WELLCOME

PYTHON FOR DATA SCIENCE AND AI

YA MANON

LOOPS

You can have data without information but you cannot have information without data.

-Daniel Keys Maran

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:

- LOOPs basics
- While Loops
- Loop control
- Foor Loops
- Nested Loops

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

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

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  initerable:
    do this
```

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

Run order:

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

FOR LOOPS

- 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**

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 iterableobjects

```
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*

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 loopsrun 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

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

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                         Trace
```

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

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)*

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
```

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 executionbased 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 6thtransaction*

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

- 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:
        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

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*