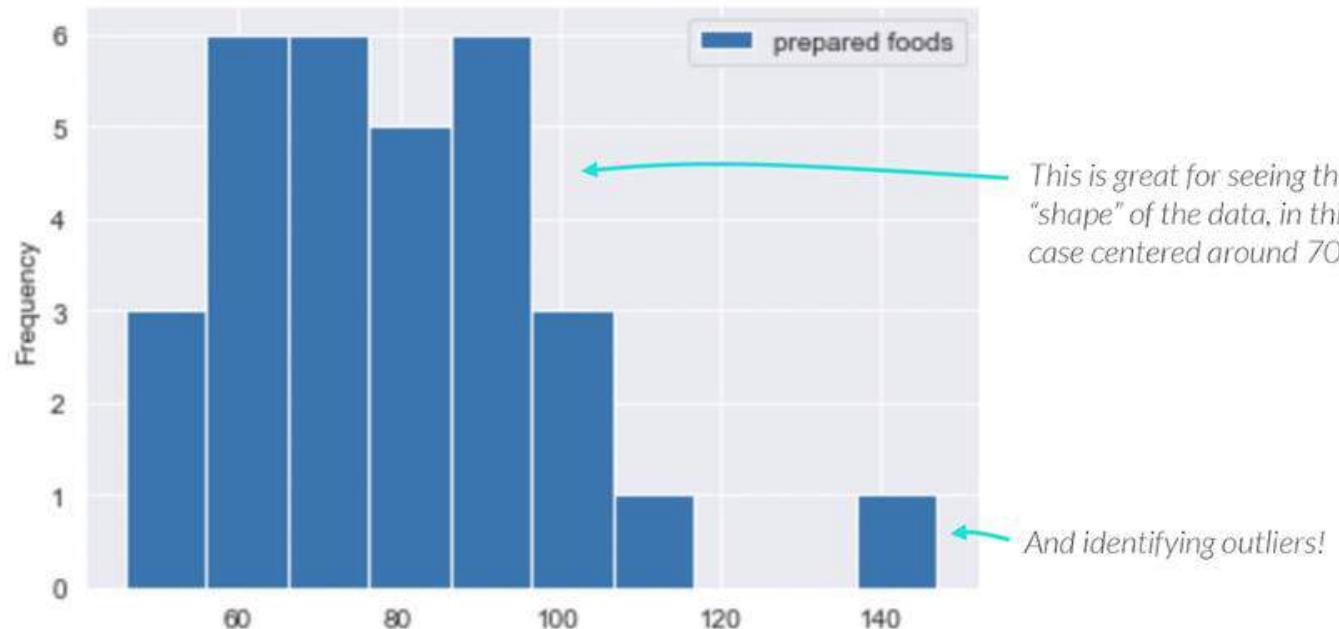


HISTOGRAMS

Histograms are used for showing the distribution of numerical series

- It divides the data in “bins” and plots the frequency of values that fall into each bin as bars

```
sales_df[["prepared foods"]].plot.hist()
```



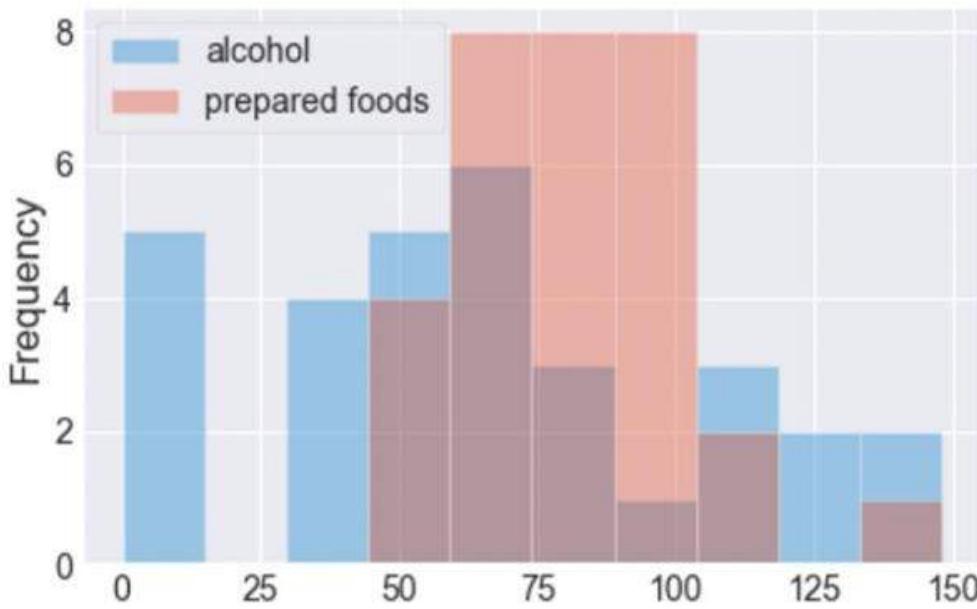


HISTOGRAMS

Histograms are used for showing the distribution of numerical series

- It divides the data in “bins” and plots the frequency of values that fall into each bin as bars

```
sales_df[["alcohol", "prepared foods"]].plot.hist(alpha=0.4)
```



“alpha” sets the transparency
(0 is invisible, 1 is solid)

Plotting multiple series lets you compare their distributions and understand drivers of statistics like mean, min, max and standard deviation



SAVING PLOTS

You can **save plots** as images with the `.figure.savefig()` method

```
plot = (
    sales_df.set_index("date")
    .loc[:, ["grocery", "beverage"]]
    .plot(
        title="March Grocery Sales",
        xlabel="Date",
        ylabel="Unit Sales",
        color=["Black", "Red"],
        style=[ "--", "-." ],
        grid=True,
    )
    .legend(bbox_to_anchor=(1.05, 1))
)

plot.figure.savefig("March_Grocery_Sales.png", bbox_inches="tight")
```

Just specify the file name & extension
(`bbox_inches="tight"` pads the image
to include the legend)



PRO TIP: You can also take a screenshot of the plot and save it





MORE DATA VISUALIZATION

There is **much more to data visualization** in Python, and most libraries are built to be compatible with Pandas Series and DataFrames

For more chart types, customization, and interactivity, consider these libraries:



Matplotlib is what we've used to visualize data so far, but there are more chart types and customization options available (*which can make it challenging to work with*)



Seaborn is an extension of Matplotlib that is easier to work with and produces better looking visuals – it also introduces new chart types!



Plotly is used for creating beautiful, interactive visuals, and has a sister app (Dash) that allows you to easily create dashboards as web applications



Folium is great for creating geospatial maps, especially since it contains features that are not available in the rest of the libraries

PREPARING FOR MODELING



DATA PREP: EDA VS MODELING

So far, we've gathered and cleaned data to **prepare for EDA**, but a few additional steps are required to **prepare for modeling**

Data ready for EDA

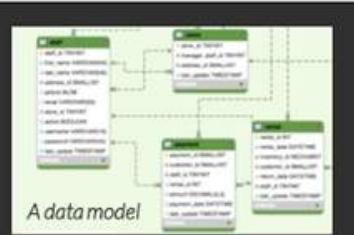
Address	City	Color	Price (\$)
200 N Michigan	Chicago	Blue	350,000
10 S State	Chicago	Blue	500,000
123 Main St	Evanston	White	180,000
50 Dempster	Evanston	Yellow	270,000

Data ready for modeling

Index	Walk Score	Blue	White	Price (\$)
0	97	1	0	350,000
1	92	1	0	500,000
2	70	0	1	180,000
3	64	0	0	270,000



Here, “modeling” refers to **applying an algorithm**, which is different from “data modeling”, where the goal is to visually show the relationship between the tables in a relational database





PREPARING FOR MODELING

To **prepare for modeling** means to transform the data into a structure and format that can be used as a direct input for a machine learning algorithm

You can prepare your data for modeling by:

- 1 Creating a **single table**
- 2 Setting the correct **row granularity**
- 3 Ensuring each **column** is non-null and numeric
- 4 **Engineering features** for the model



PREPARING COLUMNS FOR MODELING

Once you have the data in a single table with the right row granularity, you'll move on to **preparing the columns for modeling**:

1

All values should be **non-null**

- Use `df.info()` or `df.isna()` to identify null values and either remove them, impute them, or resolve them based on your domain expertise

2

All values should be **numeric**

- Turn text fields to numeric fields using dummy variables
- Turn datetime fields to numeric fields using datetime calculations



PRO TIP: There are some algorithms that can handle null and non-numeric values, including tree-based models and some classification models, but it is still best practice to prepare the data this way



DUMMY VARIABLES

A **dummy variable** is a field that only contains zeros and ones to represent the presence (1) or absence (0) of a value, also known as one-hot encoding

- They are used to transform a categorical field into multiple numeric fields

These dummy variables are **numeric representations** of the "Color" field

House ID	Price	Color	House ID	Price	Color	Blue	White	Yellow
1	\$350,000	Blue	1	\$350,000	Blue	1	0	0
2	\$500,000	Blue	2	\$500,000	Blue	1	0	0
3	\$180,000	White	3	\$180,000	White	0	1	0
4	\$270,000	Yellow	4	\$270,000	Yellow	0	0	1
5	\$245,000	White	5	\$245,000	White	0	1	0

It only takes 2 columns to distinguish 3 values, so you might sometimes see one less column ↑



DUMMY VARIABLES

Use `pd.get_dummies()` to create dummy variables in Python

houses

	House ID	Price	Color
0	1	350000	Blue
1	2	500000	Blue
2	3	180000	White
3	4	270000	Yellow
4	5	245000	White

```
# get dummy variables for all non-numeric fields  
pd.get_dummies(houses)
```

	House ID	Price	Color_Blue	Color_White	Color_Yellow
0	1	350000	1	0	0
1	2	500000	1	0	0
2	3	180000	0	1	0
3	4	270000	0	0	1
4	5	245000	0	1	0

```
# drop one of the dummy variable columns  
pd.get_dummies(houses, drop_first=True)
```

	House ID	Price	Color_White	Color_Yellow
0	1	350000	0	0
1	2	500000	0	0
2	3	180000	1	0
3	4	270000	0	1
4	5	245000	1	0

Set `drop_first=True` to remove one of the dummy variable columns
(it does not matter which column removed, as long as one is dropped)



PRO TIP: PREPARING DATETIME COLUMNS

To **prepare datetime columns** for modeling they need to be converted to numeric columns, but extracting datetime components isn't enough

	customer	purchase_date	month
0	Ava	2023-04-01	4
1	Ava	2023-04-01	4
2	Ava	2023-04-15	4
3	Ava	2023-05-01	5

The "month" field is not an appropriate numeric input because a model would interpret May (5) as being better than April (4)

Instead, you can prepare them using:

Dummy variables

	4	5	6
customer			
Aiden	4	4	1
Ava	3	1	0
Ben	2	0	1

Number of purchases by month

The days from “today”

	days_passed
customer	
Aiden	16
Ava	50
Ben	5

Days since the latest purchase

The average time between dates

	days_between
customer	
Aiden	7.625
Ava	10.000
Ben	30.500

Average days between purchases



PRO TIP: PREPARING DATETIME COLUMNS

EXAMPLE

Using **dummy variables** to prepare a date column for modeling

purchases

	customer	purchase_date	month
0	Ava	2023-04-01	4
1	Ava	2023-04-01	4
2	Ava	2023-04-15	4
3	Ava	2023-05-01	5
4	Ben	2023-04-15	4
...
105	Jenny	2023-04-20	4
106	Jenny	2023-04-20	4
107	Jenny	2023-04-20	4
108	Jenny	2023-04-20	4
109	Jenny	2023-04-20	4

110 rows x 3 columns



```
# create dummy variables and join with customers
month_dummies = pd.get_dummies(purchases.month)

monthly_purchases = pd.concat([purchases.customer, month_dummies], axis=1)
monthly_purchases.head(3)
```

	customer	4	5	6
0	Ava	1	0	0
1	Ava	1	0	0
2	Ava	1	0	0

```
# group by customer and sum the purchases by month
monthly_purchases.groupby('customer').sum().head()
```

	customer	4	5	6
	Aiden	4	4	1
	Ava	3	1	0
	Ben	2	0	1
	Bennett	5	0	0
	Blake	0	1	1



PRO TIP: PREPARING DATETIME COLUMNS

EXAMPLE

Using the *days from “today”* to prepare a date column for modeling

purchases		
	customer	purchase_date
0	Ava	2023-04-01
1	Ava	2023-04-01
2	Ava	2023-04-15
3	Ava	2023-05-01
4	Ben	2023-04-15
...
105	Jenny	2023-04-20
106	Jenny	2023-04-20
107	Jenny	2023-04-20
108	Jenny	2023-04-20
109	Jenny	2023-04-20

110 rows × 2 columns



```
# group by customer to get their latest purchase date  
last_purchase = purchases.groupby('customer').max()  
last_purchase.head(2)
```

purchase_date	
customer	
Aiden	2023-06-04
Ava	2023-05-01

```
# assume today is the max purchase date  
today = purchases.purchase_date.max()  
today
```

```
Timestamp('2023-06-20 00:00:00')
```

```
# calculate the days passed between today and their latest purchase  
last_purchase['days_passed'] = (today - last_purchase.purchase_date).dt.days  
last_purchase.head(3)
```

	purchase_date	days_passed
customer		
Aiden	2023-06-04	16
Ava	2023-05-01	50
Ben	2023-06-15	5



PRO TIP: PREPARING DATETIME COLUMNS

EXAMPLE

Using the *average time between dates* to prepare a date column for modeling

purchases

	customer	purchase_date
0	Ava	2023-04-01
1	Ava	2023-04-01
2	Ava	2023-04-15
3	Ava	2023-05-01
4	Ben	2023-04-15
...
105	Jenny	2023-04-20
106	Jenny	2023-04-20
107	Jenny	2023-04-20
108	Jenny	2023-04-20
109	Jenny	2023-04-20

110 rows × 2 columns



```
purchases['days_between'] = purchases.groupby('customer').diff(1)  
purchases.head()
```

	customer	purchase_date	days_between
0	Ava	2023-04-01	NaT
1	Ava	2023-04-01	0 days
2	Ava	2023-04-15	14 days
3	Ava	2023-05-01	16 days
4	Ben	2023-04-15	NaT

```
purchases['days_between'] = purchases['days_between'].dt.days  
purchases.groupby('customer')['days_between'].mean().head()
```

	customer	days_between
0	Aiden	7.625
1	Ava	10.000
2	Ben	30.500
3	Bennett	0.750
4	Blake	31.000



A variation of `.diff()` is `.shift()`, which will shift all the rows of a column up or down



FEATURE ENGINEERING

Feature engineering is the process of creating columns that you think will be helpful inputs for improving a model (*help predict, segment, etc.*)

- When preparing rows & columns for modeling you're already feature engineering!

`original_data.head()`

	customer	item_id	purchase_date	item_description	price	category	rating
0	Ava	1011	4/1/23	Paint	\$15.99	Arts & Crafts	3.5
1	Ava	1014	4/1/23	Brush	\$1.99	Arts & Crafts	4.2
2	Ava	1015	4/15/23	Paper	\$22.49	Arts & Crafts	4.5
3	Ava	1018	5/1/23	Scissors	\$3.50	Arts & Crafts	4.6
4	Ben	2345	4/15/23	Dog Food	\$29.99	Pet Supplies	4.9

`model_input.head()`

	customer	total_spend	Pet Supplies	days_since_purchase	june_purchases
0	Aiden	222.16	8	13.0	1.0
1	Ben	44.19	2	42.0	1.0
2	Blake	25.55	1	22.0	1.0
3	Calvin	29.99	1	16.0	1.0
4	Chloe	36.33	0	28.0	1.0

Other feature engineering techniques:

- Transformations (*log transform*)
- Scaling (*normalization & standardization*)
- Proxy variables



Once you have your data in a single table and have prepared the rows & columns, it's ready for modeling
However, being deliberate about **engineering new features** can be the difference between a good model and a great one

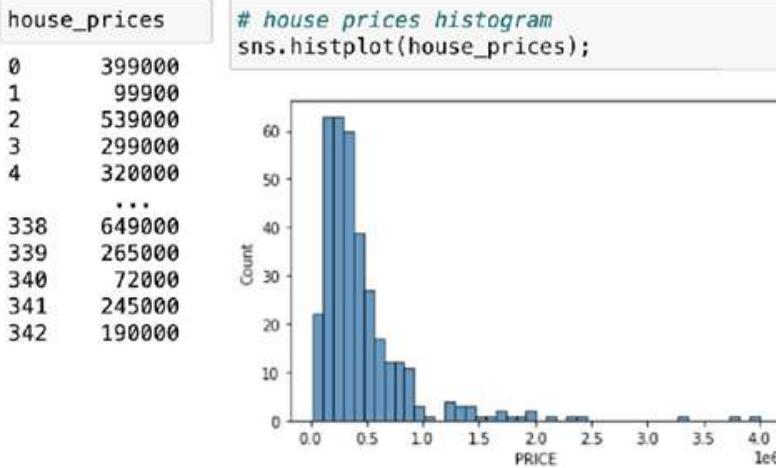


TRANSFORMATIONS: LOG TRANSFORMS

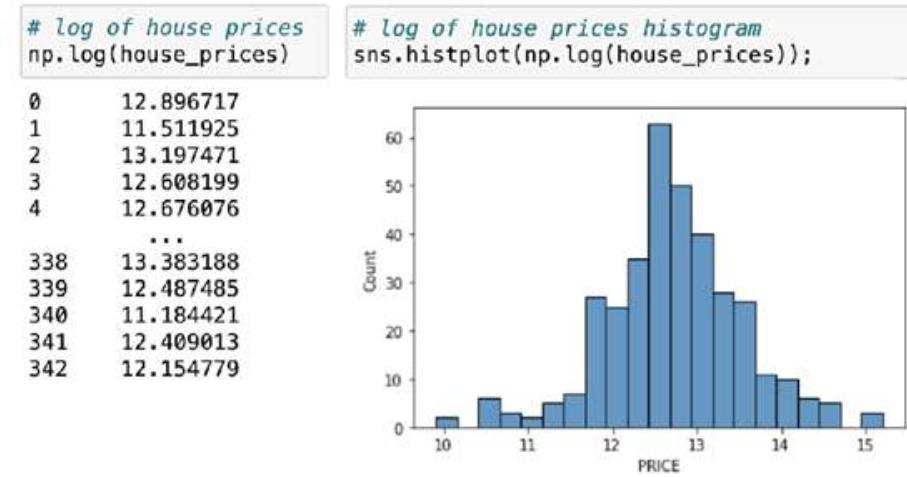
Transformations require mapping a set of values to another in a consistent way

- **Log transforms** turn skewed data into more normally-distributed data
- You can use the `np.log()` function to apply a log transform to a Series

Right-skewed data



Normally-distributed data





SCALING: NORMALIZATION

Scaling, as its name implies, requires setting all input features on a similar scale

- **Normalization** transforms all values to be between 0 and 1 (or between -1 and 1)

houses

	PRICE	BEDS
0	399000	3.0
1	99900	2.0
2	539000	3.0
3	299000	2.0
4	320000	2.0
5	699900	4.0
6	295000	3.0
7	245900	1.0
8	480000	5.0
9	375000	5.0
10	770000	3.0

(houses - houses.min()) / (houses.max() - houses.min())

	PRICE	BEDS
0	0.446351	0.50
1	0.000000	0.25
2	0.655275	0.50
3	0.297120	0.25
4	0.328458	0.25
5	0.895389	0.75
6	0.291151	0.50
7	0.217878	0.00
8	0.567229	1.00
9	0.410536	1.00
10	1.000000	0.50

Normalization equation

$$\frac{x - x_{min}}{x_{max} - x_{min}}$$

You can also use the `.MinMaxScaler()` function from the `sklearn` library



PRO TIP: Normalization is typically used when the distribution of the data is unknown



SCALING: STANDARDIZATION

Scaling, as its name implies, requires setting all input features on a similar scale

- **Normalization** transforms all values to be between 0 and 1 (or between -1 and 1)
- **Standardization** transforms all values to have a mean of 0 and standard deviation of 1

houses		
	PRICE	BEDS
0	399000	3.0
1	99900	2.0
2	539000	3.0
3	299000	2.0
4	320000	2.0
5	699900	4.0
6	295000	3.0
7	245900	1.0
8	480000	5.0
9	375000	5.0
10	770000	3.0

(houses - houses.mean()) / houses.std()		
	PRICE	BEDS
0	-0.061292	0.000000
1	-1.569570	-0.790569
2	0.644689	0.000000
3	-0.565564	-0.790569
4	-0.459667	-0.790569
5	1.456063	0.790569
6	-0.585735	0.000000
7	-0.833333	-1.581139
8	0.347168	1.581139
9	-0.182317	1.581139
10	1.809558	0.000000

Standardization equation

$$\frac{x - x_{mean}}{x_{std}}$$



You can also use the `StandardScaler()` function from the `sklearn` library



PRO TIP: Standardization is typically used when the distribution is normal (bell curve)



PROXY VARIABLES

A **proxy variable** is a feature meant to approximately represent another

- They are used when a feature is either difficult to gather or engineer into a new feature

Zip codes may look numeric, but should not be input into a model (60202 is not better than 60201)

	ZIP CODE	PRICE	BEDS	MEDIAN INCOME
0	60202	399000	3.0	\$92,433
1	60201	99900	2.0	\$81,128
2	60201	539000	3.0	\$81,128
3	60201	299000	2.0	\$81,128
4	60202	320000	2.0	\$92,433
5	60201	699900	4.0	\$81,128
6	60201	295000	3.0	\$81,128
7	60201	245900	1.0	\$81,128
8	60201	480000	5.0	\$81,128
9	60202	375000	5.0	\$92,433

Instead of turning the zip code into dummy variables, you can use a **proxy variable** like the median income for the zip code, or its distance from the city center



You may not be able to engineer proxy variables from existing data, but they can be gathered from external sources



FEATURE ENGINEERING TIPS

- 1 Anyone can apply an algorithm, but only someone with **domain expertise** can engineer relevant features, which is what makes a great model
- 2 You want your data to be long, not wide (**many rows, few columns**)
- 3 If you're working with customer data, a popular marketing technique is to engineer features related to the **recency, frequency, and monetary value** (RFM) of a customer's transactions
- 4 Once you start modeling, you're bound to find things you missed during data prep and will **continue to engineer features** and gather, clean, explore, and visualize the data