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

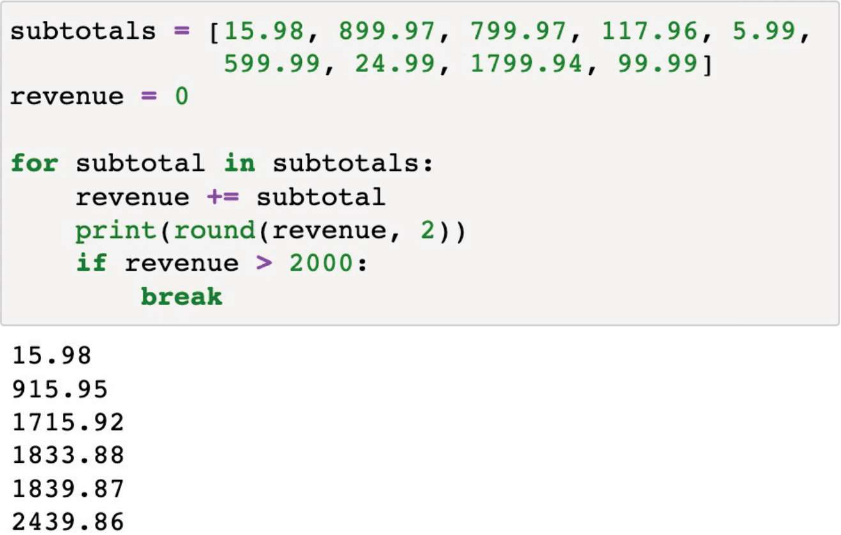
While Loops

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
```

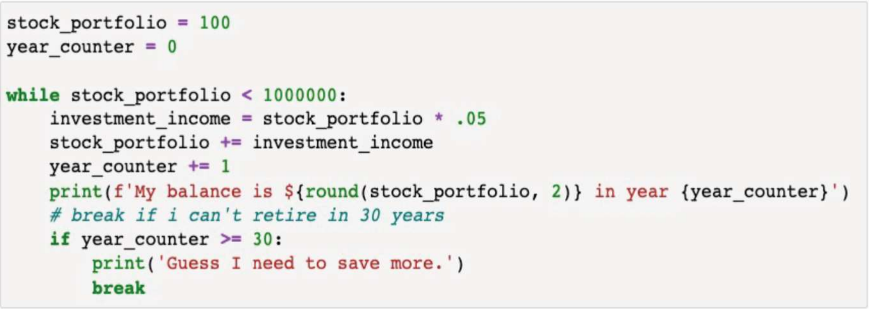
*The for loop here would normally run the length of the entire subtotals list (9 iterations), but the **break** statement triggers once the revenue is greater than 2,000 after the 6th transaction*

BREAK

Loop Basics
For Loops
While Loops
Nested Loops
Loop Control

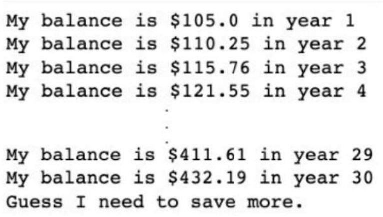
Triggering a **break** statement will exit the loop that it lives in

- 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



```python
stock_portfolio = 100
year_counter = 0

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



```
My balance is $105.0 in year 1
My balance is $110.25 in year 2
My balance is $115.76 in year 3
My balance is $121.55 in year 4
                 .
                 .
My balance is $411.61 in year 29
My balance is $432.19 in year 30
Guess I need to save more.
```

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

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

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

CONTINUE

Loop Basics

For Loops

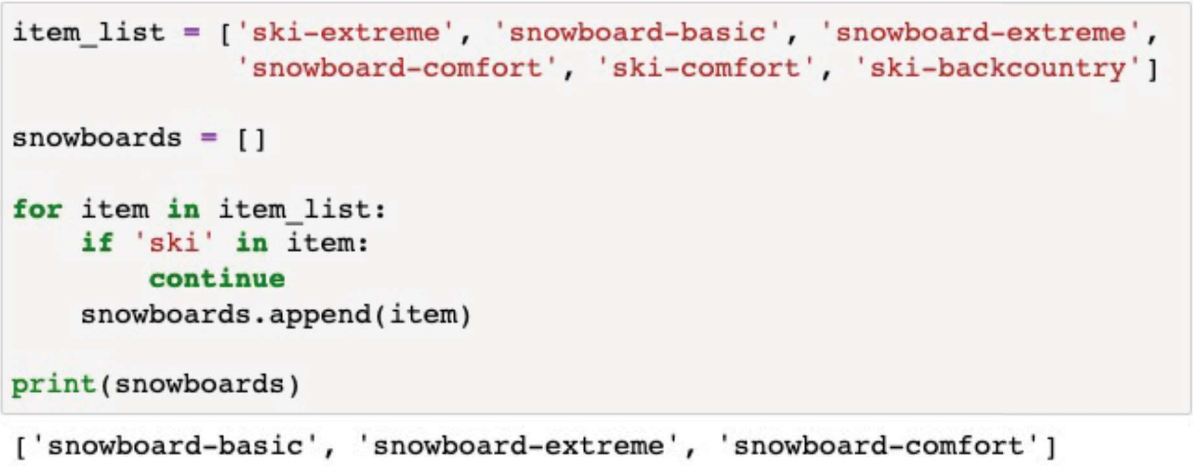
While Loops

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*

Loop Basics

For Loops

While Loops

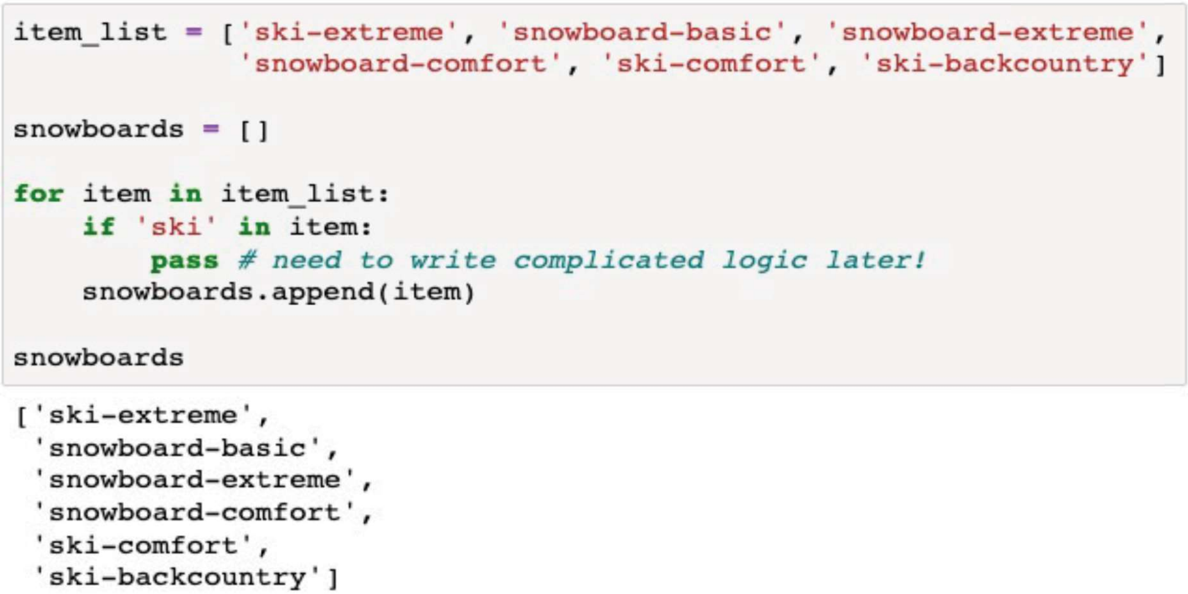
Nested Loops

Loop Control

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

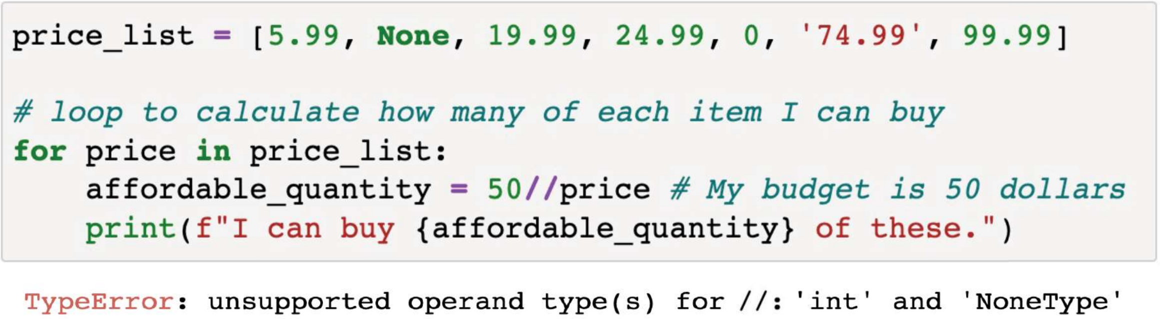
While Loops

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.")
```

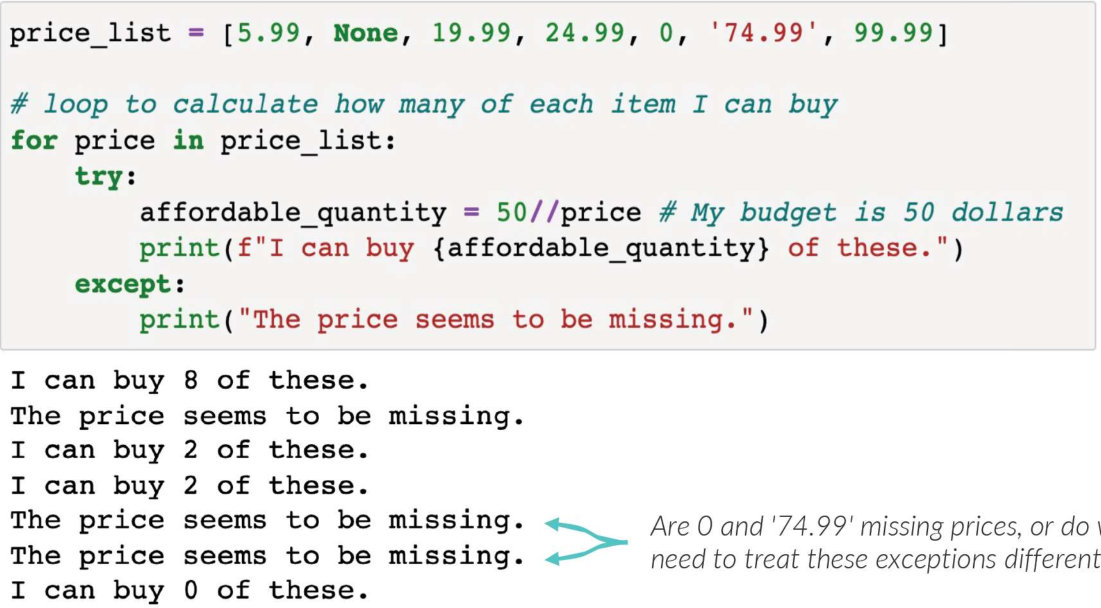
```
TypeError: unsupported operand type(s) for //: 'int' and 'NoneType'
```

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.")
```

```
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.
```

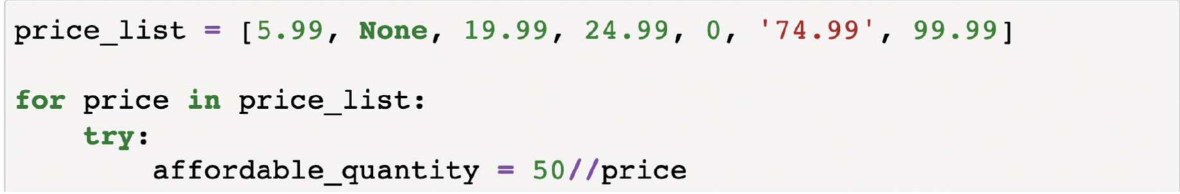
*Placing the code in a **try** statement handles the errors via the **except** statement without stopping the loop*

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

TRY, EXCEPT

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

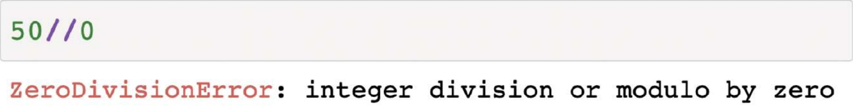
- **Try:** indicates the first block of code to run (which could result in an error)
- **Except:** indicates an optional block of code to run in case of an error in the try block



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

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

The 0 price in the price_list returns a ***ZeroDivisionError***



```
50//0
```

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

TRY, EXCEPT

Loop Basics

For Loops

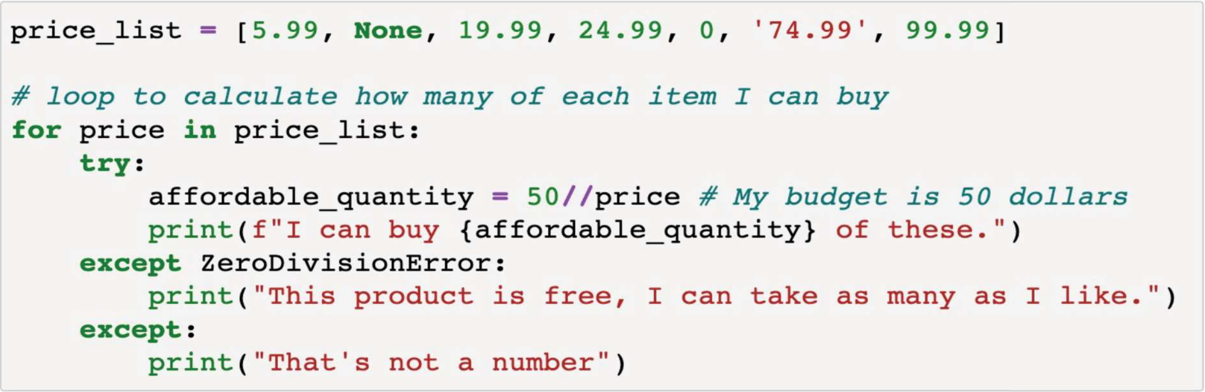
While Loops

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")
```

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

*If anything in the **try** block returns a ZeroDivisionError, the first **except** statement will run*

*The second **except** statement will run on any other error types*

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