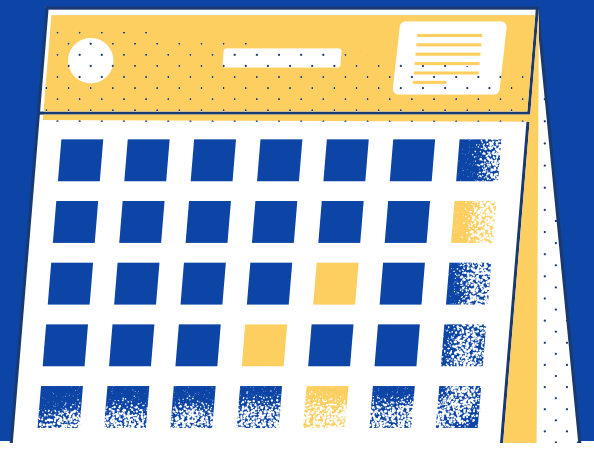
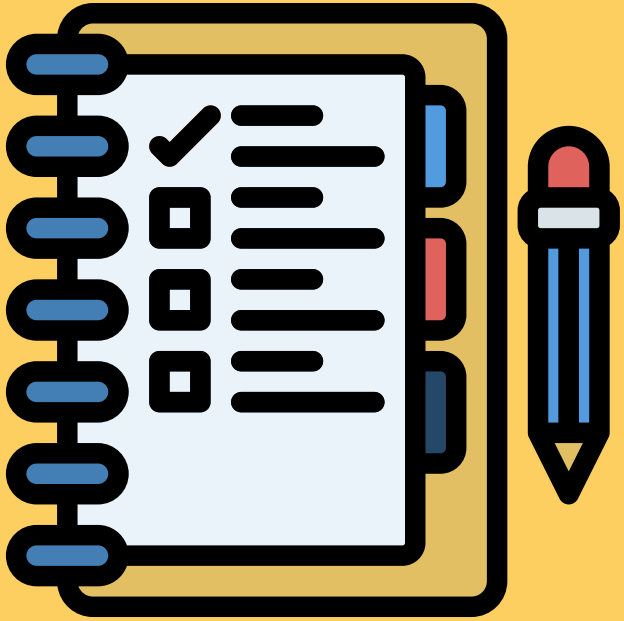
Data Fundamentals

Control Flow and Functions in Python

GOALS



- **Master Control Flow**: Learn key concepts including conditional statements, Boolean expressions, and control flow statements like 'break' and 'continue'.
- **Utilize Built-in Functions & Comprehensions**: Understand the use of Python's built-in functions like 'zip', 'enumerate', and 'range', as well as the concept of list comprehensions.
- **Deep Dive into Functions**: Grasp the importance of functions, variable scope, and the application of lambda expressions.
- **Advanced Concepts**: Uncover advanced Python concepts, specifically focusing on functional programming methods such as 'filter' and 'map'.

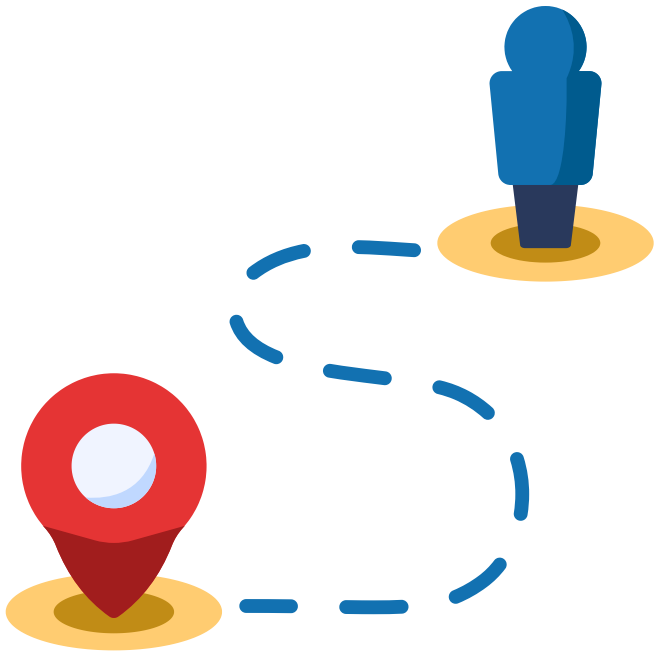


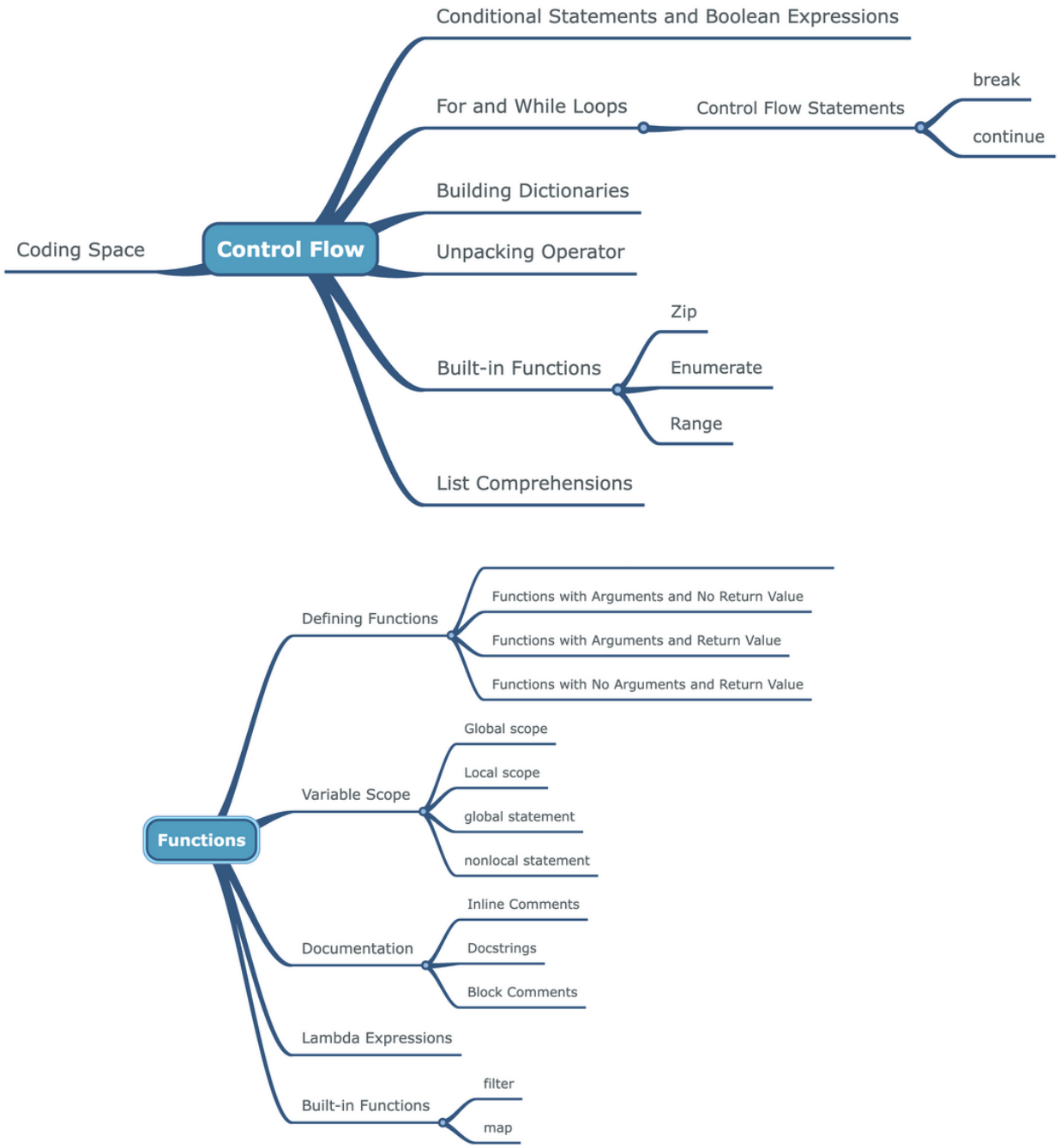
AGENDA

- Welcome
- Roadmap & Quick Review
- Introduction and Goals
- Python Data Types
- Understanding Control Flow
- Exploring Some Built-in Functions
- Grasping Functions and Their Usage
- Diving into Lambda Expressions
- Common Built-In Functions for Data Structures
- Q&A

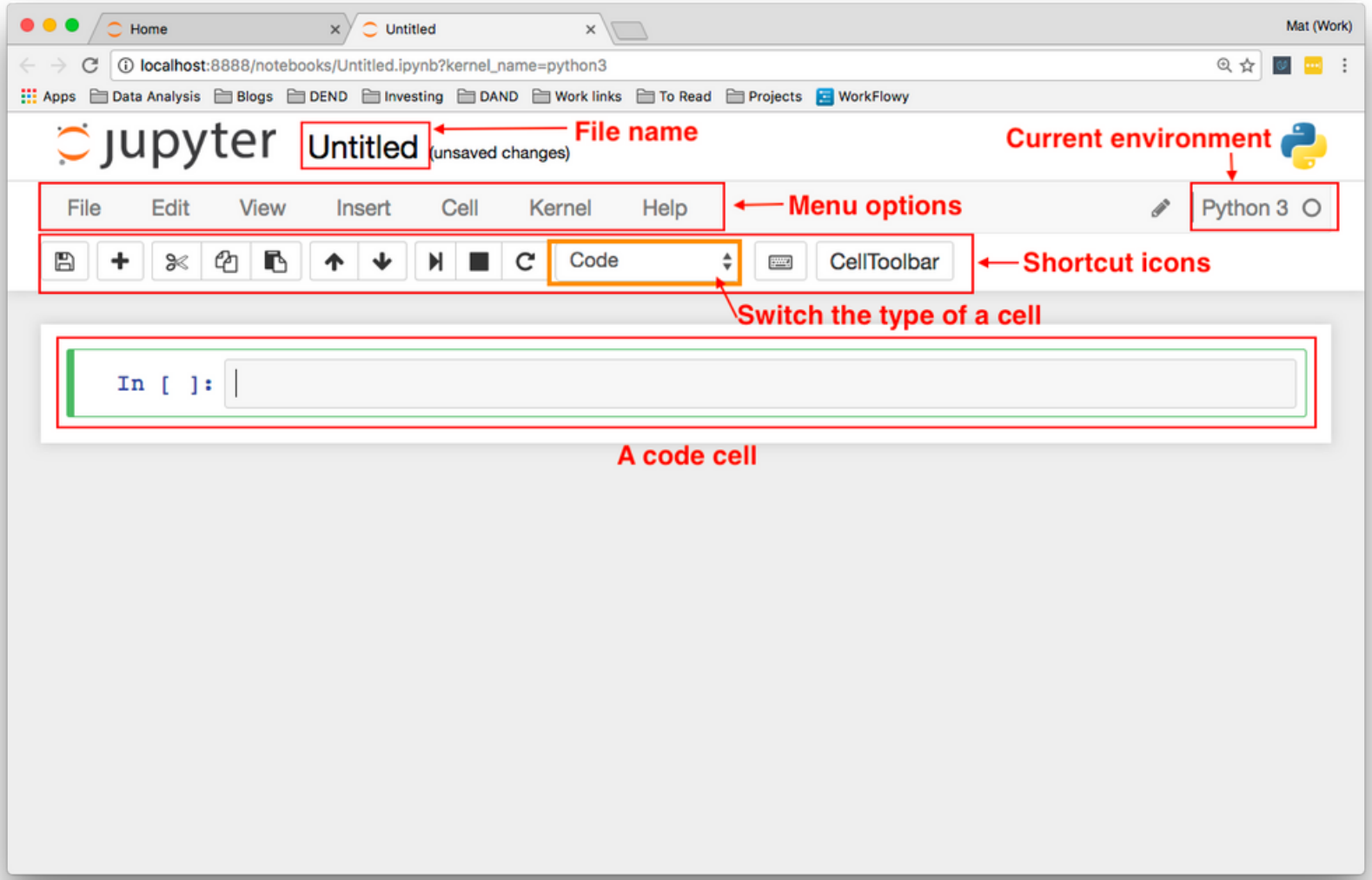
Behind every data point, there's a story waiting to be told.

ROADMAP




Coding Space
Control Flow
Conditional Statements and Boolean Expressions
For and While Loops
Control Flow Statements
break
continue
Building Dictionaries
Unpacking Operator
Built-in Functions
Zip
Enumerate
Range
List Comprehensions
Functions
Defining Functions
Functions with Arguments and No Return Value
Functions with Arguments and Return Value
Functions with No Arguments and Return Value
Variable Scope
Global scope
Local scope
global statement
nonlocal statement
Documentation
Inline Comments
Docstrings
Block Comments
Lambda Expressions
Built-in Functions
filter
map

CODING SPACE



WHAT IS CONTROL FLOW?

Control flow in Python allows you to make decisions and repeat actions based on conditions.

CONDITIONAL STATEMENTS

Conditional statements let us run different parts of code depending on certain rules. These rules are set by using true or false conditions, also known as Boolean expressions.

```
if condition1:
    # code block to execute if condition1 is True
elif condition2:
    # code block to execute if condition2 is True
else:
    # code block to execute if none of the above conditions are True
```

```
x = 10
if x > 0:
    print("x is positive.")
elif x < 0:
    print("x is negative.")
else:
    print("x is zero.")
```

LOOPS

Loops in Python allow you to execute a block of code repeatedly as long as a condition is met. There are two types of loops in Python: for loop and while loop.

FOR LOOP

The for loop iterates over a sequence (e.g., list, tuple, string, etc.) and executes the block of code for each element in the sequence.

```
for item in sequence:
    # code block to execute for each item in the sequence
```

```
fruits = ["apple", "banana", "cherry"]
for fruit in fruits:
    print(fruit)
```

WHILE LOOP

The while loop continues to execute the block of code as long as the given condition is True.

```
while condition:
    # code block to execute while the condition is True
```

```
count = 0
while count < 5:
    print(count)
    count += 1
```

CONTROL FLOW STATEMENTS

Inside loops, you can use **break** to terminate the loop prematurely and **continue** to skip the rest of the loop's body and jump to the next iteration.

```
for item in sequence:
    if condition:
        # code block
        break
    # code block
```

```
for item in sequence:
    if condition:
        # code block to skip rest of the loop body and start the next iteration
        continue
    # code block
```

```
# Break example in while loop
count = 0
while count < 5:
    print(count)
    count += 1
    if count == 3:
        break  # Stops the loop when 'count' becomes 3

# Output: 0, 1, 2

# Continue example in while loop
count = 0
while count < 5:
    count += 1
    if count == 3:
        continue  # Skips the rest of the loop body and starts the next iteration
    print(count)

# Output: 1, 2, 4, 5
```

```
# Break example in for loop
fruits = ["apple", "banana", "cherry", "orange", "grape"]
for fruit in fruits:
    if fruit == "cherry":
        break  # Stops the loop when 'cherry' is found
    print(fruit)

# Output: "apple", "banana"

# Continue example in for loop
for number in range(1, 6):
    if number == 3:
        continue  # Skips printing '3', but continues with the next iteration
    print(number)

# Output: 1, 2, 4, 5
```

BUILDING DICTIONARIES

Suppose we have a list of fruits, and we want to create a dictionary that counts how many times each fruit appears in the list. Here's how you can do it:

```
# Example list of fruits
fruits = ['apple', 'banana', 'apple', 'cherry', 'banana', 'cherry', 'apple', 'banana']

# Method 1: Using a for loop
fruit_counter1 = {}
for fruit in fruits:
    if fruit not in fruit_counter1:
        fruit_counter1[fruit] = 1
    else:
        fruit_counter1[fruit] += 1
print(fruit_counter1)  # Output: {'apple': 3, 'banana': 3, 'cherry': 2}

# Method 2: Using the get method
fruit_counter2 = {}
for fruit in fruits:
    fruit_counter2[fruit] = fruit_counter2.get(fruit, 0) + 1
print(fruit_counter2)  # Output: {'apple': 3, 'banana': 3, 'cherry': 2}
```

UNPACKING OPERATOR

The asterisk (*) operator is like unpacking a box of toys. It lets you take out all the toys at once, instead of doing it one by one.

```
toys_box = ['car', 'ball', 'doll']
print(*toys_box)  # prints: car ball doll
```

ADDITIONAL BUILT-IN FUNCTIONS

ZIP

Zip() in Python merges lists or tuples together by creating pairs of corresponding elements from each group.

```
list1 = ['a', 'b', 'c']
list2 = [1, 2, 3]
result = zip(list1, list2)
print(list(result)) # [('a', 1), ('b', 2), ('c', 3)]
```

```
list_of_tuples = [('a', 1), ('b', 2), ('c', 3)]
list1, list2 = zip(*list_of_tuples)
print(list1)  # ('a', 'b', 'c')
print(list2)  # (1, 2, 3)
```

ENUMERATE

enumerate() in Python adds a counter to an iterable, letting you track the index while looping.

```
list1 = ['apple', 'banana', 'cherry']
result = enumerate(list1)
print(list(result)) # [(0, 'apple'), (1, 'banana'), (2, 'cherry')]
```

```
list1 = ['apple', 'banana', 'cherry']
for i, item in enumerate(list1):
    print(i, item)
# Output: 0 apple
#         1 banana
#         2 cherry
```

RANGE

The **range()** function in Python generates a sequence of numbers within the given range. It's often used in loops to control the number of iterations.

range() can be used with one, two, or three arguments:

- **range(stop)**: Generates numbers from 0 up to but not including stop. For example, **range(5)** will generate numbers 0 through 4.
- **range(start, stop)**: Generates numbers from start up to but not including stop. For example, **range(1, 5)** will generate numbers 1 through 4.
- **range(start, stop, step)**: Generates numbers from start up to but not including stop, incrementing by step. For example, **range(1, 10, 2)** will generate numbers 1, 3, 5, 7, 9.

RANGE EXAMPLES

```python
# Example 1: With only stop provided
print("Example 1:")
for number in range(5):
    print(number)
# Output: 0, 1, 2, 3, 4

# Example 2: With start and stop provided
print("\nExample 2:")
for number in range(2, 8):
    print(number)
# Output: 2, 3, 4, 5, 6, 7

# Example 3: With start, stop, and step provided
print("\nExample 3:")
for number in range(1, 10, 2):
    print(number)
# Output: 1, 3, 5, 7, 9

# Creating a list using range
print("\nCreating a list:")
my_list = list(range(3, 15, 3))
print(my_list)
# Output: [3, 6, 9, 12]
```

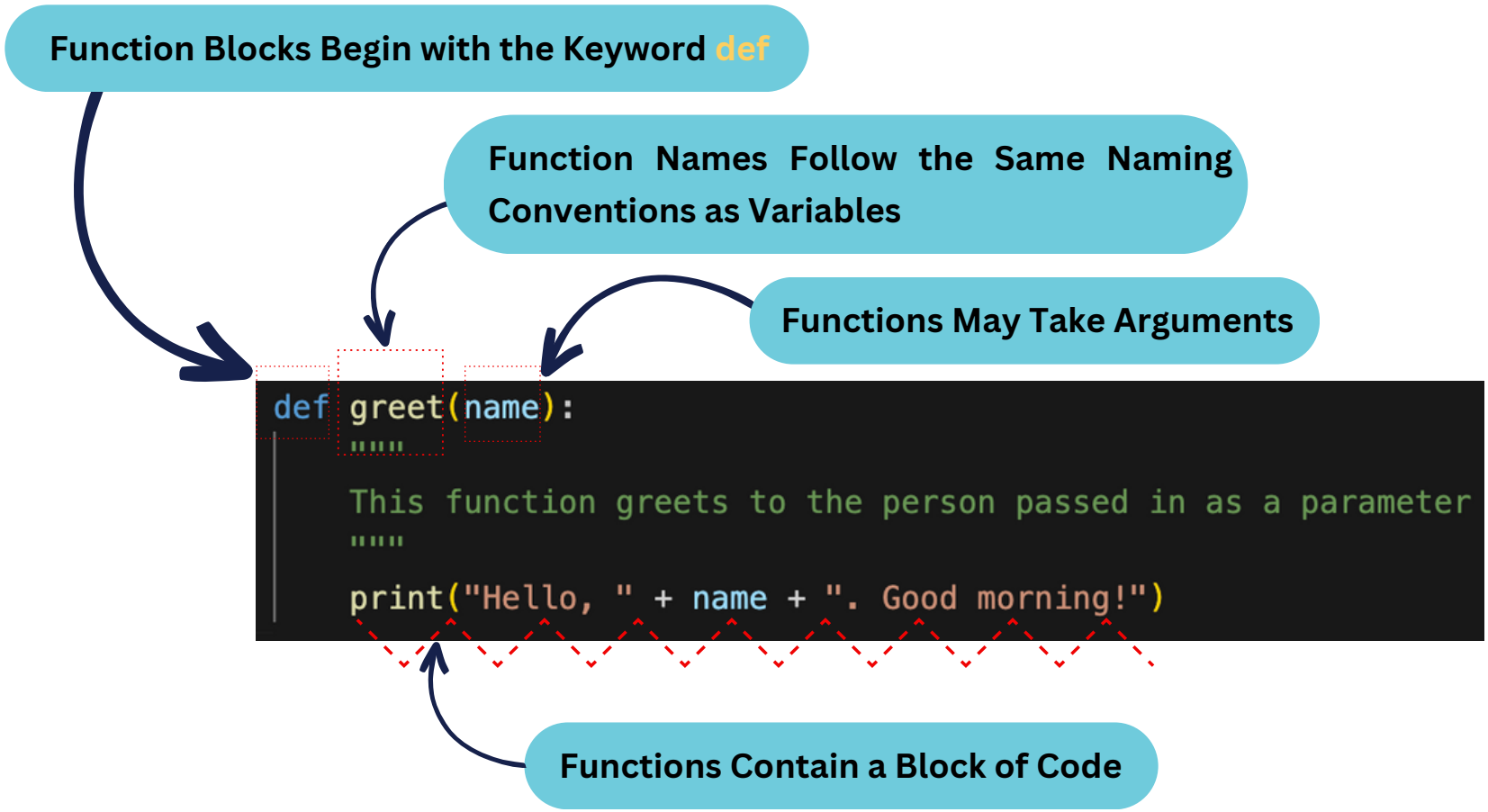
LIST COMPREHENSIONS

Python list comprehensions offer a concise and efficient way to create lists by applying functions to iterable objects, eliminating the need for separate steps to create and populate lists.

```python
# Example of list comprehension in Python

# Define a list of numbers
numbers = [1, 2, 3, 4, 5]

# Use a list comprehension to create a new list of squares of each number
squares = [n**2 for n in numbers]
print(squares)  # Output: [1, 4, 9, 16, 25]

# List comprehension with a conditional statement
# This creates a list of squares for numbers greater than 2
squares_above_2 = [n**2 for n in numbers if n > 2]
print(squares_above_2)  # Output: [9, 16, 25]

# Example of a nested list comprehension
# Define a 2-dimensional list (list of lists)
matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]

# Use a nested list comprehension to flatten the list
flattened = [value for sublist in matrix for value in sublist]
print(flattened)  # Output: [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

DEFINING FUNCTIONS

DEFINING FUNCTIONS



DEFINING FUNCTIONS

```python
def add_numbers(num1, num2):
    """
    This function adds two numbers passed in as parameters
    """
    result = num1 + num2
    return result
```

Functions May End With return

Function With No Arguments

```python
def say_hello():
    """This function prints a greeting message"""
    print("Hello, World!")
```

CALLING FUNCTIONS

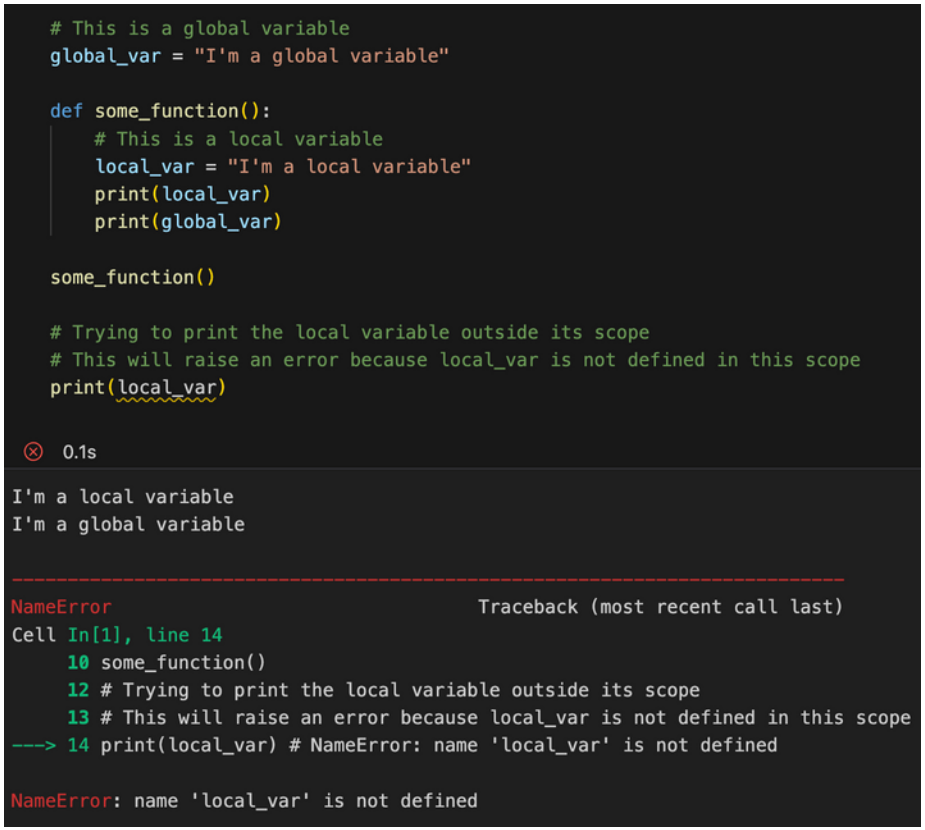
- **greet('Mahmoud')** is telling the computer to use the greet function with 'Mahmoud' as input. The greet function says hello to whatever name you give it. So, when you run greet('Mahmoud'), the computer says "Hello, Mahmoud. Good morning!".
- **sum = add_numbers(5, 10)** is giving the computer two numbers (5 and 10) and telling it to add them together using the add_numbers function. The computer does the math and finds that 5 + 10 equals 15. Then it stores this result (15) in the sum variable, so you can use it later.
- **say_hello()** is a command that tells the computer to run the say_hello function, which doesn't need any extra information to work. This function simply prints out the message "Hello, World!" every time it's called.

```
# Calling the greet function
greet('Mahmoud') # Output: Hello, Mahmoud. Good morning!

# Calling the add_numbers function
sum = add_numbers(5, 10) # Output: 15

# Calling the say_hello function
say_hello()  # outputs: Hello, World!
```

VARIABLE SCOPE

VARIABLE SCOPE

- **Global scope**: These variables are declared outside any function or are not in any class. They are accessible throughout the code, including inside functions, classes, or modules.
- **Local scope**: These variables are declared inside a function or method. They can be accessed only inside the function or method in which they were declared.



```python
# This is a global variable
global_var = "I'm a global variable"

def some_function():
    # This is a local variable
    local_var = "I'm a local variable"
    print(local_var)
    print(global_var)

some_function()

# Trying to print the local variable outside its scope
# This will raise an error because local_var is not defined in this scope
print(local_var)
```

```
⊗ 0.1s
I'm a local variable
I'm a global variable

---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[1], line 14
     10 some_function()
     12 # Trying to print the local variable outside its scope
     13 # This will raise an error because local_var is not defined in this scope
---> 14 print(local_var) # NameError: name 'local_var' is not defined

NameError: name 'local_var' is not defined
```

VARIABLE SCOPE

- If a variable is defined in the local scope, you can access it directly.
- If a variable is defined in the global scope, you can access it directly from the global scope or from a local scope.
- If you want to modify a global variable from inside a local scope, you need to declare it as global inside the local scope, like this:

```
# This is a global variable
global_var = "I'm a global variable"

def some_function():
    # Declare global_var as global
    global global_var
    # Now you can modify it
    global_var = "I'm a modified global variable"

# Call the function to modify global_var
some_function()

print(global_var) # Output: I'm a modified global variable
```

VARIABLE SCOPE

Nonlocal variables are used in nested functions where the variable should not belong to the inner function. Use the nonlocal keyword to declare a variable as **nonlocal**.

```
def outer_function():
    # This is a variable in the outer function
    outer_var = "I'm an outer variable"

    def inner_function():
        # Declare outer_var as nonlocal
        nonlocal outer_var
        # Now you can modify it
        outer_var = "I'm a modified outer variable"

    # Call the inner function to modify outer_var
    inner_function()

    # Print the modified outer_var
    print(outer_var)

# Call the outer function
outer_function() # Output: I'm a modified outer variable
```

DOCUMENTATION

INLINE COMMENTS

Inline comments are concise explanations in code, marked with "**#**" in Python for context.

```
x = 10  # Initialize variable x with the value 10
```

DOCSTRINGS

Docstrings are Python's standard way of documenting functions, methods, classes, and modules. They're enclosed by triple quotes (""" or ''') right under the definition.

```python
def add(a, b):
    """
    Adds two numbers together and returns the result.

    Args:
    a (int or float): The first number to add.
    b (int or float): The second number to add.

    Returns:
    int or float: The result of adding a and b.
    """
    return a + b
```

✓ 0.0s

```python
help(add)#prints the docstring of the function
```

✓ 0.0s

```
Help on function add in module __main__:

add(a, b)
    Adds two numbers together and returns the result.

    Args:
    a (int or float): The first number to add.
    b (int or float): The second number to add.

    Returns:
    int or float: The result of adding a and b.
```

BLOCK COMMENTS

Block comments offer detailed explanations compared to inline comments. They're usually placed at the start of a code file or function, describing its purpose, assumptions, and inputs/outputs.

```
'''
This script takes a CSV file of names and ages, calculates the average age,
and prints the result. It assumes that the CSV file has a header row with the
fields "Name" and "Age", and that all ages are integers.
'''
```

LAMBDA EXPRESSIONS

Lambda expressions in Python are anonymous functions used for short, throw-away functions at the place of declaration. They are concise compared to standard functions. Here's a comparison.

```
# Standard function
def sum(a, b):
    return a + b
print(sum(5, 3))  # Output: 8

# Lambda function
sum = lambda a, b: a + b
print(sum(5, 3))  # Output: 8
```

ADDITIONAL BUILT-IN FUNCTIONS

FILTER

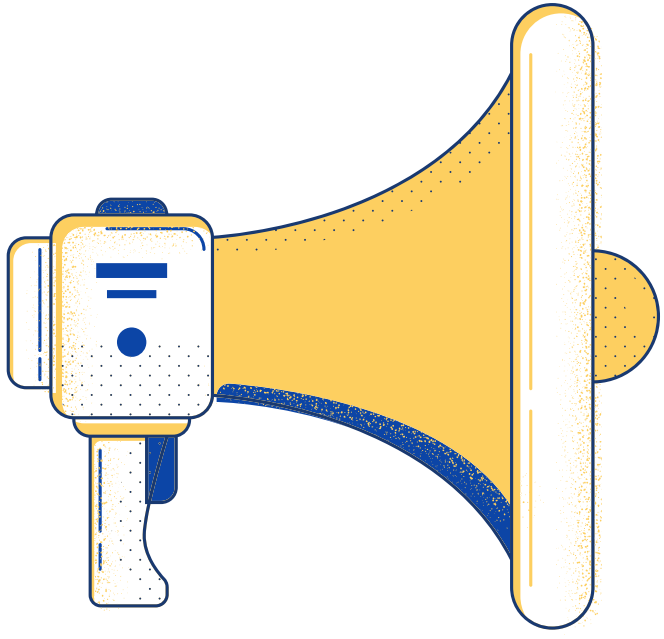
filter() function takes a function and a list as arguments to create a new list containing elements that satisfy a given condition. It forms an iterator with elements for which the function returns **True**.

```
# Function to check if number is even
def is_even(num):
    return num % 2 == 0

numbers = [1, 2, 3, 4, 5, 6]

# Using filter function to filter even numbers
even_numbers = filter(is_even, numbers)

# Convert the filter object to a list, if you want to print it
even_numbers_list = list(even_numbers)

print(even_numbers_list)  # Output: [2, 4, 6]
```

MAP

map() function takes a function and a list as arguments, applying the function to all elements in the list and returning a new list with the changed elements. It can also handle multiple iterables, executing the function until the smallest iterable is exhausted.

```python
# Function to square a number
def square(num):
    return num ** 2

numbers = [1, 2, 3, 4, 5]

# Using map function to square all numbers in the list
squared_numbers = map(square, numbers)

# Convert the map object to a list, if you want to print it
squared_numbers_list = list(squared_numbers)

print(squared_numbers_list)  # Output: [1, 4, 9, 16, 25]
```



Q&A Session: Let's explore and understand together

RESOURCES

- Control Flow
- Functions
- Oman Tourist Data Explorer
- Online Python
- Session-8 jupyter notebook

Your presence today has added value to our shared learning journey. Thank you for joining us!