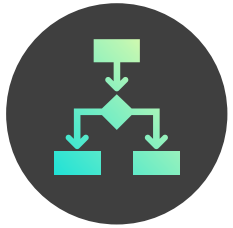
CONDITIONAL LOGIC

CONDITIONAL LOGIC



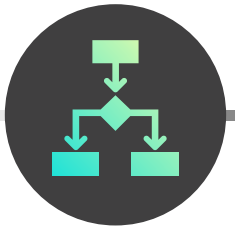
In this section we'll cover **conditional logic**, and write programs that make decisions based on given criteria using true or false expressions

TOPICS WE'LL COVER:

- The Boolean Data Type
- Boolean Operators
- Conditional Control Flow

GOALS FOR THIS SECTION:

- Understand the properties of the Boolean data type
- Learn to write true or false expressions using Boolean operators
- Learn to control the flow of the program using conditional logic statements



THE BOOLEAN DATA TYPE

The **Boolean data type** has two possible values: **True** & **False**

- **True** is equivalent to **1** in arithmetic operations
- **False** is equivalent to **0** in arithmetic operations

```
True * 1
```

```
1
```

```
False * 1
```

```
0
```

The **not** keyword inverts Boolean values:

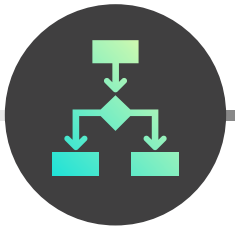
- **not True** is equivalent to **False**
- **not False** is equivalent to **True**

```
(not True) * 1
```

```
0
```

```
(not False) * 1
```

```
1
```



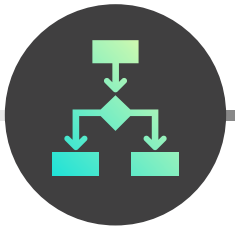
COMPARISON OPERATORS

The Boolean Data Type

Boolean Operators

Conditional Control Flow

		True	False
==	Equal	`5 == 5`	`5 == 3`
!=	Not Equal	`'Hello' != 'world!'`	`10 != 10`
<	Less Than	`5 < 6 < 7`	`21 < 12`
>	Greater Than	`10 > 5`	`10 > 5 > 7`
<=	Less Than or Equal	`5 <= 5`	`(5 + 5) <= 8`
>=	Greater Than or Equal	`11 >= 9 >= 9`	`len('I') >= len('am')`



MEMBERSHIP TESTS

The Boolean Data Type

Boolean Operators

Conditional Control Flow

Membership tests check if a value exists inside an iterable data type

- The keywords **in** and **not in** are used to conduct membership tests

```
message = 'I hope it snows tonight!'
'snow' in message
```

```
True
```

*'snow' is in the string assigned to **message**, so the membership test returns **True***

```
'rain' in message
```

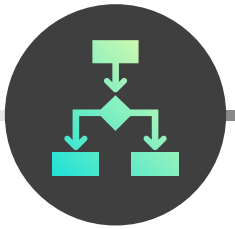
```
False
```

*'rain' is NOT in the string assigned to **message**, so the membership test returns **False***

```
message = 'I hope it snows tonight!'
'snow' not in message
```

```
False
```

*'snow' is in the string assigned to **message**, so the membership test checking if it is NOT in message returns **False***



BOOLEAN OPERATORS

- The Boolean Data Type
- Boolean Operators
- Conditional Control Flow

Boolean operators allow you to combine multiple comparison operators

- The **and** operator requires *all* statements to be true
- The **or** operator requires *one* statement to be true

One is True and the other False

```
5 > 4 and 7 > 8
```

```
False
```

```
5 > 4 or 7 > 8
```

```
True
```

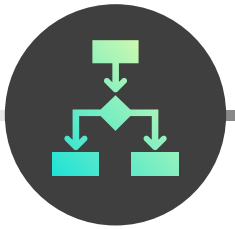
You can chain together any number of Boolean expressions

```
15 > 10 and 11 > 10 and 12 > 10
```

```
True
```

PRO TIP: If the first expression chained by **and** is **False**, or by **or** is **True**, the rest of the clauses will not be evaluated

Place simple clauses first to eliminate the need to run complex clauses, making your program more efficient



BOOLEAN OPERATORS

Boolean operators allow you to combine multiple comparison operators

- The **and** operator requires *all* statements to be true
- The **or** operator requires *one* statement to be true

Use parentheses to control the order of evaluation:

```
price = 105
item = 'skis'

(price <= 100 and item == 'ski poles') or (item == 'skis')
```

```
True
```

*Because **item == 'skis'** the **or** operator makes the expression True*

```
price <= 100 and (item == 'ski poles' or item == 'skis')
```

```
False
```

*Because **price <= 100** returns false the **and** operator makes the expression False*

ASSIGNMENT: BOOLEAN OPERATORS

NEW MESSAGE

February 6, 2022

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

Subject: **Inventory Logic**

We need develop logic that displays Boolean values based on our inventory levels (current inventory = 5):

1. Check if inventory is equal to 0
2. Check if inventory is greater than 5
3. If #2 is True, return a price of $99, 0.0 if not
4. Check if inventory is positive and price is less than $100
5. Check if the customer's name is 'Chris' and the product is 'super snowboard', or if the inventory is greater than 0

inventory_logic.ipynb

Reply | Forward

Results Preview

```
inventory_count = 5

[illegible]
```

```
False
```

```
[illegible]
```

```
False
```

```
[illegible]
```

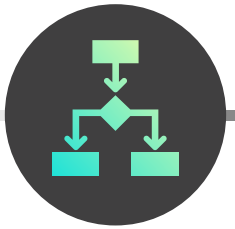
```
0.0
```

```
price = 99.99

[illegible]
```

```
True
```

```
customer_name = 'Barry'
inventory_count = 0
product = 'super_snowboard'

[illegible]
```

```
False
```



THE IF STATEMENT

The **if statement** runs lines of indented code when a given logical condition is met

- The Boolean Data Type
- Boolean Operators
- Conditional Control Flow

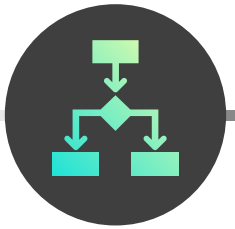
```
if condition:
```

Indicates an if statement

A logical expression that evaluates to True or False

Examples:

- *price > 100*
- *product == 'skis'*
- *inventory > 0 **and** inventory <= 10*



THE IF STATEMENT

The Boolean Data Type

Boolean Operators

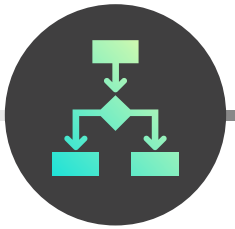
Conditional Control Flow

The **if statement** runs lines of indented code when a given logical condition is met

```
if condition:
    do this
```

Code to run if the logical expression returns True

(must be indented!)



THE IF STATEMENT

The **if statement** runs lines of indented code when a given logical condition is met

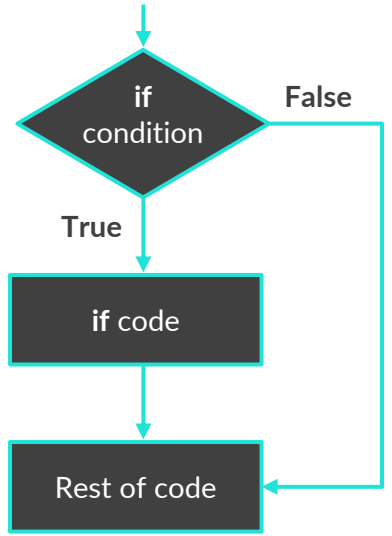
- The Boolean Data Type
- Boolean Operators
- Conditional Control Flow

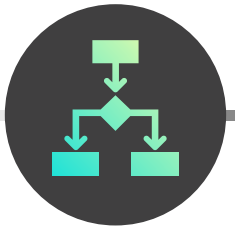
EXAMPLE *Determining experience level for a snowboard*

```
price = 999.99

if price > 500:
    print('This snowboard is designed for experienced users.')

print('This code will run whether or not the if statement is True.')
```

```
This product is for experienced users.
This code will run whether or not the if statement is True.
```

How does this code work?





CONTROL FLOW

- The Boolean Data Type
- Boolean Operators
- Conditional Control Flow

Control flow is a programming concept that allows you to choose which lines to execute, rather than simply running all the lines from top to bottom

There are three conditional statements that help achieve this: **if**, **else**, and **elif**

EXAMPLE *Determining experience level for a snowboard*

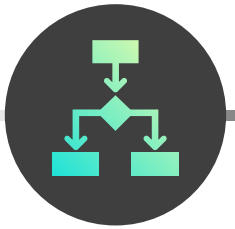
```
price = 499.99

if price > 500:
    print('This product is for experienced users.')

print('This code will run whether or not the if statement is True.')
```

```
This code will run whether or not the if statement is True.
```

*Python uses **indentation** to depart from a linear flow*

In this case, the first print statement only runs if price is greater than 500

Press tab, or use four spaces, to indent your code when writing if statements or you will receive an **IndentationError**



THE ELSE STATEMENT

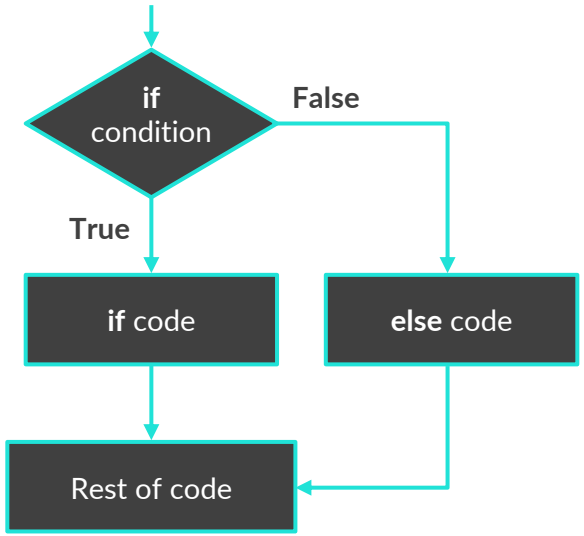
The **else statement** runs lines of indented code when the none of the logical conditions in an if or elif statements are met

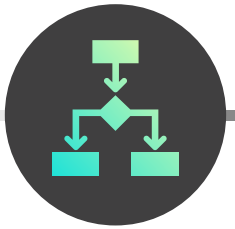
EXAMPLE *Determining experience level for a snowboard*

```
price = 499.99
price_threshold = 500

if price > price_threshold:
    print('This snowboard is for experienced users.')
else:
    print('This board is suitable for a beginner.')
```

```
This board is suitable for a beginner.
```

How does this code work?





THE ELIF STATEMENT

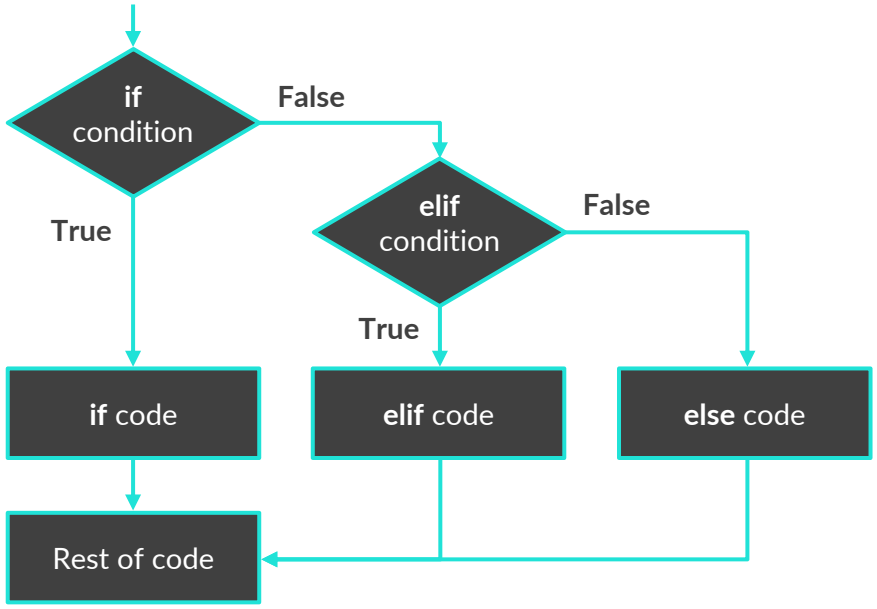
The **elif statement** lets you specify additional criteria to evaluate when the logical condition in an if statement is not met

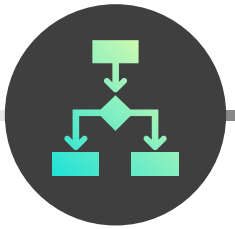
EXAMPLE *Determining experience level for a snowboard*

```
price = 499.99
expert_threshold = 500
intermediate_threshold = 300

if price > expert_threshold:
    print('This board is for experienced users.')
elif price > intermediate_threshold:
    print('This board is good for intermediate_users.')
else:
    print('This should be suitable for a beginner.')
```

```
This board is good for intermediate_users.
```

How does this code work?





THE ELIF STATEMENT

The Boolean Data Type

Boolean Operators

Conditional Control Flow

Any number of **elif statements** can be used between the if and else statements

As more logical statements are added, it's important to be careful with their order

Will the elifs below ever run?

```
price = 1499.99
luxury_threshold = 1000
expert_threshold = 500
intermediate_threshold = 300

if price > intermediate_threshold:
    print('This board is good for intermediate users')
elif price > expert_threshold:
    print('This board is for experienced users')
elif price > luxury_threshold:
    print("The gold doesn't improve the ride but it looks great!")
else:
    print('This should be suitable for a beginner.')
```

```
This board is good for intermediate users
```

***No**, because any value that is true for them will also be true for the if statement above them*

Either put the most restrictive conditions first, or be more explicit with your logic

ASSIGNMENT: CONTROL FLOW

NEW MESSAGE

February 6, 2022

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

Subject: **Inventory Query**

Hi there, great work on the last assignment!

Time to take it a step further:

- If inventory is 0 or less, print 'OUT OF STOCK'
- If inventory is between 1-5, print 'Low Stock'
- If inventory is over 5, print 'In Stock'

Make sure to test your logic with several values.

Can't wait to see your work!

inventory_query.ipynb

Reply | Forward

Results Preview

```
inventory = 3
```

```
Low Stock
```

NESTED IF STATEMENTS

The Boolean Data Type

Boolean Operators

Conditional Control Flow

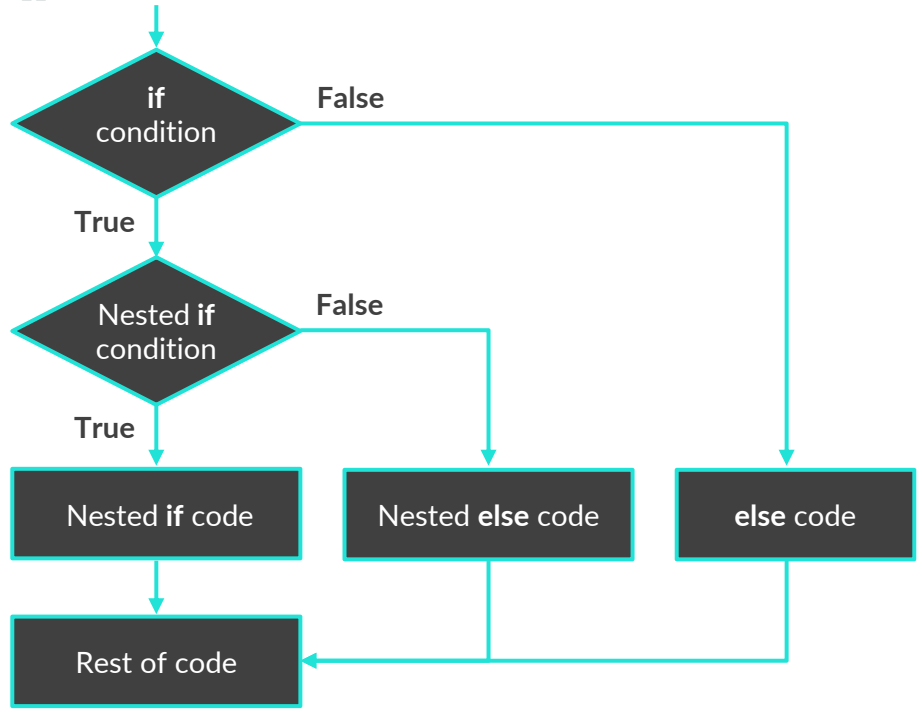
Nested if statements let you specify additional criteria to evaluate after a logical condition in an if statement is met

EXAMPLE | *Trying to purchase a product*

```
price = 499.99
budget = 500
inventory = 0


if budget > price:
    if inventory > 0:
        print('You can afford this and we have it in stock!')
    else:  # equivalent to if inventory <= 0
        print("You can afford this but it's out of stock.")
else:
    print(f'Unfortunately, this board costs more than ${budget}.')
```

```
You can afford this but it's out of stock.
```

How does this code work?



ASSIGNMENT: NESTED IF STATEMENTS

NEW MESSAGE

February 6, 2022

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

Subject: **Inventory Query Edit**

Hi there, we need to make one final change to our inventory logic for Chris, our owner:

- If inventory is 0 or less, and if the customer's name is 'Chris', print 'You can have the display model, sir'
- Otherwise, print 'Out of stock'

The rest of the logic should stay the same.

Thanks again!

inventory_query2.ipynb

Reply Forward

Results Preview

```
inventory = 0
customer_name = 'Chris'
```

```
You can have the display model, sir
```

KEY TAKEAWAYS

The Boolean data type is the foundation of conditional logic

- *The only two values for the Boolean data type are True and False, which are equivalent to 1 and 0, respectively, when performing arithmetic operations*

You can write logical statements using **comparison operators, membership tests** and **Boolean operators**

- *Comparison operators return a Boolean value and include equal, not equal, greater than, and less than*
- *Membership tests determine if a specified value is in an iterable, and return a Boolean value*
- *Boolean operators (and, or) let you evaluate multiple logical statements as a single condition*

Conditional control flow lets you determine which lines of code to execute

- *Conditional statements include IF, ELIF, and ELSE, which can be nested for multiple layers of logic*

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**

ASSIGNMENT: LIST OPERATIONS

NEW MESSAGE

February 7, 2022

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

Subject: **Customer Lists**

Hi there!

Can you create a list from the customers that made purchases on Black Friday (attached) and:

- Return 99.99 if C00009 made a purchase; otherwise return 0.0
- Create separate lists for the 5th and 6th customers, every third customer, and the last 2 customers, we're conducting market research.

Thanks again!

black_friday_customers.ipynb

Reply | Forward

Results Preview

```
customer_list =

print(customer_list)
```

```
['C00001', 'C00003', 'C00004', 'C00005', 'C00006', 'C00007', 'C00018',
'C00010', 'C00013', 'C00014', 'C00015', 'C00016', 'C00029']
```

```
* 99.99
```

```
0.0
```

```
fifth_sixth =

fifth_sixth
```

```
['C00006', 'C00007']
```

```
every_third =

every_third
```

```
['C00001', 'C00005', 'C00018', 'C00014', 'C00029']
```

```
last_two =
last_two
```

```
['C00016', 'C00020']
```

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
```

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

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

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*)**

ASSIGNMENT: ADDING LIST ELEMENTS

NEW MESSAGE

February 7, 2022

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

Subject: **Adding Customers**

Hi again!

We just found a receipt for customer C00009 in the warehouse, can you add them to the customer list?

While you're at it, we decided to extend the sale through Saturday since things were so crazy.

Can you add the customers in saturday_list to customer_list?

Thank you!

updated_customer_lists.ipynb

Reply | Forward

Results Preview

Adding C00009:

```
[redacted]

print(customer_list)
```

```
['C00001', 'C00003', 'C00004', 'C00005', 'C00006', 'C00007', 'C00008',
'C00010', 'C00013', 'C00014', 'C00015', 'C00016', 'C00020', 'C00009']
```

Adding saturday_list:

```
[redacted]

print(customer_list)
```

```
['C00001', 'C00003', 'C00004', 'C00005', 'C00006', 'C00007', 'C00008',
'C00010', 'C00013', 'C00014', 'C00015', 'C00016', 'C00020', 'C00009',
'C00004', 'C00017', 'C00019', 'C00002', 'C00008', 'C00021', 'C00022']
```

ASSIGNMENT: REMOVING LIST ELEMENTS

NEW MESSAGE

February 7, 2022

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

Subject: **Change of plan!**

Hi there!

Sorry to do this to you, but we decided to keep the Friday and Saturday sales separate for now. Can you remove Saturday's customers from the customer list?

Also, C00004 is a friend of the owner and should not be counted in the Black Friday sales, so please remove them from the list.

Thanks again!

updated_updated_customer_lists.ipynb

Reply | Forward

Results Preview

Removing Saturday's customers:

```
print(customer_list)
```

```
['C00001', 'C00003', 'C00004', 'C00005', 'C00006', 'C00007', 'C00008',
'C00010', 'C00013', 'C00014', 'C00015', 'C00016', 'C00020', 'C00009']
```

Removing C00004:

```
print(customer_list)
```

```
['C00001', 'C00003', 'C00005', 'C00006', 'C00007', 'C00008', 'C00010',
'C00013', 'C00014', 'C00015', 'C00016', 'C00020', 'C00009']
```

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
```

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

```
sum(transactions)
```

```
1483.88
```

min(list_name) *Returns the smallest element in the list*

```
min(transactions)
```

```
2.07
```

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"

ASSIGNMENT: LIST FUNCTIONS & METHODS

NEW MESSAGE

February 7, 2022

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

Subject: **Customer C00010 Transactions**

I've attached a file with two lists: customer_IDs and subtotals. The index for customer ID matches the index for the subtotal on each of their transactions. *(customer_ids[0] made the transaction subtotals[0])*

Can you look into customer C00010's behavior?

1. Calculate the average value of a transaction
2. Report how many transactions C00010 made
3. Look up the value of their first transaction
4. Compare it to the average value
5. Print a list of sorted IDs *(don't change the list!)*

customer_transactions.ipynb

Reply Forward

Results Preview

Average transaction value:

```
323.3877777777776
```

C00010's transactions:

```
2
```

First transaction index:

```
5
```

First transaction value:

```
599.99
```

Sorted ID list:

```
['C00001', 'C00001', 'C00001', 'C00002', 'C00003', 'C00004',
'C00004', 'C00005', 'C00006', 'C00006', 'C00007', 'C00008',
'C00008', 'C00010', 'C00010', 'C00013', 'C00014', 'C00015',
'C00016', 'C00016', 'C00017', 'C00018', 'C00018', 'C00019',
'C00020', 'C00021', 'C00022']
```

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
```

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

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

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

'h' appears once in the third list

```
1
```

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

VARIABLE ASSIGNMENT

Copying a list via **variable assignment** creates a "view" of the list

- Both variables point to the same object in memory
- Changing an element in one list will result in a change to the other

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

# copy list using variable assignment
list_of_lists2 = list_of_lists

# change element in original list
list_of_lists[1] = ['x', 'y', 'z']

# The change will be reflected in the copy made by assignment
list_of_lists2
```

```
[['a', 'b', 'c'], ['x', 'y', 'z'], ['g', 'h', 'i']]
```

*The change made to **list_of_lists** is reflected in **list_of_lists2***

COPY

Copying a list with the **.copy()** method creates a copy of the list

- 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

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

# use the copy method to create a copy
list_of_lists2 = list_of_lists.copy()

# replace the entire nested listed at index 0
list_of_lists[0] = ['x', 'y', 'z']

list_of_lists2
```

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

Since the entire nested list at index 0 was replaced, the change is NOT reflected in the copy

```
# change the second element of the first nested list
list_of_lists[0][1] = 'Oh no!'

list_of_lists2
```

```
[['a', 'Oh no!', 'c'], ['d', 'e', 'f'], ['g', 'h', 'i']]
```

Since a single element (index of 1) within the nested list was modified, the change IS reflected in the copy

DEEPCOPY

Copying a list with the **deepcopy()** function creates a separate copy of the list

- Any changes made to one of the lists will NOT impact the other

This function is not part of base Python, so the copy library must be imported

```
from copy import deepcopy

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

list_of_lists2 = deepcopy(list_of_lists)

list_of_lists[0][1] = 'Oh no!'

list_of_lists2
```

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

The change is NOT reflected in the copy, even though a single element (index of 1) within the nested list was modified

ASSIGNMENT: NESTED LISTS & COPYING

NEW MESSAGE

February 7, 2022

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

Subject: **Customer C00010 Transactions**

I've attached a file with five lists, each for a specific customer's orders. We're interested in these customers because they have multiple transactions.

Can you help:

- Create a list with these 5 lists as elements
- Print the second and third orders for the customer that made three total orders
- Create a copy of this list and change the transaction values for C00004 (friend of the owner) to 0.0 – without changing the original!

multi_order_customers.ipynb

Reply | Forward

Results Preview

```
orders_c00001 = [1799.94, 29.98, 99.99]
orders_c00004 = [15.98, 119.99]
orders_c00006 = [24.99, 24.99]
orders_c00008 = [649.99, 99.99]
orders_c00010 = [599.99, 399.97]
```

```
vip_list = [redacted]

vip_list
```

```
[[1799.94, 29.98, 99.99],
 [15.98, 119.99],
 [24.99, 24.99],
 [649.99, 99.99],
 [599.99, 399.97]]
```

```
vip_list [redacted]
```

```
[29.98, 99.99]
```

```
revenue_adjusted_list = [redacted]
```

```
[redacted]

revenue_adjusted_list
```

```
[[1799.94, 29.98, 99.99],
 [0.0, 0.0],
 [24.99, 24.99],
 [649.99, 99.99],
 [599.99, 399.97]]
```

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')
```

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

```
item_tuple[3]
```

```
'Goggles'
```

```
item_tuple[:3]
```

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

```
len(item_tuple)
```

```
5
```

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

ASSIGNMENT: TUPLES

NEW MESSAGE

February 7, 2022

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

Subject: **Sales Tax**

Hi there, we need to apply sales tax to each of customer C00001's transactions.

Can you grab these from multi_order_list, create a tuple, and unpack the tuple into three variables: transaction1, transaction2, and transaction3?

Then we just need to multiply them by 0.08 to calculate the sales tax of 8%, and round to the nearest cent.

Thanks in advance!

sales_tax.ipynb

Reply | Forward

Results Preview

```
144.0
2.4
8.0
```

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]
```

```
print(tuple(range(len('Hello'))))
```

```
(0, 1, 2, 3, 4)
```

You can retrieve them by converting to a list or tuple

ASSIGNMENT: RANGES

NEW MESSAGE

February 7, 2022

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

Subject: **Customer Giveaway**

We are running a special email campaign to give products away to lucky customers who make a purchase.

We need the following integers generated:

- Even numbers in our first 10 customers (starting with our 2nd customer). They will receive a coffee mug.
- Odd numbers in our first 10 customers (starting with our 1st customer). They will get keychains.
- Every seventh customer in our first 100 customers (starting with our 7th customer). They get beanies!

customer_promo_generator.ipynb

Reply | Forward

Results Preview

Even Customers:

```
print
```

```
[2, 4, 6, 8, 10]
```

Odd Customers:

```
print
```

```
[1, 3, 5, 7, 9]
```

Every 7th Customer:

```
print
```

```
[7, 14, 21, 28, 35, 42, 49, 56, 63, 70, 77, 84, 91, 98]
```

KEY TAKEAWAYS

Lists and tuples are sequence data types capable of storing many values

- *Lists allow you to add, remove, and change elements, while tuples do not*

Use **indexing** and **slicing** to access & modify list elements

- *Make sure you are comfortable with this syntax, as you will use it frequently in data analysis libraries as well*

Built-in **functions** and **methods** make working with sequences easier

- *All the list functions covered work with tuples as well, and tuples have many identical methods, like index() and count(). Take a minute to skim through the available methods in the Python documentation.*

Ranges help create a **sequence of integers** efficiently

- *These will become particularly powerful when combined with loops*

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

- Loop Basics
- For Loops
- While Loops
- Nested Loops
- Loop Control

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

- Loop Basics
- For Loops
- While Loops
- Nested Loops
- Loop Control

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

- Loop Basics
- For Loops
- While Loops
- Nested Loops
- Loop Control

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

Examples:

- *Price*
- *Product*
- *Customer*

Iterable object to loop through

Examples:

- *List*
- *Tuple*
- *String*
- *Dictionary*
- *Set*

FOR LOOPS

- Loop Basics
- For Loops
- While Loops
- Nested Loops
- Loop Control

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]*
-
-
- *item = iterable[len(iterable)-1]*

FOR LOOPS

- Loop Basics
- For Loops
- While Loops
- Nested Loops
- Loop Control

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

- Loop Basics
- For Loops
- While Loops
- Nested Loops
- Loop Control

EXAMPLE | *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

- Loop Basics
- For Loops
- While Loops
- Nested Loops
- Loop Control

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

- Loop Basics
- For Loops
- While Loops
- Nested Loops
- Loop Control

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)
```

The index is selecting the elements in the list

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

*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

- Loop Basics
- For Loops
- While Loops
- Nested Loops
- Loop Control

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] = Snowboard*
- *euro_list[i] = 5.27*

PRO TIP: ENUMERATE

- Loop Basics
- For Loops
- While Loops
- Nested Loops
- Loop Control

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)
```

```
0 5.27
1 8.79
2 17.59
3 21.99
4 87.99
```

Use the index for multiple lists!

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.

```
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
```

*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*

ASSIGNMENT: FOR LOOPS

NEW MESSAGE

February 8, 2022

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

Subject: **Sales Tax**

Good morning,

Our billing system failed to apply sales tax to our transactions! Can you add a sales tax of 8% to each of the subtotals?

Once you've calculated tax, calculate the total by adding the tax to the subtotal, and create lists to store both the tax amounts and the totals.

Thanks in advance!

taxable_transactions.ipynb

Reply | Forward

Results Preview

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

```
taxes = []
totals = []

for subtotal in subtotals:
    tax = round(subtotal * .08, 2)
    total = round(subtotal + tax, 2)
    taxes.append(tax)
    totals.append(total)

print(taxes)
print(totals)
```

```
[1.28, 72.0, 64.0, 9.44, 0.48, 48.0, 2.0, 144.0, 8.0]
[17.26, 971.97, 863.97, 127.4, 6.47, 647.99, 26.99, 1943.94, 107.99]
```

*1.28 = round(subtotal[0] * .08, 2)*

17.26 = round(1.28 + subtotal[0], 2)

ASSIGNMENT: ENUMERATE

NEW MESSAGE

February 8, 2022

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

Subject: **Sales Tax By State**

Hi again!

I forgot to mention we need to apply different tax rates to different locations. The list attached has the location of each transaction. Our Sun Valley store is taxed at 8%, Stowe at 6%, and Mammoth at 7.75%.

Can you adjust your tax calculator code to apply the correct tax to each location?

Thanks!

taxable_transactions2.ipynb

Reply | Forward

Results Preview

```
location = ['Sun Valley', 'Stowe', 'Mammoth',
            'Stowe', 'Sun Valley', 'Mammoth',
            'Mammoth', 'Mammoth', 'Sun Valley']
```

```
taxes = []
totals = []

for i, subtotal in enumerate(subtotals):
    [redacted]

print(taxes)
print(totals)
```

```
[1.28, 54.0, 62.0, 7.08, 0.48, 46.5, 1.94, 139.5, 8.0]
[17.26, 953.97, 861.97, 125.04, 6.47, 646.49, 26.93, 1939.44, 107.99]
```

WHILE LOOPS

- Loop Basics
- For Loops
- While Loops
- Nested Loops
- Loop Control

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

- Loop Basics
- For Loops
- While Loops
- Nested Loops
- Loop Control

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)
```

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

```
counter = counter + 1
```

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*

WHILE LOOPS

- Loop Basics
- For Loops
- While Loops
- Nested Loops
- Loop Control

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

- Loop Basics
- For Loops
- While Loops
- Nested Loops
- Loop Control

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

- Loop Basics
- For Loops
- While Loops
- Nested Loops
- Loop Control

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

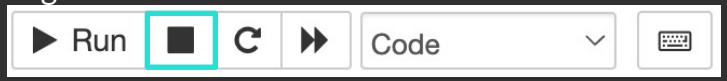
```
stock_portfolio = 800000
year_counter = 0

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)}')
```

*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*

```
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

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



ASSIGNMENT: WHILE LOOPS

NEW MESSAGE

February 9, 2022

From: **Lucy Lift** *(CEO)*

Subject: **Inventory Projections**

Good morning,

Our investors are considering loaning us money to purchase more skis.

Can you advise how many months of stock we have left at the current monthly sales volume?

I would like to view the stock level at the end of each month as well.

Thanks!

inventory_logic.ipynb

Reply | Forward

Results Preview

```
current_inventory = 686
monthly_sales = 84
month = 0

while current_inventory > 0:
    current_inventory -= monthly_sales
    month += 1
    print('At the end of month {month}, we have {current_inventory} pairs of skis')
```

```
At the end of month 1, we have 602 pairs of skis
At the end of month 2, we have 518 pairs of skis
At the end of month 3, we have 434 pairs of skis
At the end of month 4, we have 350 pairs of skis
At the end of month 5, we have 266 pairs of skis
At the end of month 6, we have 182 pairs of skis
At the end of month 7, we have 98 pairs of skis
At the end of month 8, we have 14 pairs of skis
At the end of month 9, we have -70 pairs of skis
```

NESTED LOOPS

- Loop Basics
- For Loops
- While Loops
- Nested Loops
- Loop Control

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

- Loop Basics
- For Loops
- While Loops
- Nested Loops
- Loop Control

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)

0	**Snowboard**
1	Boots

size_list (inner loop)

0	**small**
1	medium
2	large

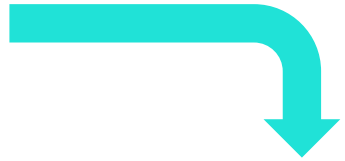
NESTED LOOPS

- Loop Basics
- For Loops
- While Loops
- Nested Loops
- Loop Control

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)

0	**Snowboard**
1	Boots

size_list (inner loop)

0	small
1	**medium**
2	large

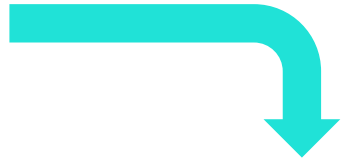
NESTED LOOPS

- Loop Basics
- For Loops
- While Loops
- Nested Loops
- Loop Control

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)

0	**Snowboard**
1	Boots

size_list (inner loop)

0	small
1	medium
2	**large**

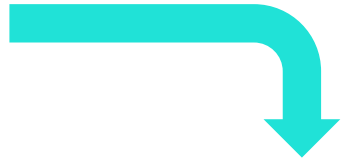
NESTED LOOPS

- Loop Basics
- For Loops
- While Loops
- Nested Loops
- Loop Control

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*)

0	Snowboard
1	**Boots**

size_list (*inner loop*)

0	**small**
1	medium
2	large

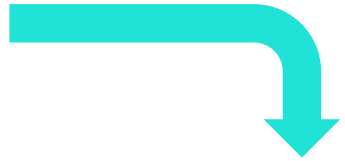
NESTED LOOPS

- Loop Basics
- For Loops
- While Loops
- Nested Loops
- Loop Control

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*)

```
0   Snowboard
1   Boots
```

size_list (*inner loop*)

```
0   small
1   medium
2   large
```

NESTED LOOPS

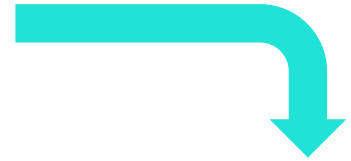
- Loop Basics
- For Loops
- While Loops
- Nested Loops
- Loop Control

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***



item_list (outer loop)

```
0   Snowboard
1   Boots
```

size_list (inner loop)

```
0   small
1   medium
2   large
```

ASSIGNMENT: NESTED LOOPS

NEW MESSAGE

February 9, 2022

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

Subject: **Multi-Order Discount**

Good morning,

I've attached a nested list containing the subtotals for customers that have made multiple orders.

We need to apply a 10% discount for each of their transactions and return the new totals in a list with the same structure as the original.

Thanks!

multi_order_discounts.ipynb

Reply | Forward

Results Preview

```
multi_order_customers
```

```
[[1799.94, 29.98, 99.99],
 [15.98, 119.99],
 [24.99, 24.99],
 [649.99, 99.99],
 [599.99, 399.97]]
```

```
discounted_prices = []

for customer in multi_order_customers:
    customer_discounts = []
    for transaction in customer:
        customer_discounts.append(transaction * .9)
    discounted_prices.append(customer_discounts)

discounted_prices
```

```
[[1619.95, 26.98, 89.99],
 [14.38, 107.99],
 [22.49, 22.49],
 [584.99, 89.99],
 [539.99, 359.97]]
```

LOOP CONTROL

- Loop Basics
- For Loops
- While Loops
- Nested Loops
- Loop Control

Loop control statements help refine loop behavior and handle potential errors

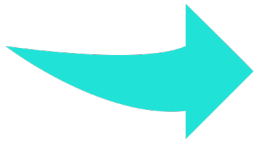
- These are used to change the flow of loop execution based on certain conditions

Break

Stops the loop before completion

Good for avoiding infinite loops & exiting loops early

Continue



Skips to the next iteration in the loop

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

Pass

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

Loop Basics

For Loops

While Loops

Nested Loops

Loop Control

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

- Loop Basics
- For Loops
- While Loops
- Nested Loops
- Loop Control

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
```

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*

```
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.
```

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

CONTINUE

- Loop Basics
- For Loops
- While Loops
- Nested Loops
- Loop Control

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

- Loop Basics
- For Loops
- While Loops
- Nested Loops
- Loop Control

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:
    try:
        affordable_quantity = 50//price # My budget is 50 dollars
        print(f"I can buy {affordable_quantity} of these.")
    except:
        print("The price seems to be missing.")
```

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

```
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

- Loop Basics
- For Loops
- While Loops
- Nested Loops
- Loop Control

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

- Loop Basics
- For Loops
- While Loops
- Nested Loops
- Loop Control

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")
```

*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*

```
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.
```

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

ASSIGNMENT: LOOP CONTROL

NEW MESSAGE

February 9, 2022

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

Subject: **Affordability Calculator**

Good morning,

We've made good progress on the item affordability calculator, which really helps our customers buy with confidence. We just need these tweaks to the logic:

- Skip processing for any 'None' elements
- Make sure prices stored as strings get processed (we'll fix the database later)
- Change 50 to a variable 'budget' so we can change it based on customer input
- Clean up general 'except' statement

Improved_affordability_calculator.ipynb

Reply | Forward

Results Preview

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

# Create Budget Variable
```

```
I can buy 8.0 of these.
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.
string data detected
I can buy 0.0 of these.
I can buy 0.0 of these.
```

KEY TAKEAWAYS

For loops run a **predetermined** number of times

- *Analysts use for loops most often, as we usually work with datasets that have a known length*

While loops run until a given **logical condition** is no longer met

- *The number of iterations is often unknown beforehand, which means they carry a risk of entering an infinite loop – always double check your logic or create a backup stopping condition!*

Loop control statements help **refine logic** and **handle potential errors**

- *These are key in "fool proofing" loops to avoid infinite loops, set placeholders for future logic, skip iterations, and resolve errors*