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

Dictionaries store key-value pairs, where keys are used to look up values

- **Keys** must be unique & immutable *(simple data types like strings are immutable)*
- **Values** do not need to be unique and can be any data type

EXAMPLE | *Looking up the inventory status for goggles*

```
inventory_status = {'skis': 'in stock',
                    'snowboard': 'sold out',
                    'goggles': 'sold out',
                    'boots': 'in stock'}

inventory_status['goggles']
```

```
'sold out'
```

Dictionaries are created with curly braces {}
Keys and values are separated by colons :
Key-value pairs are separated by commas ,

To retrieve dictionary values, simply enter the associated key

- *NOTE: You cannot look up dictionary values or indices*

*The **KeyError** will be returned if a given key is not in the dictionary*

```
inventory_status['in stock']
```

```
KeyError: 'in stock'
```

```
inventory_status[2]
```

```
KeyError: 2
```

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

*Prior to Python 3.7, python dictionaries were not ordered – which meant the order of key-value pairs could change each time we iterated through them This isn't the case anymore, but in some workplaces, you may encounter an older version of Python being used

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

ASSIGNMENT: DICTIONARY BASICS

NEW MESSAGE

February 10, 2022

From: **Alfie Alpine** *(Marketing Manager)*

Subject: **Birthday Snacks**

Good morning,

I'm glad we brought you on – I'm passing on the responsibility of managing birthday snacks to you.

Here are some of the changes needed:

- Can you let me know what Stuart's snack is?
- Add me – I like 'Cheese and Crackers'
- Nobody else wants raisins, can you change Jerry's snack to 'Fig Bars'?
- Remove Sierra – she doesn't work here

snack_dictionary.ipynb

Reply | Forward

Results Preview

```
snacks = {
    'Sally': 'Popcorn',
    'Ricard': 'Chocolate Ice Cream',
    'Stuart': 'Apple Pie',
    'Jerry': 'Raisins',
    'Sierra': 'Peanut Butter Cookies'
}
```

```
[redacted]
```

```
'Apple Pie'
```

```
[redacted]

snacks
```

```
{'Sally': 'Popcorn',
 'Ricard': 'Chocolate Ice Cream',
 'Stuart': 'Apple Pie',
 'Jerry': 'Fig Bars',
 'Alfie': 'Cheese and Crackers'}
```

ASSIGNMENT: DICTIONARY CREATION

NEW MESSAGE

February 10, 2022

From: **Sally Snow** *(Ski Shop Manager)*

Subject: **Inventory Status**

Hello!

We need a quick way for our sales staff to look up the inventory of our items (I've attached two lists).

Can you create a dictionary with each item as a key, with 'in stock' if the corresponding inventory value is greater than 0, and 'sold out' if the inventory value equals zero?

These lists are a small sample, so make sure you aren't manually coding the dictionary.

inventory_status.ipynb

Reply | Forward

Results Preview

```
items = ['skis', 'snowboard', 'goggles', 'boots']
inventory = [10, 0, 0, 7]

[redacted]

inventory_status
```

```
{'skis': 'in stock',
 'snowboard': 'sold out',
 'goggles': 'sold out',
 'boots': 'in stock'}
```

DICTIONARY METHODS

- List Limitations
- Dictionary Basics
- Modifying Dictionaries
- Dictionary Methods
- Nested Dictionaries
- Sets

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 they 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

ASSIGNMENT: DICTIONARY METHODS

NEW MESSAGE

February 10, 2022

From: **Stuart Slope** *(Business Analyst)*

Subject: **European Planning**

We're exploring the launch of our first store in Torino, Italy and I need help building out some of the data.

First, I need a dictionary containing the item numbers as keys, the number of sizes for each item as values.

Then, I need you to add items 10010 and 10011 with size counts of 4 and 7, respectively.

Finally, pull the prices out of the product dictionary and return a list with converted Euro pricing.

european_planning.ipynb

Reply | Forward

Results Preview

```
size_counts = {}
[redacted]

print(size_counts)
```

```
{10001: 1, 10002: 2, ... 10008: 3, 10009: 7}
```

```
[redacted]

print(size_counts)
```

```
{10001: 1, 10002: 2, ... 10009: 7, 10010: 4, 10011: 7}
```

```
euro_prices = []
[redacted]

euro_prices
```

```
[5.27, 8.79, 17.59, 21.99, 87.99, 70.39, 105.59, 87.99, 175.99]
```

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

ASSIGNMENT: ZIP

NEW MESSAGE

February 10, 2022

From: **Jerry Slush** *(IT Manager)*

Subject: **European Dictionary**

Hi there!

We're developing the architecture for our planned European store.

Can you create a dictionary by combining item_ids as keys with the lists item_names, euro_prices, item_category, and sizes as the values?

We'll upload this to our database shortly.

Exciting times!

european_dictionary.ipynb

Reply | Forward

Results Preview

```
euro_items
```

```
{10001: ('Coffee', 5.27, 'beverage', ['250mL']),
 10002: ('Beanie', 8.79, 'clothing', ['Child', 'Adult']),
 10003: ('Gloves', 17.59, 'clothing', ['Child', 'Adult']),
 10004: ('Sweatshirt', 21.99, 'clothing', ['XS', 'S', 'M', 'L', 'XL', 'XXL']),
 10005: ('Helmet', 87.99, 'safety', ['Child', 'Adult']),
 10006: ('Snow Pants', 70.39, 'clothing', ['XS', 'S', 'M', 'L', 'XL', 'XXL']),
 10007: ('Coat', 105.59, 'clothing', ['S', 'M', 'L']),
 10008: ('Ski Poles', 87.99, 'hardware', ['S', 'M', 'L']),
 10009: ('Ski Boots', 175.99, 'hardware', [5, 6, 7, 8, 9, 10, 11])}
```

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

ASSIGNMENT: NESTED DICTIONARIES

NEW MESSAGE

February 10, 2022

From: **Jerry Slush** *(IT Manager)*

Subject: **European Dictionary Updates**

Hi again!

We decided to restructure the dictionary you created earlier into a nested dictionary with each attribute represented by a key for fast lookup. Can you:

1. Verify the price on item 10009
2. Update the sizes for item 10009 to European sizing (they are stored in a list)
3. Create a dictionary based on the new one that includes item name as keys and sizes as values

modified_european_dictionary.ipynb

Reply | Forward

Results Preview

```
175.99
```

```
euro_data[10009]
```

```
{'name': 'Ski Boots',
 'price': 175.99,
 'category': 'hardware',
 'sizes': [37, 38, 39.5, 40.5, 41.5, 43.5, 44.5, 46.5]}
```

```
product_sizes
```

```
{'Coffee': ['250mL'],
 'Beanie': ['Child', 'Adult'],
 'Gloves': ['Child', 'Adult'],
 'Sweatshirt': ['XS', 'S', 'M', 'L', 'XL', 'XXL'],
 'Helmet': ['Child', 'Adult'],
 'Snow Pants': ['XS', 'S', 'M', 'L', 'XL', 'XXL'],
 'Coat': ['S', 'M', 'L'],
 'Ski Poles': ['S', 'M', 'L'],
 'Ski Boots': [37, 38, 39.5, 40.5, 41.5, 43.5, 44.5, 46.5
```

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

- List Limitations
- Dictionary Basics
- Modifying Dictionaries
- Dictionary Methods
- Nested Dictionaries
- 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
```

ASSIGNMENT: SETS

NEW MESSAGE

February 10, 2022

From: **Stuart Slope** *(Business Analyst)*

Subject: **Product Categories**

Hey,

I'm doing an analysis on product categories.

Can you collect the unique product category values from our European dictionary??

How many unique categories are there?

Once you have that, can you check if 'outdoor' is in there yet?

This will be really helpful, thanks!

product_categories.ipynb

Reply | Forward

Results Preview

```
print(unique_categories)
```

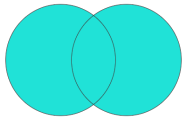
```
{'clothing', 'hardware', 'beverage', 'safety'}
```

```
4
```

```
False
```

SET OPERATIONS

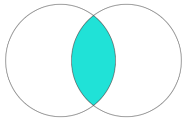
Python has useful **operations** that can be performed between sets



union

Returns all unique values in both sets

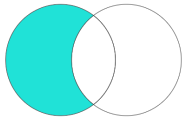
set1.union(set2)



intersection

Returns values present in both sets

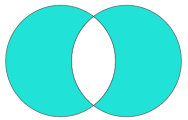
set1.intersection(set2)



difference

Returns values present in set 1, but not set 2

set1.difference(set2)



symmetric difference

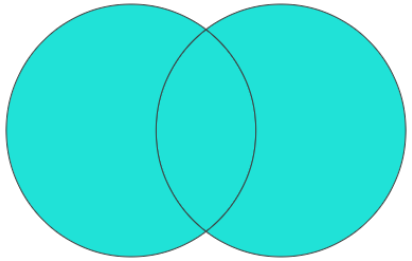
Returns values not shared between sets (opposite of intersection)

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

- List Limitations
- Dictionary Basics
- Modifying Dictionaries
- Dictionary Methods
- Nested Dictionaries
- Sets



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

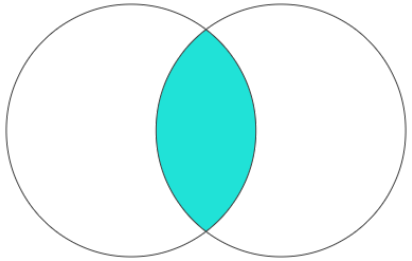
```
sunday_items = {'coffee'}

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

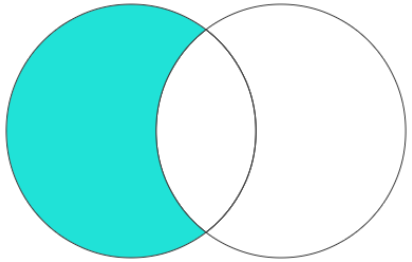
```
sunday_items = {'coffee'}

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 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

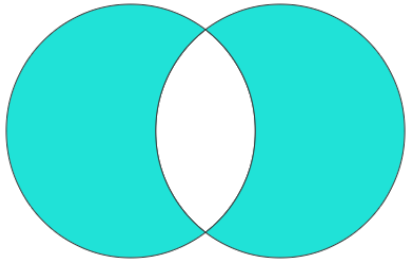
```
saturday_items - friday_items
```

```
{'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'}
```

ASSIGNMENT: SET OPERATIONS

NEW MESSAGE

February 10, 2022

From: **Stuart Slope** *(Business Analyst)*

Subject: **Weekend Sale Analysis**

Hey,

There are three lists with the customers who made a purchase on Friday, Saturday, and Sunday.

Can you get me the set of unique customers who made purchases on Saturday or Sunday?

Then return the customers that made purchases on Friday AND during the weekend; we want to target them with additional promotions.

weekend_sale_followup.ipynb

Reply Forward

Results Preview

```
weekend_set
```

```
{'C00002',
 'C00004',
 'C00006',
 'C00008',
 'C00010',
 'C00016',
 'C00017',
 'C00018',
 'C00019',
 'C00021',
 'C00022'}
```

```
{'C00004', 'C00006', 'C00008', 'C00010', 'C00016'}
```

KEY TAKEAWAYS

Dictionaries are used to store data in key-value pairs

- *Looking up values in dictionaries by their keys is more efficient than scanning through lists*
- *Dictionaries can mimic a structure like tabular data (e.g., Excel sheets).*

Dictionary methods are essential for working with dictionaries

- *The .keys(), .values(), and .items() methods are critical, but .get() and .update() are also very helpful*

Use the **zip()** function to create dictionaries with ease

- *The zip function stitches together multiple iterables into a single iterable of tuples*

Use **sets** to identify unique values or the relationships between datasets

- *Converting lists to sets can help perform these operations more efficiently*

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

- Function Components
- Defining Functions
- Variable Scope
- Modules
- Packages
- Lambda Functions
- Comprehensions

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

Input

Inputs to a function are known as **arguments**

Data, is some form, is usually one of the arguments passed into a function

Other arguments might be options for processing the data or formatting the output

Function

Functions are blocks of code that perform a specific task

When you use a function, it is known as a **function call**, or **calling a function**

The function itself is stored somewhere else, and you are 'calling' it by using its name

Output

Outputs are **values** returned by a function

Side Effects

Side effects are changes or actions made by the function, other than the output

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

THE ANATOMY OF MAX

- Function Components
- Defining Functions
- Variable Scope
- Modules
- Packages
- Lambda Functions
- Comprehensions

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()

Input

Possible arguments:

- A single iterable
- Multiple objects

Function

Identifies the largest value in the argument(s) provided

Output

The largest value in the argument(s)

Side Effects

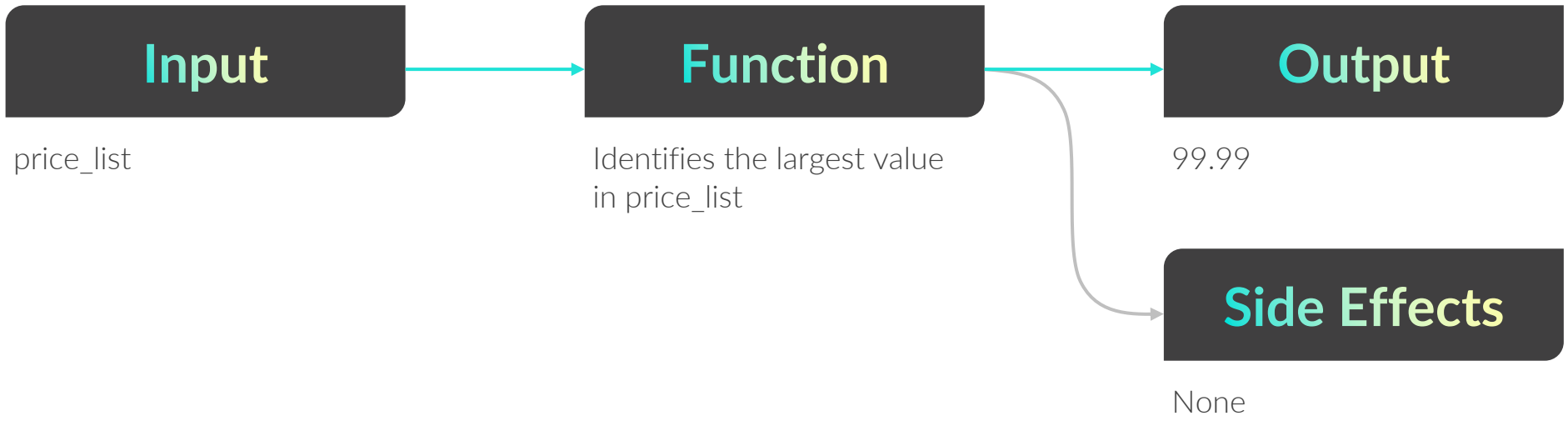
None

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 you function by

Examples:

- *avg*
- *usd_converter*

Variable names for the function inputs, separated by commas

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

- Function Components
- Defining Functions
- Variable Scope
- Modules
- Packages
- Lambda Functions
- Comprehensions

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?

ASSIGNMENT: DEFINING A FUNCTION

1 **NEW MESSAGE**

February 11, 2022

From: **Sally Snow** *(Ski Shop Manager)*

Subject: **Sales Tax Calculator**

Good morning,

We want to expand the work you've done on sales tax to all our stores, which requires a function to account for changes to taxes or expansion into new states.

Write a function that takes a given subtotal and tax rate, and returns the total amount owed (subtotal plus tax).

Thanks in advance!

tax_calculator_function.ipynb

Reply Forward

Results Preview

```
tax_calculator(100, .09)
```

```
109.0
```

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!'
```

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

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

```
'World! Hello'
```

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
```

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

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

```
'Hola World!'
```

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

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

```
'Hola Mundo!'
```

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

- Function Components
- Defining Functions
- Variable Scope
- Modules
- Packages
- Lambda Functions
- Comprehensions

**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)
```

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

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

exponentiator(**param_dict)
```

```
18
```

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
```

The values to return must be separated by commas

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

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

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?
```

You can unpack the tuple into variables during the function call

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

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

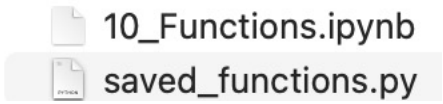
```
%%writefile saved_functions.py

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

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!*

```
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
```

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.concatenator('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 concatenator, multiplier

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

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

```
multiplier(5, 10)
```

```
50
```

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

ASSIGNMENT: CREATING A MODULE

NEW MESSAGE

February 11, 2022

From: **Sally Snow** *(Ski Shop Manager)*

Subject: **Modified Sales Tax Calculator**

Hey again,

Can you make the following changes to our sales tax calculator?

1. Since we've created this calculator for Stowe employees, set the default sales tax to 6%
2. We want the function to return subtotal, tax and sales tax in a list
3. Save this function to tax_calculator.py

Thanks!

modified_tax_calculator.ipynb

Reply | Forward

Results Preview

```
tax_calculator(100)
```

```
[100, 6.0, 106.0]
```

ASSIGNMENT: IMPORTING A MODULE

NEW MESSAGE

February 11, 2022

From: **Sally Snow** *(Ski Shop Manager)*

Subject: **Applied Sales Tax Calculator**

Hey again,

Now that you've updated and saved the function, can you import it, pass a list of transactions through, and create a list containing the transaction information for each?

Once you've done that, I'd like you to create a dictionary with the supplied customer IDs as keys and the transaction information as values.

Thanks!

applied_tax_calculator.ipynb

Reply | Forward

Results Preview

```
subtotals = [15.98, 899.97, 799.97, 117.96, 5.99, 599.99]
[redacted]

full_transactions
```

```
[[15.98, 0.96, 16.94],
 [899.97, 54.0, 953.97],
 [799.97, 48.0, 847.97],
 [117.96, 7.08, 125.04],
 [5.99, 0.36, 6.35],
 [599.99, 36.0, 635.99]]
```

```
customer_ids = ['C00004', 'C00007', 'C00015', 'C00016', 'C00020', 'C00010']

[redacted]

customer_dict
```

```
{'C00004': [15.98, 0.96, 16.94],
 'C00007': [899.97, 54.0, 953.97],
 'C00015': [799.97, 48.0, 847.97],
 'C00016': [117.96, 7.08, 125.04],
 'C00020': [5.99, 0.36, 6.35],
 'C00010': [599.99, 36.0, 635.99]}
```

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

- *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*

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

These are the two functions inside the module

IMPORTING EXTERNAL FUNCTIONS

EXAMPLE *Importing the math package, which contains handy mathematical functions*

```
import math

dir(math)
```

```
['__doc__',
 '__file__',
 '__loader__',
 '__name__',
 '__package__',
 '__spec__',
 'acos',
 'acosh',
 'asin',
 'asinh',
 'atan',
 'atan2',
 'atanh',
```

This is arc cosine

This package has a LOT of functions

(too many for a single page!)

What does math.sqrt return?

```
math.sqrt(81)
```

```
9.0
```

It's helpful to look up the official documentation, which will usually give a helpful overview of the package and its contents

docs.python.org/3/library/math.html

PRO TIP: Import packages with an alias using the **as** keyword – this saves keystrokes while still explicitly referencing the package to help avoid naming conflicts

```
import math as m

m.sqrt(81)
```

```
9.0
```

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
```

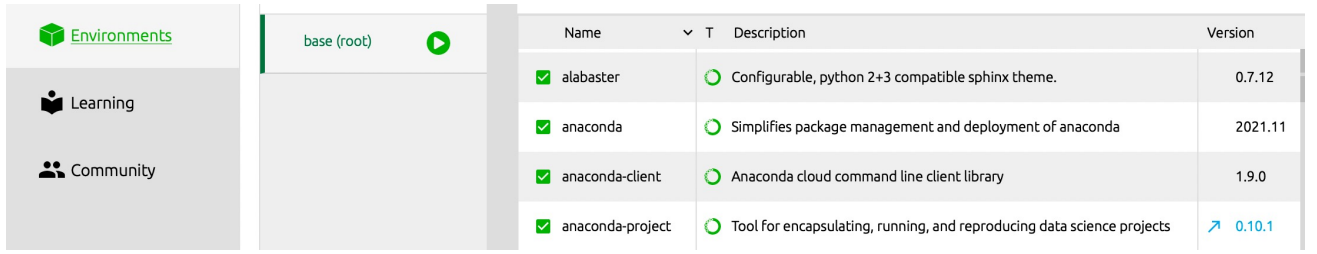
Adding this extra code snippet ensures the package gets installed correctly

PRO TIP: MANAGING PACKAGES

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

```
!conda list
notebook                  6.4.8            py39hecd8cb5_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*



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

```
import sys

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

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

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:

NumPy

matplotlib

seaborn

pandas

plotly

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>
```

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

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

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

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

ASSIGNMENT: MAP

NEW MESSAGE

February 11, 2022

From: **Sally Snow** *(Ski Shop Manager)*

Subject: **Re: Applied Sales Tax Calculator**

Hey again,

We're really making some great progress on implementing Python – can you import the tax calculator and apply it to the newest list of transactions?

Don't use a for loop!

Thanks!

mapping_the_calculator.ipynb

Reply | Forward

Results Preview

```
from tax_calculator import tax_calculator

subtotals = [1799.94, 99.99, 254.95, 29.98, 99.99]

list(map(tax_calculator, subtotals))
```

```
[[1799.94, 108.0, 1907.94],
 [99.99, 6.0, 105.99],
 [254.95, 15.3, 270.25],
 [29.98, 1.8, 31.78],
 [99.99, 6.0, 105.99]]
```

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.

ASSIGNMENT: LAMBDA FUNCTIONS

NEW MESSAGE

February 11, 2022

From: **Alfie Alpine** *(Marketing Manager)*

Subject: **One-time discount**

Hey,

We need to apply a 10% discount for customers who purchased more than $500 worth of product – they're starting to complain it hasn't been applied.

This is a one-time promotion, so can you code up something quick?

We won't reuse it.

Thanks!

discount_orders.ipynb

Reply | Forward

Results Preview

```
subtotals = [15.98, 899.97, 799.97, 117.96, 5.99, 599.99]

discounted_subtotals = list(
    [redacted]
)

discounted_subtotals
```

```
[15.98, 809.97, 719.97, 117.96, 5.99, 539.99]
```

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
```

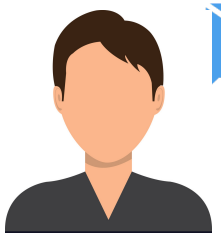
```
[5.27, 8.79, 17.59, 21.99, 87.99]
```

```
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!

ASSIGNMENT: COMPREHENSIONS



NEW MESSAGE

February 11, 2022

From: **Ricardo Nieva** *(Financial Planner)*

Subject: **Cost of European Items**

Hi there,

I'm working on some numbers for our European expansion. Can you help me calculate the combined cost of all of our European items?

The data is stored in the euro_data dictionary.

Thanks!

cost_of_european_items.ipynb

Reply | Forward

Results Preview

```
euro_data
```

```
{10001: ['Coffee', 5.27, 'beverage', ['250mL']],
 10002: ['Beanie', 8.79, 'clothing', ['Child', 'Adult']],
 10003: ['Gloves', 17.59, 'clothing', ['Child', 'Adult']],
 10004: ['Sweatshirt', 21.99, 'clothing', ['XS', 'S', 'M', 'L', 'XL', 'XXL']],
 10005: ['Helmet', 87.99, 'safety', ['Child', 'Adult']],
 10006: ['Snow Pants', 70.39, 'clothing', ['XS', 'S', 'M', 'L', 'XL', 'XXL']],
 10007: ['Coat', 105.59, 'clothing', ['S', 'M', 'L']],
 10008: ['Ski Poles', 87.99, 'hardware', ['S', 'M', 'L']],
 10009: ['Ski Boots', 175.99, 'hardware', [5, 6, 7, 8, 9, 10, 11]]}
```

```
581.59
```

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!

ASSIGNMENT: DICTIONARY COMPREHENSIONS

NEW MESSAGE

February 11, 2022

From: **Jerry Slush** *(IT Manager)*

Subject: **Final Tax Calculator**

Hey there,

We're getting close to having a final ETL process for our European dictionary. Can you create a function that applies our tax calculator to a list of subtotals and matches the output up with customer_IDs?

Store them in a dictionary, with customer IDs as keys, and the transactions as values.

Thanks – we're getting close to implementing this!

final_tax_calculator.ipynb

Reply | Forward

Results Preview

```
from tax_calculator import tax_calculator

customer_ids = ['C00004', 'C00007', 'C00015', 'C00016', 'C00020', 'C00010']

subtotals = [15.98, 899.97, 799.97, 117.96, 5.99, 599.99]
```

```
transaction_dict_creator(customer_ids, subtotals, .08)
```

```
{'C00004': [15.98, 1.28, 17.26],
 'C00007': [899.97, 72.0, 971.97],
 'C00015': [799.97, 64.0, 863.97],
 'C00016': [117.96, 9.44, 127.4],
 'C00020': [5.99, 0.48, 6.47],
 'C00010': [599.99, 48.0, 647.99]}
```

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)
```

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, exchange_rate=.85) for price in price_list]
```

In the comprehension, the exchange_rate argument is easy to specify

```
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]
```

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.

KEY TAKEAWAYS

Functions are reusable blocks of code that make us more efficient analysts

- *If you find yourself writing the same code over and over again, consider making it a function*

External **packages** & **libraries** provide access to functions developed by others

- *Most data analysts work with features from packages like pandas and matplotlib regularly*

Map() applies a function to an iterable without the use of a for loop

- *This is often paired with lambda functions for handy one-off procedures*

Comprehensions can create sequences and dictionaries from other iterables

- *They are more efficient than map() at applying a function to a list, and allow you to specify different arguments*

MANIPULATING EXCEL SHEETS

MANIPULATING EXCEL SHEETS

In this section we'll import the openpyxl package and **manipulate data from Excel sheets** by using the Python skills learned throughout the course

TOPICS WE'LL COVER:

- The openpyxl Package
- Navigating Workbooks
- Looping Through Cells
- Modifying Cells

GOALS FOR THIS SECTION:

- Read and save Excel files in Python
- Navigate workbooks, worksheets, and cells
- Loop through cell ranges and modify cell values

THE OPENPYXL PACKAGE

The **openpyxl** package is designed for reading and writing Excel files

Without ever needing to open Excel itself, you can use openpyxl to:

- Create, modify, or delete workbooks, worksheets, rows, or columns
- Leverage custom Python functions or Excel formulas *(yes, really!)*
- Automate Excel chart creation
- … or do almost anything else that can be done in Excel natively

Example use cases:

- Cleaning or manipulating Excel sheets before uploading them into a database
- Automating database pulls (sqlalchemy library) and creating Excel sheets for end users
- Summarizing Excel data before sending it to another user

openpyxl **should be installed** in Anaconda already, but if it isn't you can install it like you would any other package

THE MAVEN SKI SHOP DATA

Orders_info

	A	B	C	D	E	F	G	H
1	Order_ID	Customer_ID	Order_Date	Subtotal	Tax	Total	Location	Items_Ordered
2	100000	C00004	11/26/2021	15.98			Sun Valley	10001, 10002
3	100001	C00007	11/26/2021	899.97			Stowe	10008, 10009, 10010
4	100002	C00015	11/26/2021	799.97			Mammoth	10011, 10012, 10013
5	100003	C00016	11/26/2021	117.96			Stowe	10002, 10003, 10004, 10006
6	100004	C00020	11/26/2021	5.99			Sun Valley	10001
7	100005	C00010	11/26/2021	599.99			Mammoth	10010
8	100006	C00006	11/26/2021	24.99			Mammoth	10004
9	100007	C00001	11/26/2021	1799.94			Mammoth	10008, 10008, 10009, 10009, 10009, 10010, 10010
10	100008	C00003	11/26/2021	99.99			Sun Valley	10005
11	100009	C00014	11/26/2021	254.95			Sun Valley	10002, 10003, 10004, 10006, 10007
12	100010	C00001	11/26/2021	29.98			Mammoth	10002, 10003
13	100011	C00001	11/26/2021	99.99			Mammoth	10005
14	100012	C00005	11/26/2021	25.98			Sun Valley	10001, 10003
15	100013	C00008	11/26/2021	649.98			Stowe	10012, 10013
16	100014	C00013	11/26/2021	89.99			Sun Valley	10014
17	100020	C00004	11/27/2021	119.99			Sun Valley	10007
18	100021	C00017	11/27/2021	599.99			Stowe	10010
19	100022	C00019	11/27/2021	649.98			Sun Valley	10012, 10013
20	100023	C00002	11/27/2021	24.99			Stowe	10004
21	100024	C00008	11/27/2021	99.99			Stowe	10005
22	100025	C00021	11/27/2021	99.99			Mammoth	10008
23	100026	C00022	11/27/2021	5.99			Sun Valley	10001
24	100027	C00006	11/28/2021	24.99			Mammoth	10002
25	100031	C00018	11/28/2021	999.96			Stowe	10005, 10008, 10009, 10010
26	100032	C00018	11/28/2021	99.99			Stowe	10006
27	100033	C00010	11/28/2021	399.97			Mammoth	10005, 10008, 10009
28	100034	C00016	11/28/2021	89.99			Stowe	10014

MAVEN
SKI SHOP

Item_info

Product_ID	Product_Name	Price	Cost	Available Sizes
10001	Coffee	5.99	0.99	250mL
10002	Beanie	9.99	4.29	Child, Adult
10003	Gloves	19.99	7.99	Child, Adult
10004	Sweatshirt	24.99	10.59	XS, S, M, L , XL, XXL
10005	Helmet	99.99	49.99	Child, Adult
10006	Snow Pants	79.99	32.49	XS, S, M, L , XL, XXL
10007	Coat	119.99	54.55	S, M, L
10008	Ski Poles	99.99	69.99	S, M, L
10009	Ski Boots	199.99	89.99	5,6,7,8,9,10,11
10010	Skis	599.99	249.99	S, M, L
10011	Snowboard Boo	129.99	64.99	5,6,7,8,9,10,11
10012	Bindings	149.99	89.99	NA
10013	Snowboard	499.99	199.99	S, M, L, Powder

NAVIGATING WORKBOOKS

The **load_workbook()** function reads in an Excel workbook

```
import openpyxl as xl

workbook = xl.load_workbook(filename='maven_ski_shop_data.xlsx')
```

*Just supply a file name!**

workbook.sheetnames returns the worksheet names in a workbook

```
workbook.sheetnames
```

```
['Item_Info', 'Inventory_Levels', 'Orders_Info']
```

Note that 'workbook' is simply a variable name that stores the workbook object

Using this intuitive name makes the code easier to read, but any name can be used instead (wb is common as well)

* *As long as the Excel workbook is in the same folder as your Jupyter notebook*

NAVIGATING WORKSHEETS

- The openpyxl Package
- Navigating Workbooks
- Iterating Through Cells
- Modifying Cells

workbook.active returns the name of the worksheet openpyxl is pointed to

```
workbook.active
```

```
<Worksheet "Orders_Info">
```

```
workbook.active = 1
workbook.active
```

```
<Worksheet "Inventory_Levels">
```

*The first sheet (value of 0) is active by default, but you can change it by assigning the sheet's index to **workbook.active***

```
workbook.active.title
```

```
'Orders_Info'
```

Add .title to return the title as text

Sheets can also be **referenced by name** (like dictionary keys)

```
workbook['Item_Info']
```

```
<Worksheet "Item_Info">
```

```
items = workbook['Item_Info']
inventory = workbook['Inventory_Levels']
orders = workbook['Orders_Info']
```

Assign sheet names to variables to make the workbook easier to navigate

NAVIGATING CELLS

- The openpyxl Package
- Navigating Workbooks
- Iterating Through Cells
- Modifying Cells

You can **navigate cells** by using 'A1' style coordinates, or Python-esque indices

sheet['coordinate'] returns the specified cell object, and **cell.value** returns the cell's contents

```
items['B1']
```

```
<Cell 'Item_Info'.B1>
```

Remember that 'items' is the name assigned to workbook['Item_Info']

```
print(items["B1"].value)
print(items["B4"].value)
```

```
Product_Name
Gloves
```

sheet.cell(row= , column=) returns cell objects as well

```
items.cell(row=4, column=2).value
```

This is equivalent to 'B4'
(0 indexing doesn't apply!)

'Item_Info' worksheet

	A	B	C	D	E
1	Product_ID	Product_Name	Price	Cost	Available Sizes
2	10001	Coffee	5.99	0.99	250mL
3	10002	Beanie	9.99	4.29	Child, Adult
4	10003	Gloves	19.99	7.99	Child, Adult
5	10004	Sweatshirt	24.99	10.59	XS, S, M, L , XL, XXL
6	10005	Helmet	99.99	49.99	Child, Adult
7	10006	Snow Pants	79.99	32.49	XS, S, M, L , XL, XXL
8	10007	Coat	119.99	54.55	S, M, L
9	10008	Ski Poles	99.99	69.99	S, M, L
10	10009	Ski Boots	199.99	89.99	5,6,7,8,9,10,11
11	10010	Skis	599.99	249.99	S, M, L
12	10011	Snowboard Boo	129.99	64.99	5,6,7,8,9,10,11
13	10012	Bindings	149.99	89.99	NA
14	10013	Snowboard	499.99	199.99	S, M, L, Powder

ASSIGNMENT: NAVIGATING WORKBOOKS

NEW MESSAGE

February 12, 2022

From: **Sally Snow** *(Ski Shop Manager)*

Subject: **Tax Calculation**

Hey there, we've just run our annual Black Friday Sale, and there have been issues with the data. Most of our data team is skiing this weekend, so we need your help.

A customer mentioned they weren't charged sales tax and have graciously reached out to pay it.

Can you calculate the sales tax (8%) and total for customer C00003? It should be in row 10.

Thanks!

excel_data_sales_tax.ipynb

Reply | Forward

Results Preview

```
import openpyxl as xl
```

```
Sales Tax: $8.0
Total: $107.99
```

DETERMINING SHEET RANGES

- The openpyxl Package
- Navigating Workbooks
- Looping Through Cells
- Modifying Cells

sheet.max_row and **sheet.max_column** help determine the number of rows and columns with data in a worksheet, to then use as stopping conditions for loops

```
items.max_row
```

```
14
```

This returns the index of the last row with data

```
items.max_column
```

```
5
```

This returns the index of the last column with data

('E' is the fifth column)

	A	B	C	D	E
1	Product_ID	Product_Name	Price	Cost	Available Sizes
2	10001	Coffee	5.99	0.99	250mL
3	10002	Beanie	9.99	4.29	Child, Adult
4	10003	Gloves	19.99	7.99	Child, Adult
5	10004	Sweatshirt	24.99	10.59	XS, S, M, L , XL, XXL
6	10005	Helmet	99.99	49.99	Child, Adult
7	10006	Snow Pants	79.99	32.49	XS, S, M, L , XL, XXL
8	10007	Coat	119.99	54.55	S, M, L
9	10008	Ski Poles	99.99	69.99	S, M, L
10	10009	Ski Boots	199.99	89.99	5,6,7,8,9,10,11
11	10010	Skis	599.99	249.99	S, M, L
12	10011	Snowboard Boo	129.99	64.99	5,6,7,8,9,10,11
13	10012	Bindings	149.99	89.99	NA
14	10013	Snowboard	499.99	199.99	S, M, L, Powder

LOOPING THROUGH CELLS

Excel columns usually contain data fields, while rows contain individual records

To **loop through cells** in a column, you need to move row by row in that column

```
for row in range(1, items.max_row + 1):
    print(f'B{row}', items[f'B{row}'].value)
```

```
B1 Product_Name
B2 Coffee
B3 Beanie
B4 Gloves
B5 Sweatshirt
B6 Helmet
B7 Snow Pants
B8 Coat
B9 Ski Poles
B10 Ski Boots
B11 Skis
B12 Snowboard Boots
B13 Bindings
B14 Snowboard
```

How does this code work?

- The for loop is iterating through each **row** in a specified range
- Since the end of **range** is not inclusive, we need to stop at **items.max_row** + 1
- The range goes from 1-14, so the loop will run 14 times

WRITING DATA TO CELLS

- The openpyxl Package
- Navigating Workbooks
- Looping Through Cells
- Modifying Cells

You can **write data to a cell** by assigning a value to it using the cell's coordinates

```
items['F1'].value
```

```
items['F1'] = 'Euro Price'
```

```
items['F1'].value
```

```
'Euro Price'
```

Cell 'F1' starts with no data, and then a value of 'Euro Price' is assigned to it

```
items.max_column
```

```
6
```

Now that column 'F' has data, max_column has increased to 6

WRITING DATA TO A COLUMN

You can **write data to a column** by looping through its rows and assigning values

```
exchange_rate = .88

for row in range(2, items.max_row + 1):
    items[f'F{row}'] = round(items[f'C{row}'].value * exchange_rate, 2)
```

Each row in column 'F' is now equal to the matching value in column 'C' multiplied by .88

Note that the range is starting at 2, since the first row is reserved for the column header

```
for index, cell in enumerate(items['F'], start=1):
    print(f'F{index}', cell.value)
```

You can reference all the cells in a column!

```
F1 Euro Price
F2 5.27
F3 8.79
F4 17.59
F5 21.99
F6 87.99
F7 70.39
F8 105.59
F9 87.99
F10 175.99
F11 527.99
F12 114.39
F13 131.99
F14 439.99
```

Use **enumerate's** start argument to start the index at 1, rather than the default of 0, to align with Excel row numbers

ASSIGNMENT: WRITING DATA TO A COLUMN

NEW MESSAGE

February 12, 2022

From: **Sally Snow** *(Ski Shop Manager)*

Subject: **New Currency Prices**

Hi again!

In addition to a planned EU expansion this year, we're considering expanding into Japan and the UK next year.

Since we're going to do this a few times, can you create a currency converter function?

Once we have that, create a column for 'GBP Price' and 'JPY Price', to store Pound and Yen prices.

The notebook has conversion rates and more details.

pound_and_yen_pricing.ipynb

Reply | Forward

Results Preview

```
def currency_converter(price,
```

Product_ID	Product_Name	Price	Cost	Available Sizes	Euro Price	GBP Price	JPY Price
10001	Coffee	5.99	0.99	250mL	5.27	4.55	736.77
10002	Beanie	9.99	4.29	Child, Adult	8.79	7.59	1228.77
10003	Gloves	19.99	7.99	Child, Adult	17.59	15.19	2458.77
10004	Sweatshirt	24.99	10.59	XS, S, M, L , XL, XXL	21.99	18.99	3073.77
10005	Helmet	99.99	49.99	Child, Adult	87.99	75.99	12298.77
10006	Snow Pants	79.99	32.49	XS, S, M, L , XL, XXL	70.39	60.79	9838.77
10007	Coat	119.99	54.55	S, M, L	105.59	91.19	14758.77
10008	Ski Poles	99.99	69.99	S, M, L	87.99	75.99	12298.77
10009	Ski Boots	199.99	89.99	5,6,7,8,9,10,11	175.99	151.99	24598.77
10010	Skis	599.99	249.99	S, M, L	527.99	455.99	73798.77
10011	Snowboard Boo	129.99	64.99	5,6,7,8,9,10,11	114.39	98.79	15988.77
10012	Bindings	149.99	89.99	NA	131.99	113.99	18448.77
10013	Snowboard	499.99	199.99	S, M, L, Powder	439.99	379.99	61498.77

INSERTING COLUMNS

- The openpyxl Package
- Navigating Workbooks
- Looping Through Cells
- Modifying Cells

You can **insert columns** to a worksheet without overwriting existing data

sheet.insert_cols(idx=index) inserts a column in the specified sheet and index

```
items['E1'].value
```

```
'Available Sizes'
```

'Available Sizes' is the current value for cell 'E1'

```
items.insert_cols(idx=5)

items['E1'] = 'Euro Price'

print('Column E header: ' + items['E1'].value)
print('Column F header: ' + items['F1'].value)
```

```
Column E header: Euro Price
Column F header: Available Sizes
```

After inserting a column at index 5 (column 'E'), and assigning 'E1' a value of 'Euro Price', note that 'Available Sizes' was shifted right to 'F1'

DELETING COLUMNS

- The openpyxl Package
- Navigating Workbooks
- Looping Through Cells
- Modifying Cells

You can **delete columns** from a worksheet as well

sheet.delete_cols(idx=index) deletes the column at the specified sheet and index

```
items['E1'].value
```

```
'Euro_Price'
```

'Euro_Price' is now the current value for cell 'E1'

```
items.delete_cols(idx=5)

print('Column E header: ' + items['E1'].value)
```

After deleting the column at index 5 (column 'E'), 'Available Sizes' was moved back to 'E1' from 'F1'

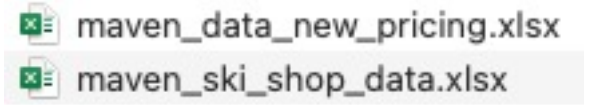
SAVING YOUR WORKBOOK

- The openpyxl Package
- Navigating Workbooks
- Looping Through Cells
- Modifying Cells

Use **workbook.save** to save your workbook in your local machine

```
wb.save('maven_data_new_pricing.xlsx')
```

	A	B	C	D	E	F
1	Product_ID	Product_Name	Price	Cost	Available Sizes	Euro Price
2	10001	Coffee	5.99	0.99	250mL	5.27
3	10002	Beanie	9.99	4.29	Child, Adult	8.79
4	10003	Gloves	19.99	7.99	Child, Adult	17.59
5	10004	Sweatshirt	24.99	10.59	XS, S, M, L , XL, XXL	21.99
6	10005	Helmet	99.99	49.99	Child, Adult	87.99
7	10006	Snow Pants	79.99	32.49	XS, S, M, L , XL, XXL	70.39
8	10007	Coat	119.99	54.55	S, M, L	105.59
9	10008	Ski Poles	99.99	69.99	S, M, L	87.99
10	10009	Ski Boots	199.99	89.99	5,6,7,8,9,10,11	175.99
11	10010	Skis	599.99	249.99	S, M, L	527.99
12	10011	Snowboard Boo	129.99	64.99	5,6,7,8,9,10,11	114.39
13	10012	Bindings	149.99	89.99	NA	131.99
14	10013	Snowboard	499.99	199.99	S, M, L, Powder	439.99



WARNING

This will replace any existing files with the same name in your folder

If you overwrite your input excel sheet, you could lose raw data that could be impossible to recover

Be careful!

BRINGING IT ALL TOGETHER

- The openpyxl Package
- Navigating Workbooks
- Looping Through Cells
- Modifying Cells

You now have a workflow that reads in an Excel workbook, creates a 'Euro Price' column based on a given exchange rate, and saves it back out

```
import openpyxl as xl
```

Import openpyxl

```
workbook = xl.load_workbook(filename='maven_ski_shop_data.xlsx')

items = workbook['Item_Info']
```

Read in the Excel workbook

Assign the sheet to a variable

```
items['F1'] = 'Euro Price'

exchange_rate = .88

for row in range(2, items.max_row + 1):
    items[f'F{row}'] = round(items[f'C{row}'].value * exchange_rate, 2)
```

Write the column header

Set the exchange rate

Loop through the column, convert, and write the values

```
workbook.save('maven_data_new_pricing.xlsx')
```

Save the Excel workbook

KEY TAKEAWAYS

The **openpyxl** package can manipulate Excel data using Python

- *You can read, modify, and save Excel workbooks without ever needing to open Excel*

Most of the **other Excel functionalities** are possible as well

- *You can create charts, apply conditional formatting, and write Excel formulas (just write them as strings in the cells where you want to place them!)*

This is just a sneak peak of Python's **capabilities to automate** external tasks

- *Beyond Excel, Python can manipulate flat files, read and write to databases, scrape web data, and more*