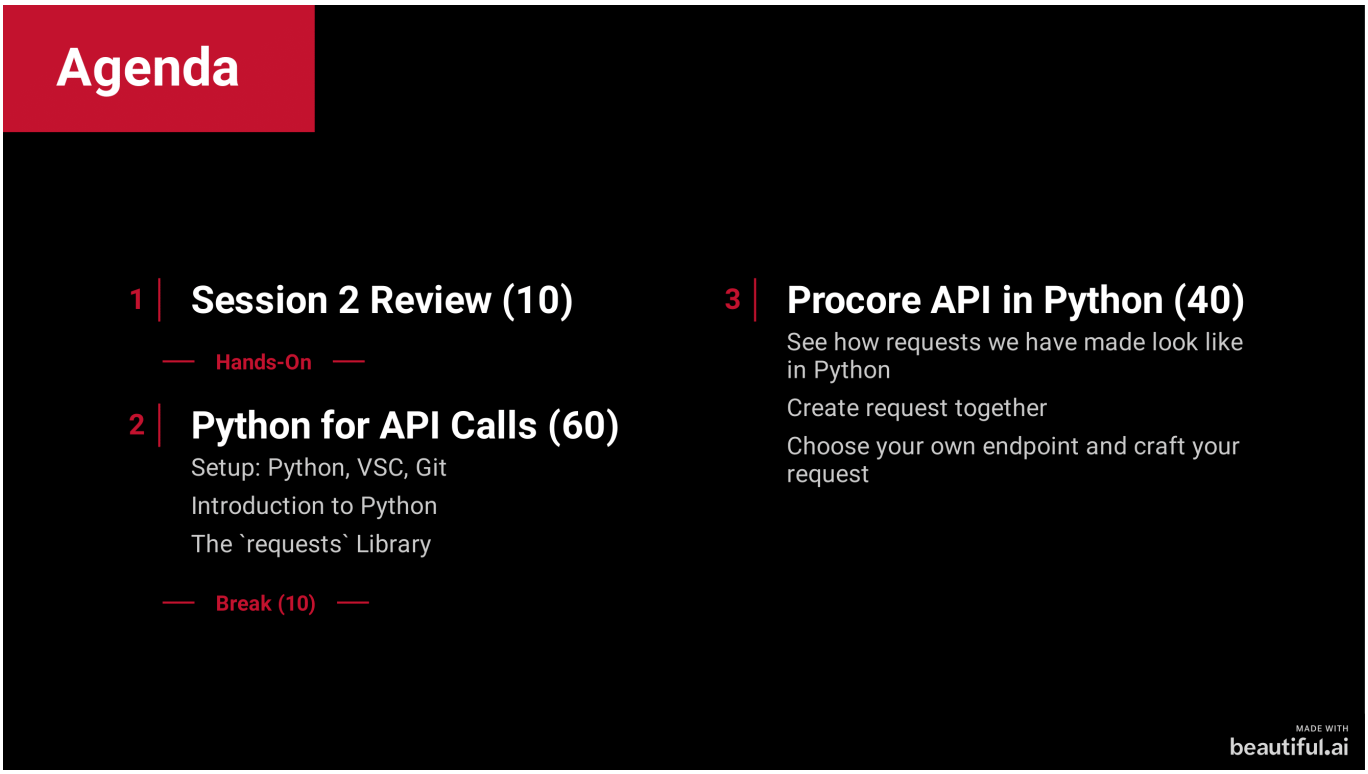
Session 3: Using Python for API Calls

Leveling up from Postman to Python

Link to Presentation

Slide 1: Title

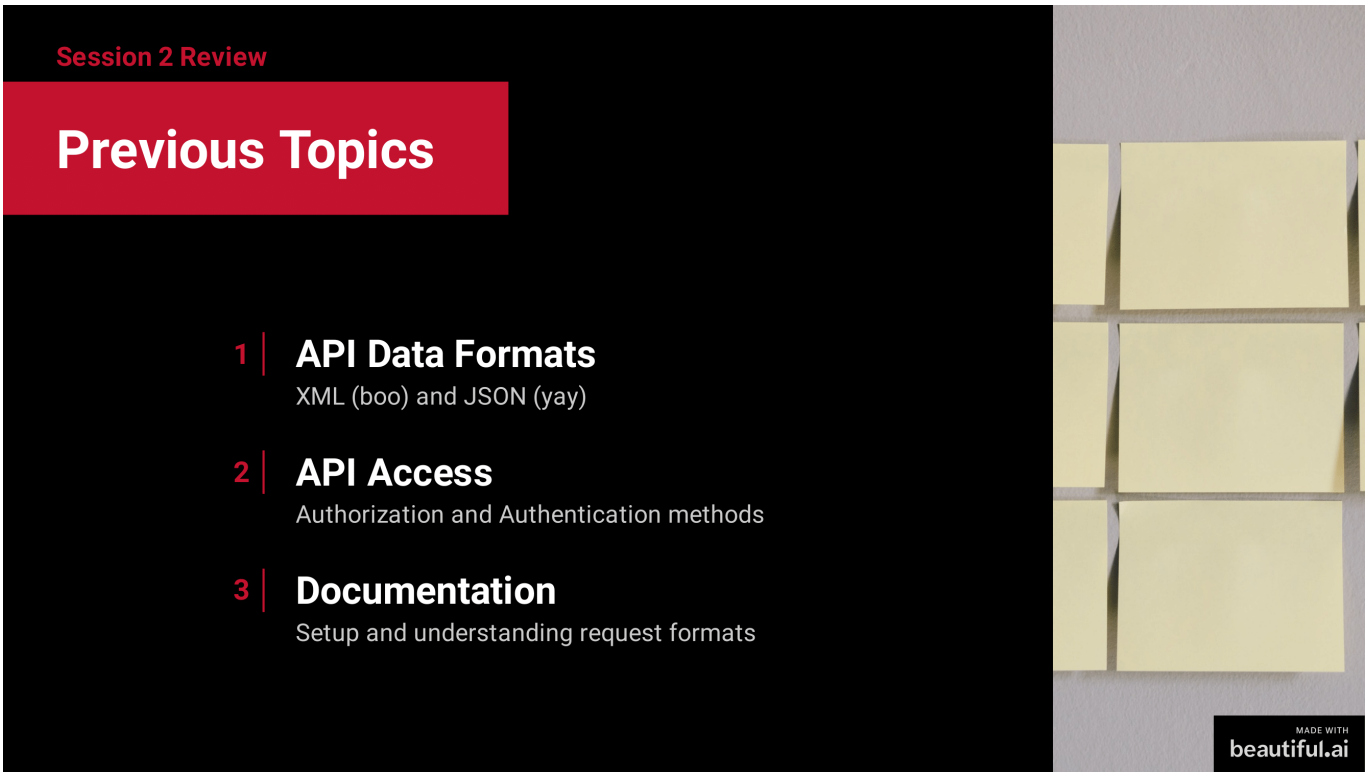
Slide 2: Agenda



Session 2 Review

▶ Click to Expand

Slide 4: Previous Topics



1. **API Data Formats**: XML (boo) and JSON (yay)
2. **API Access**: Authorization and Authentication methods
3. **Documentation**: Setup and understanding request formats

What does JSON stand for?

Slide 6: JSON Data

What is JSON and what are some key characteristics

JSON (JavaScript Object Notation) is:

- **Definition**: a lightweight data interchange format that is easy for both humans and machines to read and write.
- **Purpose**: used to structure and represent data in a format that is efficient for data exchange between different software systems.

JSON in API Context

- **Data Exchange**: APIs often use JSON as a format for exchanging structured data between clients and servers.
- **Data Representation**: JSON represents data as key-value pairs, arrays, and nested objects, making it suitable for various types of information.
- **Data Types**: JSON supports basic data types such as strings, numbers, booleans, arrays, and objects.
- **Simplicity**: JSON's syntax is less verbose than XML, which contributes to its simplicity and ease of use.

JSON Structure

- **Objects**: JSON data is organized into objects, which consist of key-value pairs enclosed in curly braces ({}).
- **Arrays**: Arrays in JSON are ordered lists of values enclosed in square brackets ([]).
- **Values**: Values can be strings, numbers, booleans, objects, arrays, or null.
- **Keys**: Keys are strings that represent the names of values within objects.

JSON Example

```
{
    "bookstore": {
        "books": [
```

```
            {
                "category": "Fiction",
                "title": "The Great Gatsby",
                "author": "F. Scott Fitzgerald",
                "price": 10.99
            },
            {
                "category": "Non-Fiction",
                "title": "Sapiens",
                "author": "Yuval Noah Harari",
                "price": 15.95
            }
        ]
    }
}
```

❓ What is the difference between Authentication and Authorization?

Slide 8: Authentication versus Authorization

Differences between the widely interchanged words

Authentication

Authentication is the process of verifying the identity of a user, system, or entity. It ensures that the person or entity claiming access to a system or resource is indeed who they say they are. Authentication is the first step in the security process and is typically based on providing credentials such as a username and password, a security token, a fingerprint, or other forms of identity verification. Once authenticated, a user gains access to a system or application.

Authorization

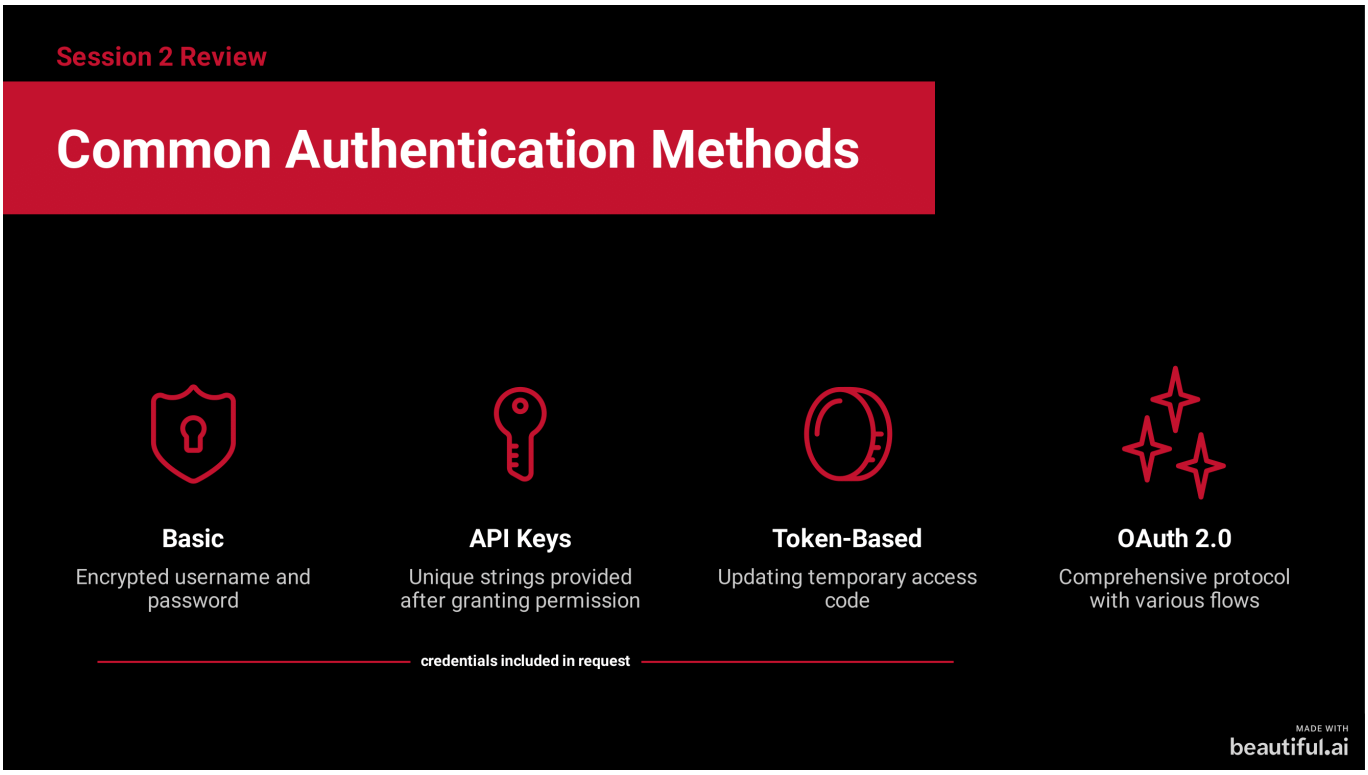
Authorization, on the other hand, comes after authentication and involves granting or denying access to specific resources or actions based on the authenticated user's permissions. In essence, authorization determines what actions a user or entity is allowed to perform within a system or application. Authorization is typically defined by roles, permissions, or access levels that are associated with the authenticated user. It ensures that users only have access to the functionalities and data they are entitled to based on their roles or privileges.

TL;DR

- **Authentication** is about confirming the identity of a user.
- **Authorization** is about granting or denying access to specific resources or actions based on the user's verified identity and permissions

What are some common authentication methods?

Slide 10: Common Authentication Methods



API Keys

- API keys are simple and widely used for authentication.
- They are unique alphanumeric strings issued to clients (applications or users) by the API provider.
- Clients include the API key in the request headers or query parameters to authenticate themselves.
- API keys are suitable for public APIs with lower security requirements.

Bearer Token Authentication (Token-based)

- Bearer token authentication is used with tokens like JWT (JSON Web Token) or OAuth 2.0 access tokens.

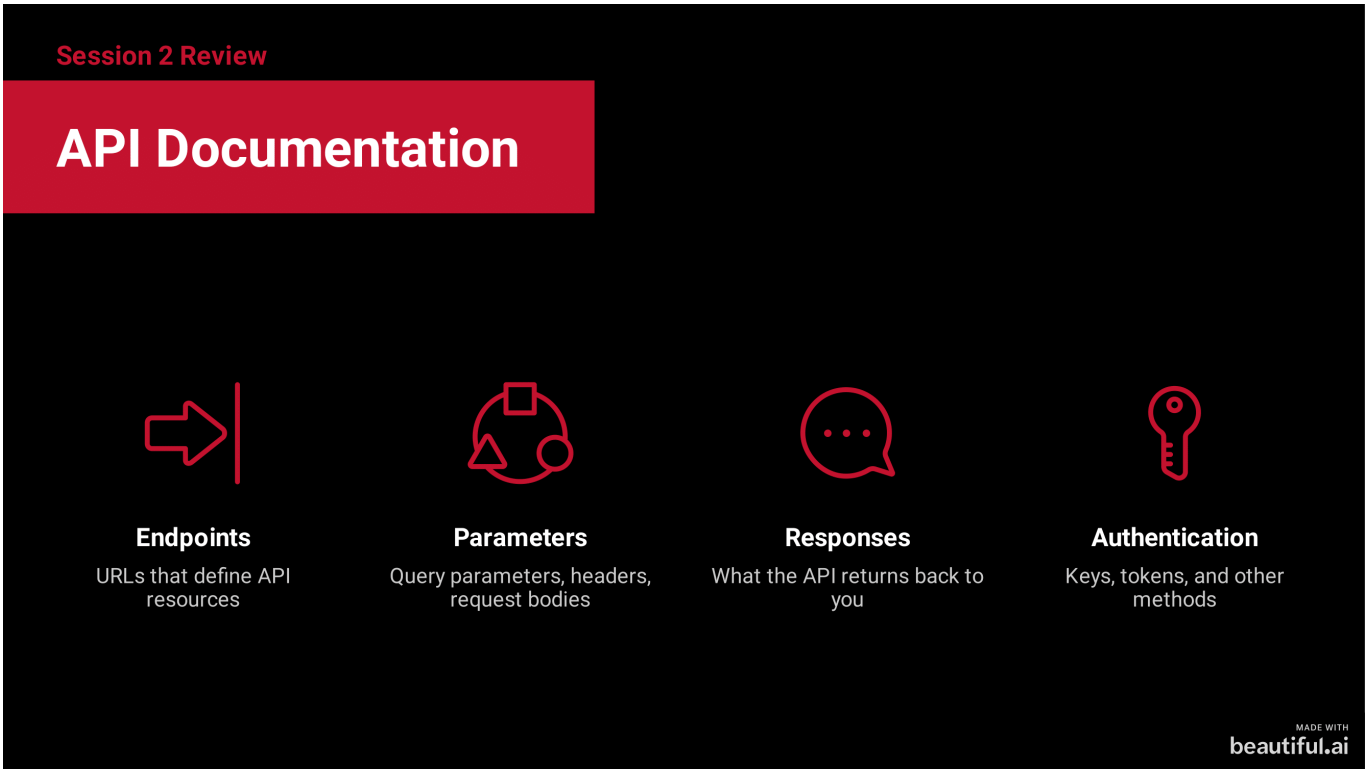
- After successful authentication, clients receive an access token, which they include in the request headers.
- The server validates the token to authorize the client's access to resources.
- Bearer token authentication provides flexibility and scalability.

Basic Authentication

- Basic Authentication involves sending a username and password in the request headers.
- The credentials are typically base64-encoded (but not encrypted), making it important to use HTTPS for secure transmission.
- While simple to implement, Basic Authentication is less secure due to the risk of credentials being intercepted.

Slide 11: API Documentation

A guidebook to the API's capabilities and usage



Key components of API documentation include:

- **Endpoints**: These are URLs that define specific functions or resources the API provides.
- **Parameters**: These are inputs required to customize your API requests, such as query parameters, headers, or request bodies.
- **Responses**: Documentation explains what data the API returns in response to different requests.
- **Authentication**: Details about how to authenticate and authorize your requests using API keys, tokens, or other methods.

Some examples:

- Procore API Docs
- OpenWeatherMap API Docs

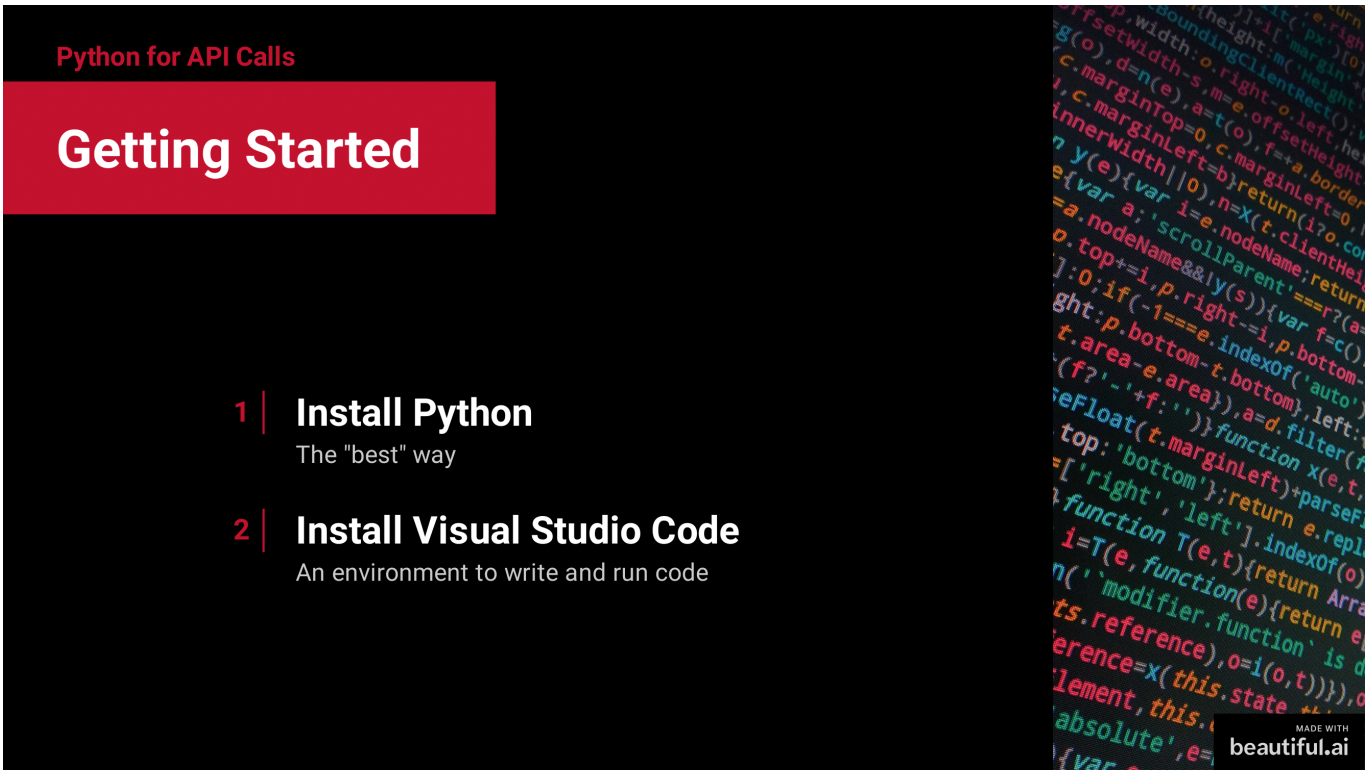
- NASA API Docs

Hands-On: Python Basics

▶ Click to Expand

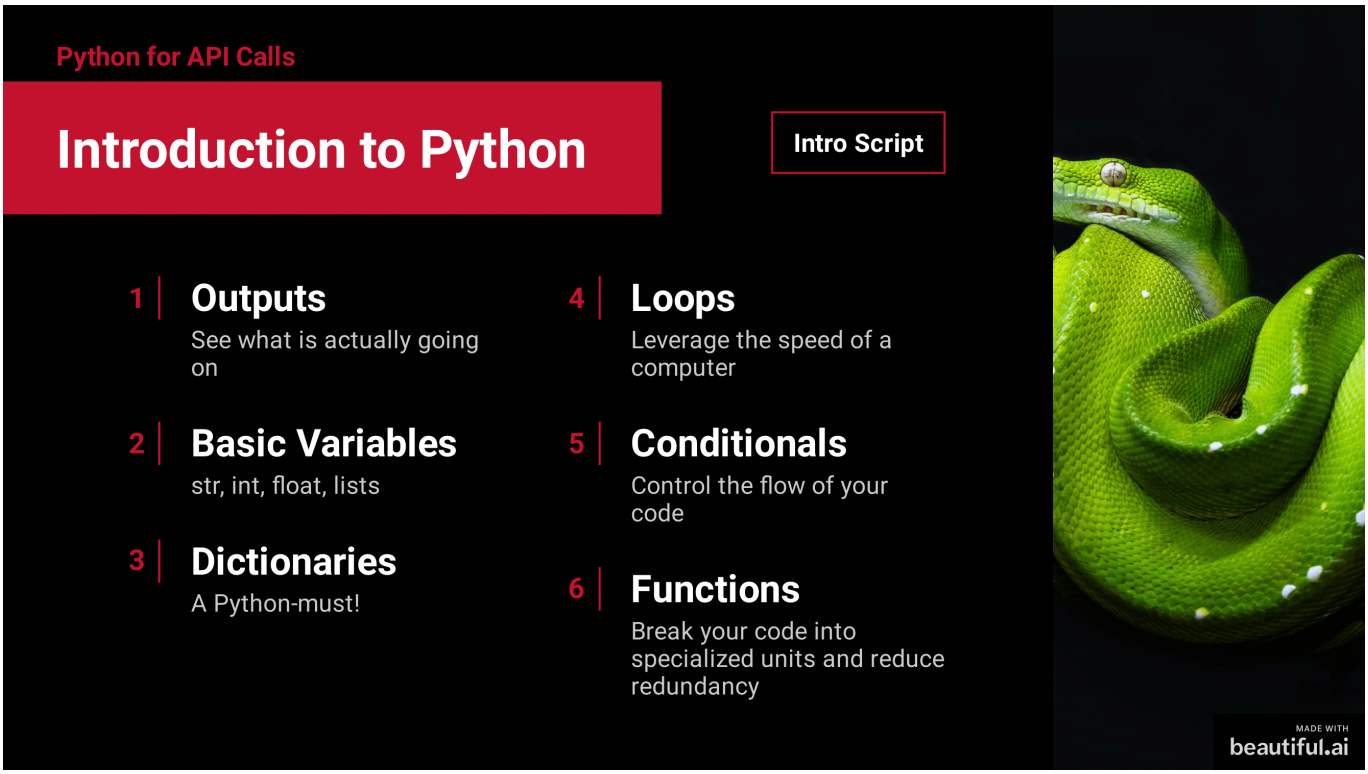
Slide 13: Getting Started

Get Python up and everything else you need



1. Install Python
    - Naturally you will need this program to run Python scripts!
    - Playbook Article
2. Install Visual Studio Code
    - A nice environment to code in with a lot of features and plug-ins (including AI tools!)
    - Playbook Article - same article as above, just look in different section!

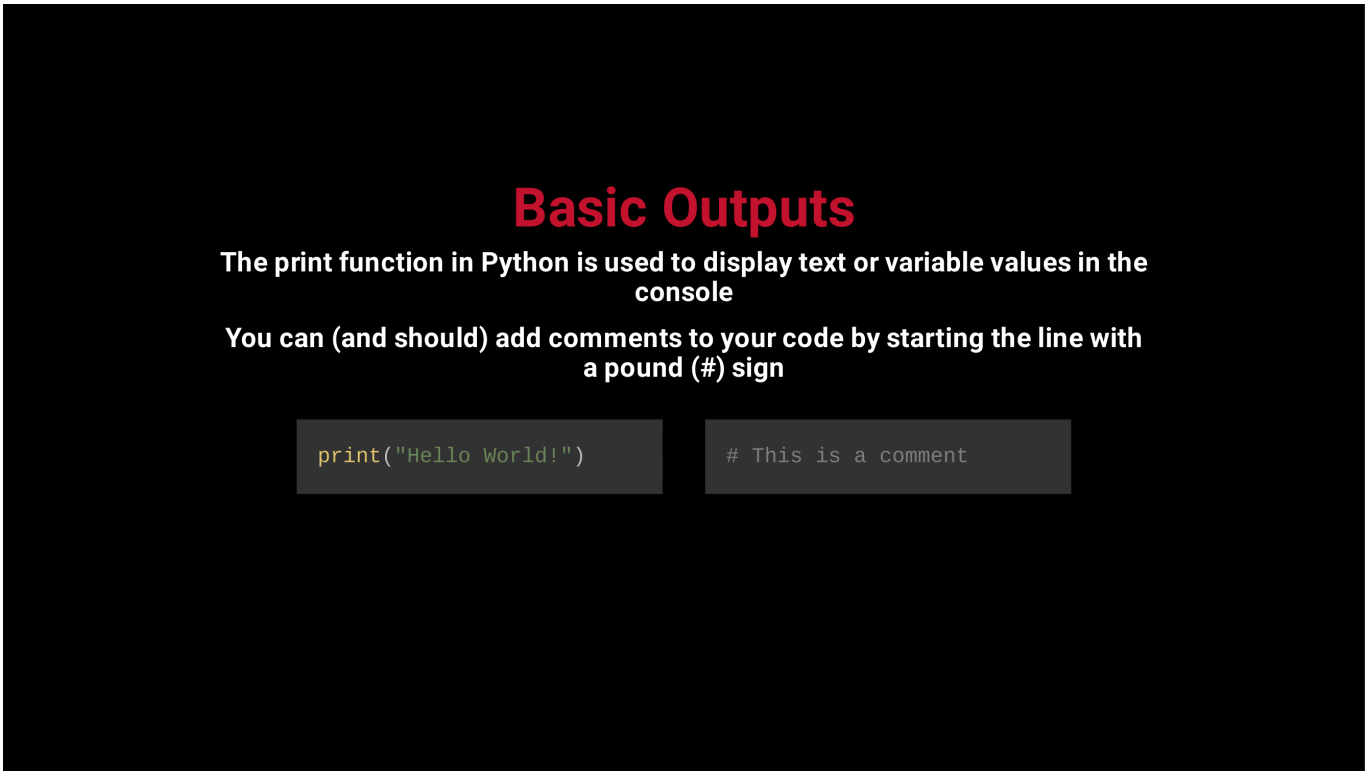
Slide 15: Introduction to Python



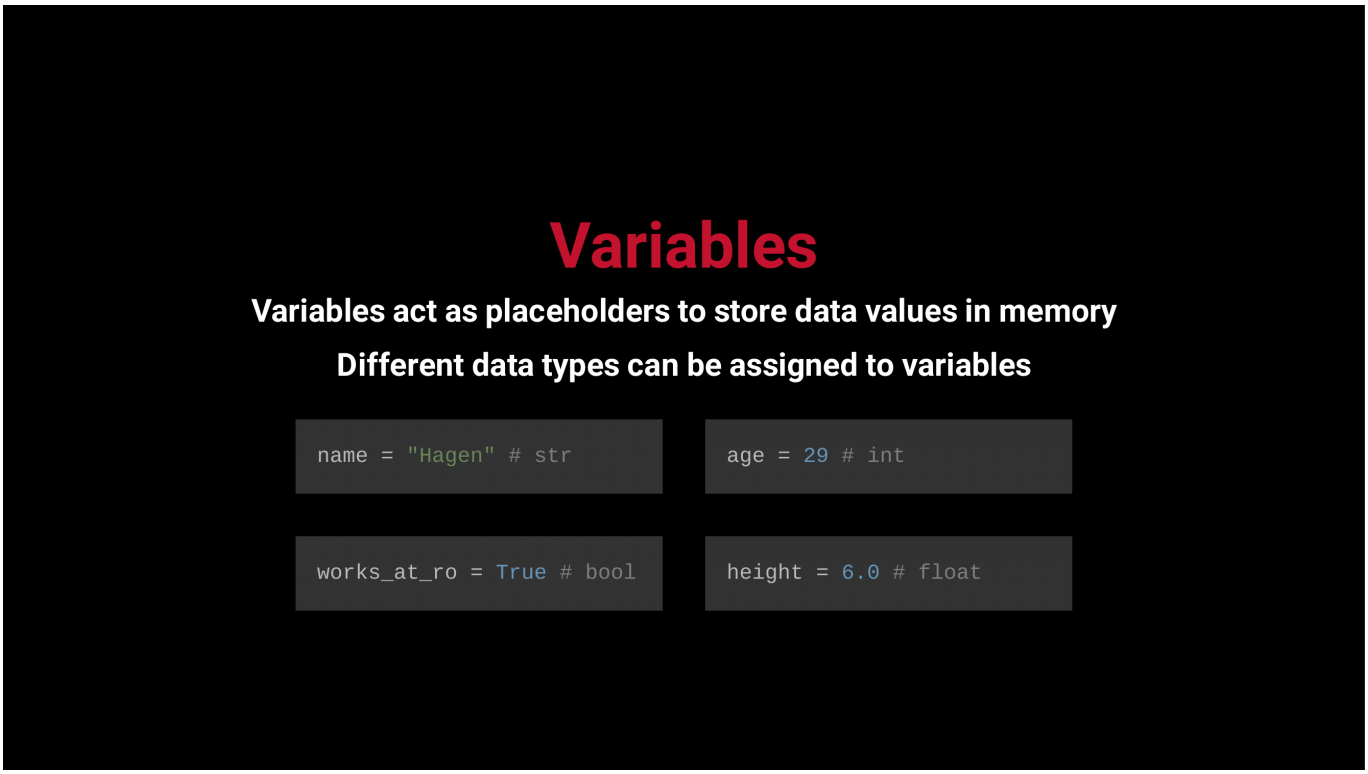
Slide 16: Basic Outputs

The print function in Python is used to display text or variable values in the console.

```
print("Hello, World!")
```



Slide 17: Variables



Variables Types

Variables act as placeholders to store data values in memory. Different data types can be assigned to variables such as:

- String: Textual data

```
name = "Hagen"
```

- Integer: Whole numbers

```
age = 29
```

- Float: Decimal numbers

```
height = 6.0
```

- Boolean: binary true or false (0 or 1)

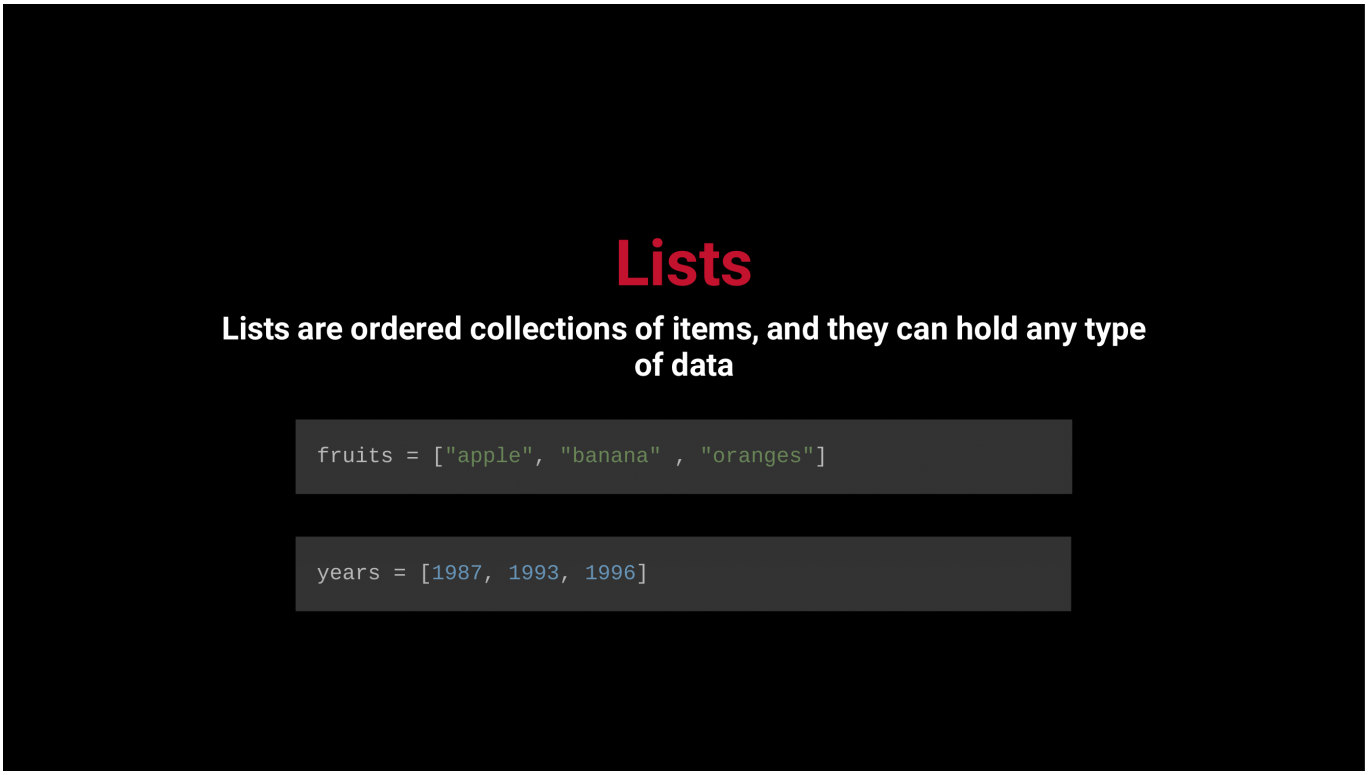
```
works_at_ro = True
```

> Unlike other languages, Python does not require you to declare the type of variable when declaring it. Python will figure out the variable type for you when you run your code.

Variable Naming

- **Starting Character**: Variable names must start with a letter (a-z, A-Z) or an underscore (_). The rest of the name can contain letters, numbers, or underscores.
- **Case-Sensitive**: Variable names are case-sensitive (age, Age, and AGE are three different variables).
- **Reserved Words**: Python has defined keywords like `if`, `else`, `while`, etc.) that cannot be used as variable names.

Slide 18: Lists



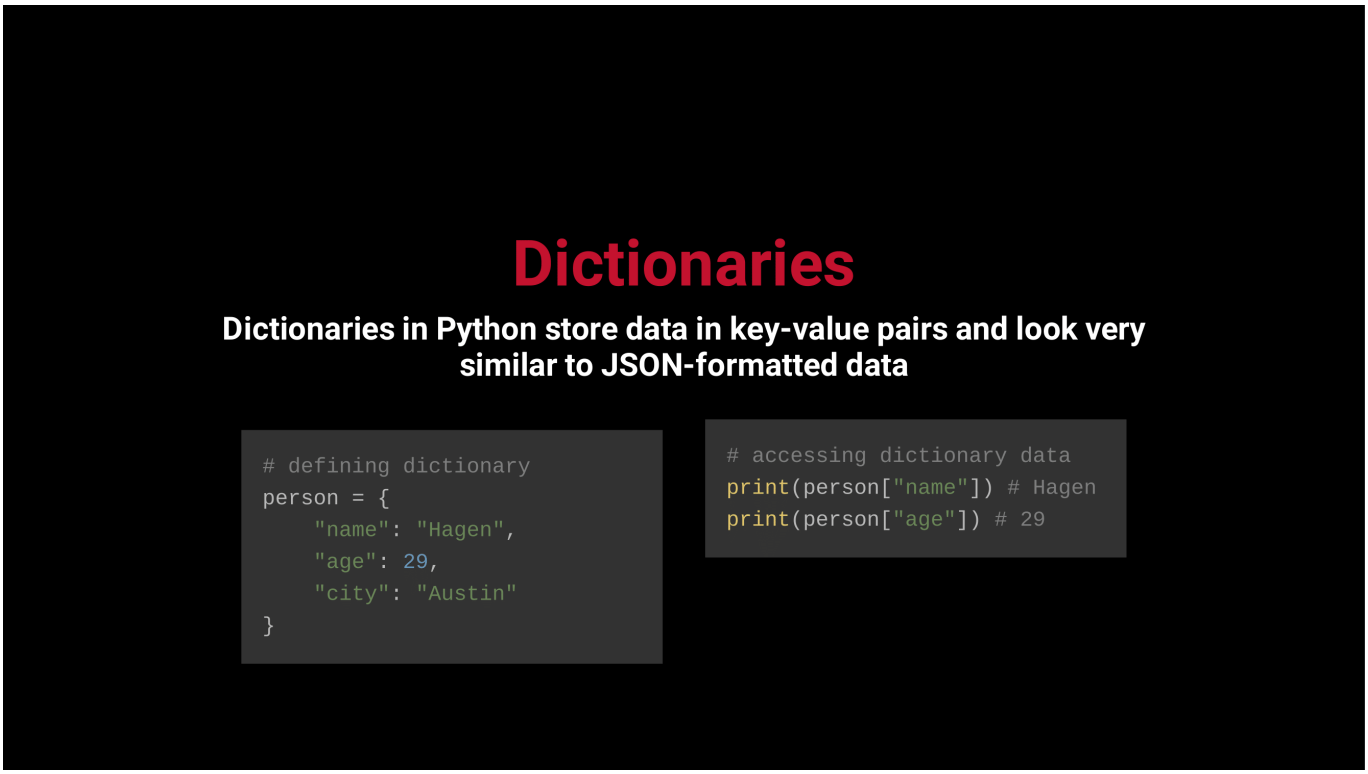
Lists are ordered collections of items, and they can hold any type of data.

```
fruits = ["apple", "banana", "cherry"]
```

Items in lists can be accessed by their index, with indices starting from 0 for the first item.

```
print(fruits[0])
```

Slide 19: Dictionaries



Dictionaries in Python store data in key-value pairs and look very similar to JSON-formatted data:

```
person = {
    "name": "Hagen",
    "age": 29,
    "city": "Austin"
}
```

Values in a dictionary can be accessed using their respective keys:

```
print(person["name"]) # Hagen
print(person["age"]) # 29
```

Slide 20: Advanced Outputs: F-Strings (Formatted String Literals)

Introduced in Python 3.6, f-strings offer a concise way to embed expressions inside string literals. They are prefixed with an 'f' and use curly braces {} to embed Python expressions within the string.

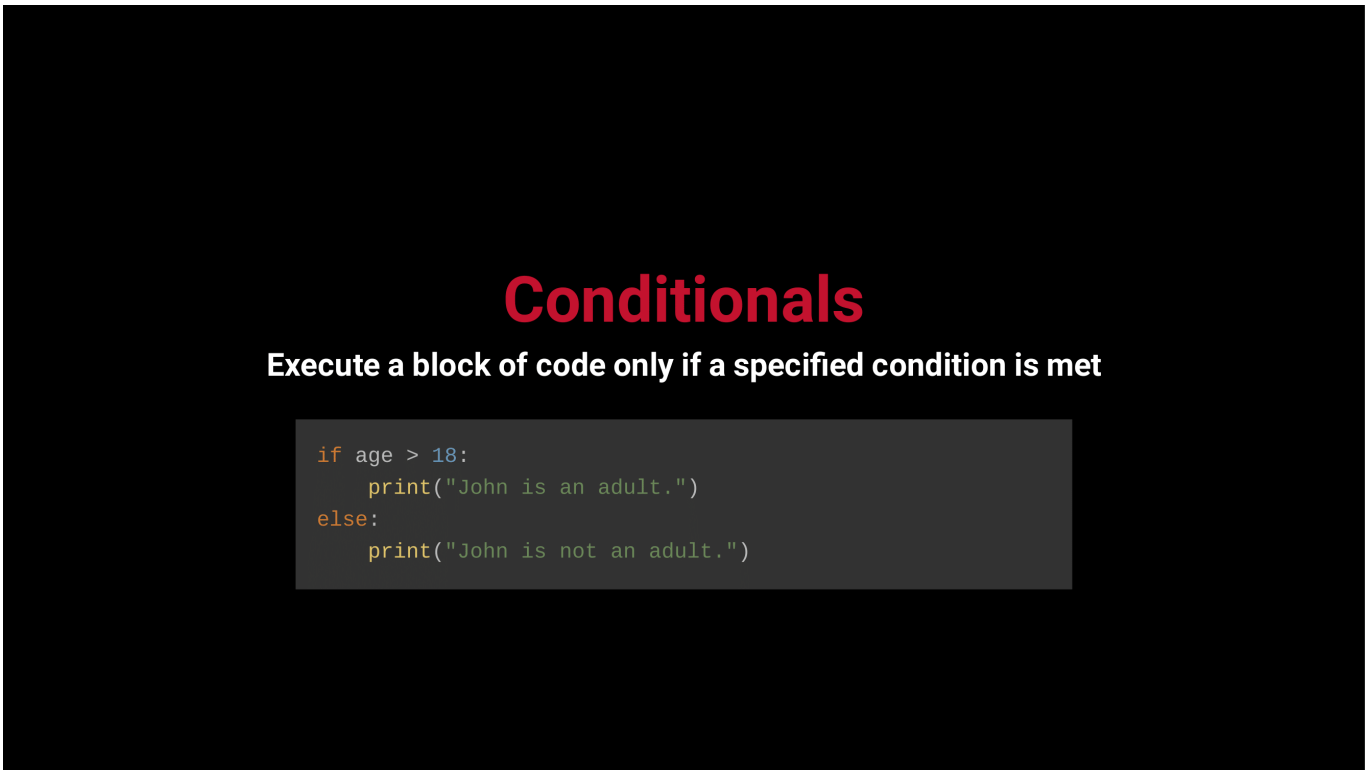
```
name = "John"
age = 25
greeting = f"My name is {name} and I am {age} years old."
print(greeting)  # Output: My name is John and I am 25 years old.
```

You can also perform operations within the curly braces of an f-string.

```
double_age = f"Twice my age is {age * 2}."
print(double_age)  # Output: Twice my age is 50.
```

F-strings provide a readable and convenient way to include variable values and expressions directly within strings, making code cleaner and more intuitive.

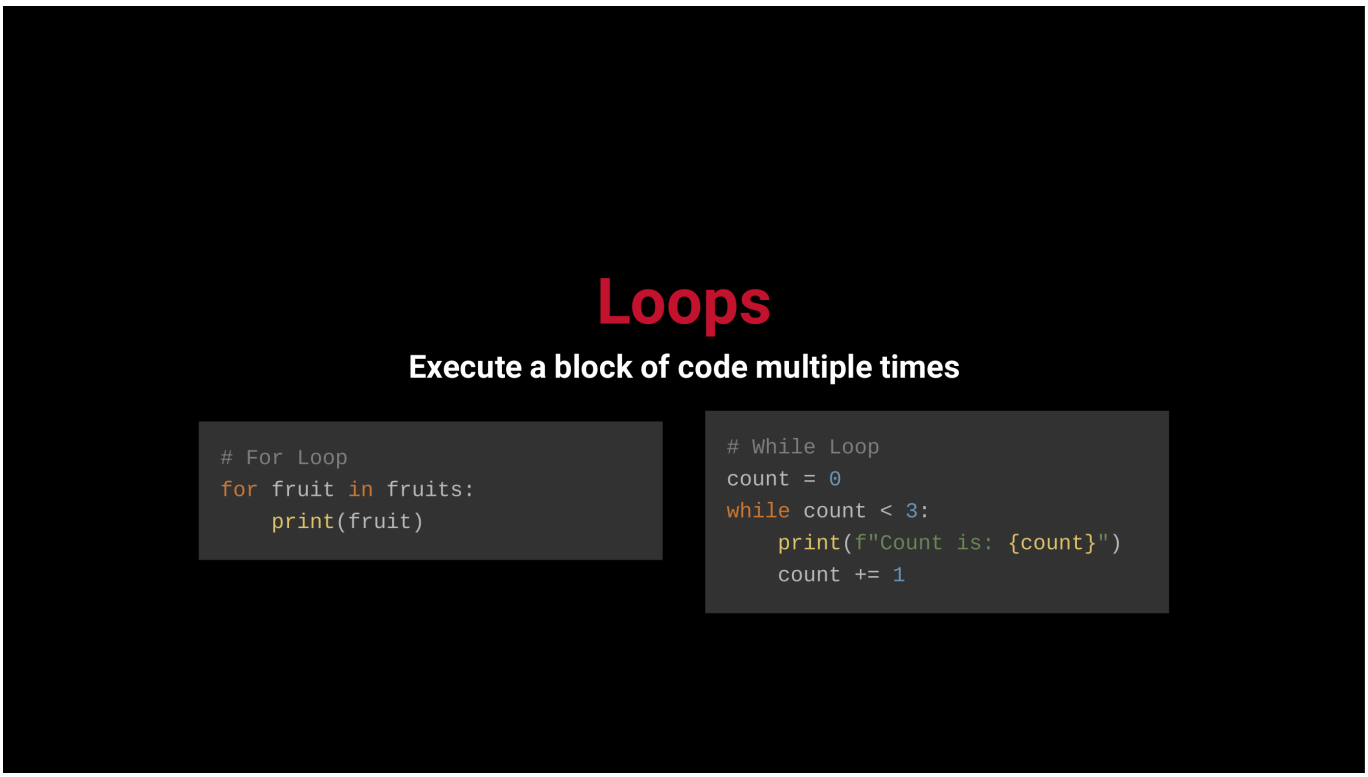
Slide 21: Conditionals



Conditionals allow for the execution of a block of code only if a specified condition is met.

```
if age > 18:
    print("John is an adult.")
else:
    print("John is not an adult.")
```

Slide 22: Loops



Loops in Python are used to execute a block of code multiple times.

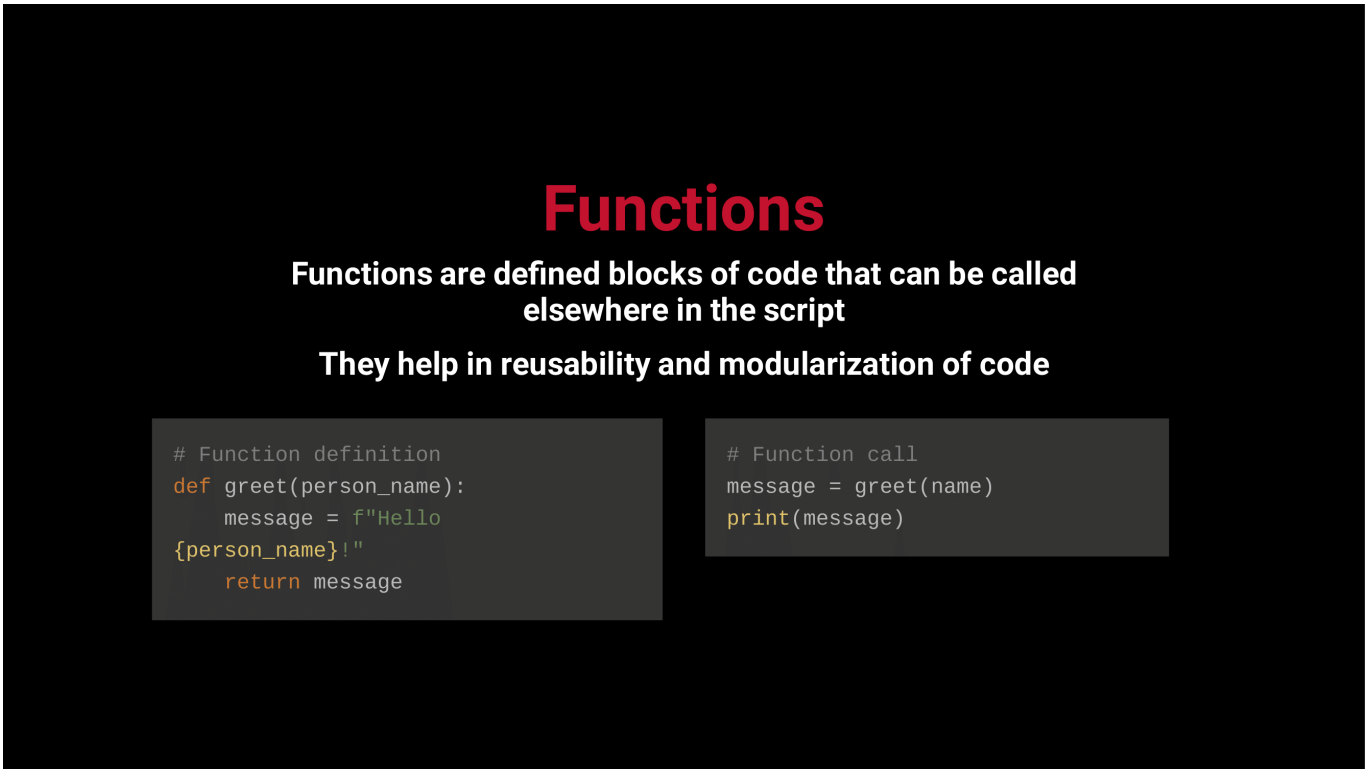
- for loop: Iterates through a list or range.

```
for fruit in fruits:
    print(fruit)
```

- while loop: Executes as long as a specified condition remains true.

```
count = 0
while count < 3:
    print(f"Count is: {count}")
    count += 1
```

Slide 23: Functions



- **Definition**: Functions are blocks of organized and reusable code designed to perform a specific task. They are a fundamental concept in programming, allowing for modularity and code reuse.

```
def function_name(parameters):
    """docstring: provides a brief explanation of what the function does, the
input(s), and the output(s)"""
    # function body
    return output
```

- **Parameters and Arguments**

- Parameters are the names listed in the function's definition.
- Arguments are the real values passed to the function when it's called.

```
def greet(person_name):
    message = f"Hello, {person_name}!"
    return message
```

- **Return Statement**: The return keyword is used to exit a function and return a value.

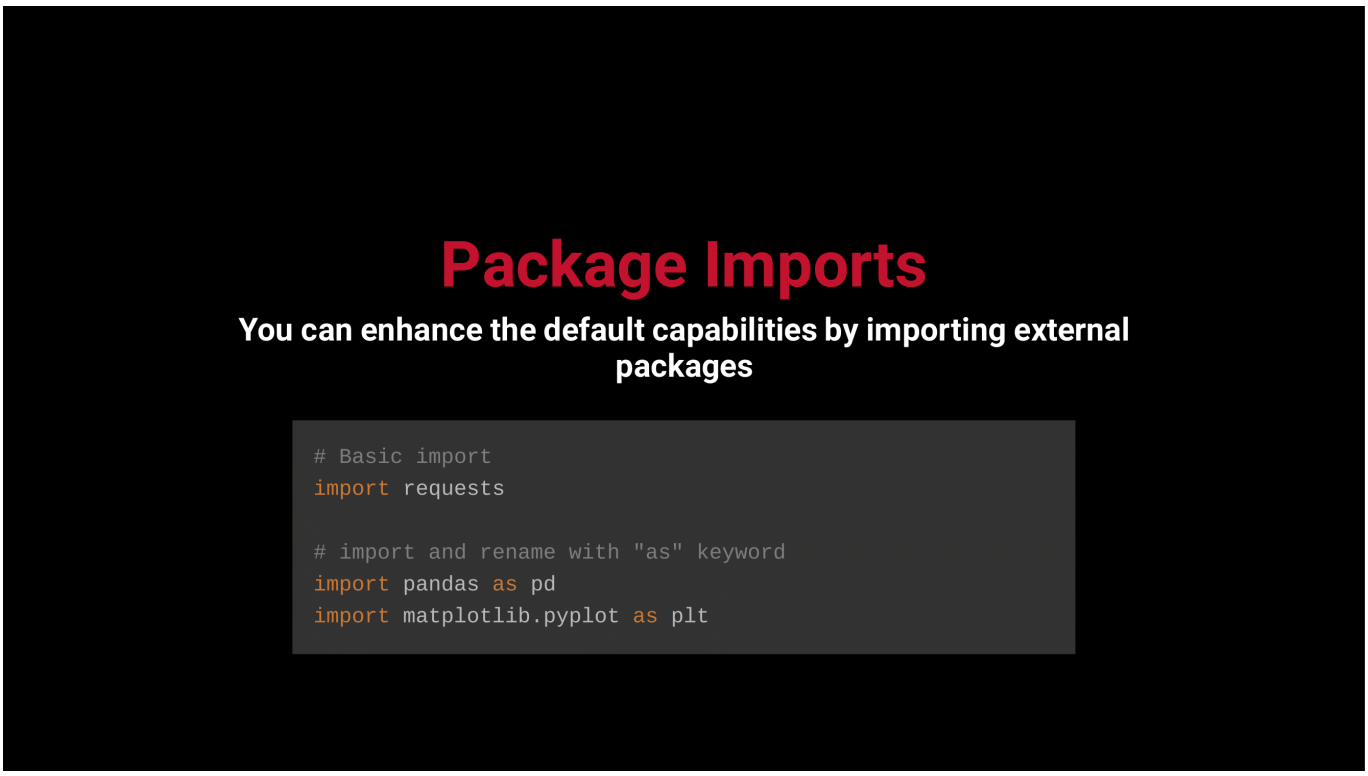
```
def add(x, y):
    return x + y
```

- **Variable Scope**
  - Local Variables: Variables declared inside a function have a local scope, meaning they can only be accessed within that function
  - Global Variables: Variables declared outside of the function (or in global scope) can be accessed inside or outside of the function

```
x = 10  # This is a global variable

def check_value():
    y = 5  # This is a local variable
    return x + y  # Can access global variable 'x' inside this function
```

Slide 24: Package Imports



In Python, we can enhance the default capabilities by importing external packages. This is done using the import keyword. In the script, two packages are imported:

- requests: Used for making HTTP requests.

```
import requests
```

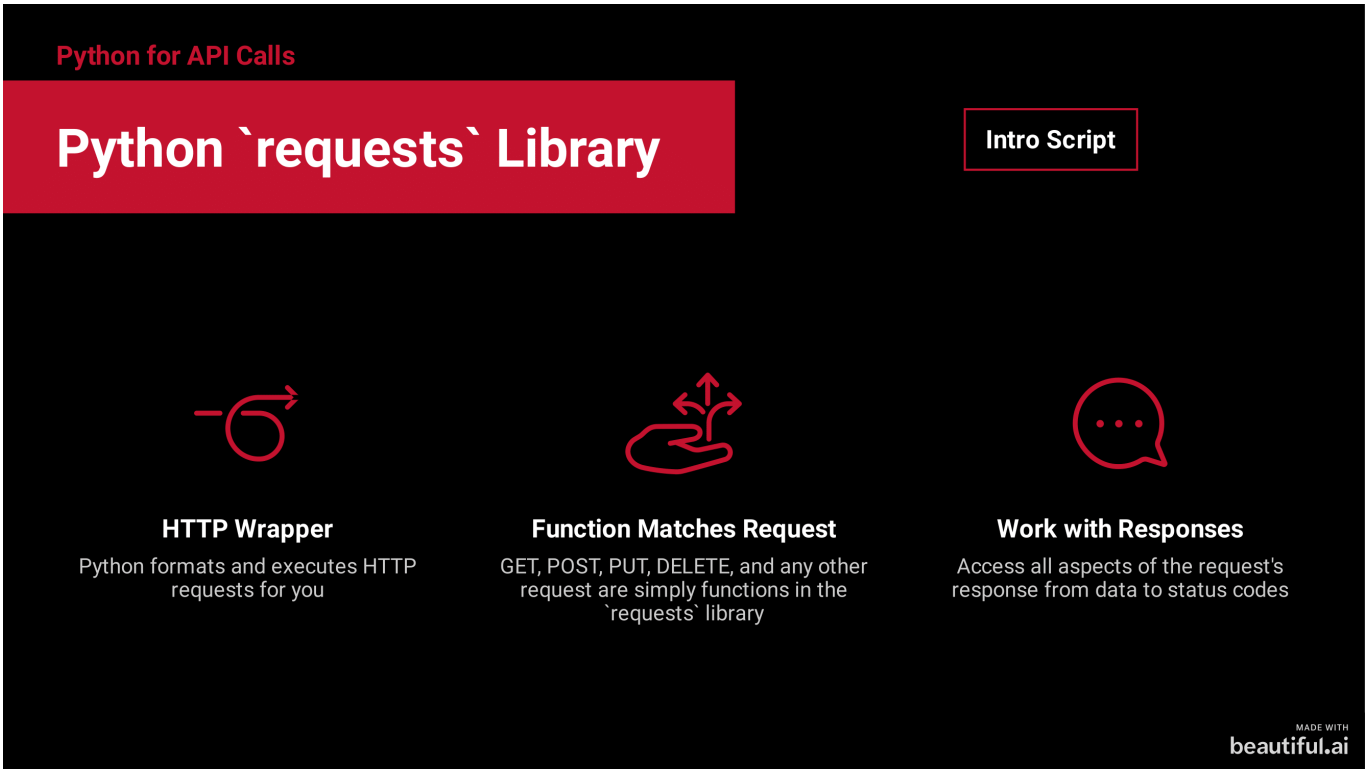
- pandas (often imported as pd): Used for data manipulation and analysis.

```
import pandas as pd
```

Hands-On: API Requests in Python

▶ Click to Expand

Slide 25: Python `requests` Library



1. **HTTP Wrapper**: The requests library is a popular Python package used for making HTTP requests. It abstracts the complexities of making requests behind a simple API, providing an intuitive way to send HTTP requests and handle responses.
2. **Methods**: It supports all major HTTP methods like GET, POST, PUT, DELETE, etc., through simple method calls.
3. **Response Handling**: Responses from servers can be easily parsed, and the library provides convenient methods to extract useful information, such as `response.text`, `response.json()`, and `response.status_code`.

Slide 26: GET Request

This code snippet shows how to make an HTTP GET request to retrieve data from a specified URL using the `requests` library in Python.

```
url = "https://jsonplaceholder.typicode.com/posts/1"
headers = {}  # Empty headers dictionary
params = {}  # Empty parameters dictionary

response_get = requests.get(url, headers=headers, params=params)
```

1. The target URL, `https://jsonplaceholder.typicode.com/posts/1`, is stored in the variable `url`.
2. An empty dictionary named `headers` is created; in this case, no headers are being sent.
3. Another empty dictionary named `params` is created; Again, in this instance, no parameters are being sent.
4. The `requests.get()` method is called using the provided `url`, `headers`, and `params`. This sends an HTTP GET request to the specified URL. The response from this request (which will typically include the data from the server) is stored in the variable `response_get`.

Slide 27: Check Status Code

The snippet checks the status code of a response from a GET request. If the request was successful (status code 200), it prints a success message. Otherwise, it prints an error message with the received status code.

```
if response_get.status_code == 200:
    print("GET request was successful!")
else:
    print(f"Failed with status code: {response_get.status_code}")
```

1. `response_get.status_code`: This accesses the status code of the HTTP response that was stored in the response_get object. The status code is a numerical value indicating the result of the HTTP request.
2. `if response_get.status_code == 200:`: The condition checks if the status code is 200. An HTTP status code of 200 means "OK" and indicates that the request was successful.
3. `print("GET request was successful!")`: If the status code is indeed 200, this message will be printed, indicating a successful GET request.
4. `else:`: If the status code is anything other than 200, the code within this block will execute.
5. `print(f"Failed with status code: {response_get.status_code}")`: This prints an error message along with the actual status code that was received. This provides insight into what might have gone wrong, as different status codes represent different types of errors or statuses.

Slide 28: Check Response Content

This snippet first shows the raw content of the server's response and then displays a parsed version if the content is in JSON format.

```
 The 'text' attribute provides the response content as a string.
print("\nResponse Content:")
print(response_get.text)

# --- Parsing JSON Responses ---
# If the response is in JSON format, we can parse it into a Python dictionary.
data = response_get.json()
print("\nParsed JSON:")
print(data)
```

1. `response_get.text`: provides the raw content of the server's response as a string
2. `response_get.json()`: If the server's response is formatted in JSON, it can be converted into a Python dictionary.

Slide 29: Check Response Headers

This snippet shows how to access the response headers:

```
# Headers provide meta-information about the response or request.
print("\nResponse Headers:")
print(response_get.headers)

# Accessing a specific header value
content_type = response_get.headers['Content-Type']
print(f"Content Type: {content_type}")
```

1. `response_get.headers`: provides metadata about the response in the form of headers.
2. `response_get.headers['Content-Type']`: Extracts the `Content-Type` header's value from the response headers.

Slide 30: POST Request

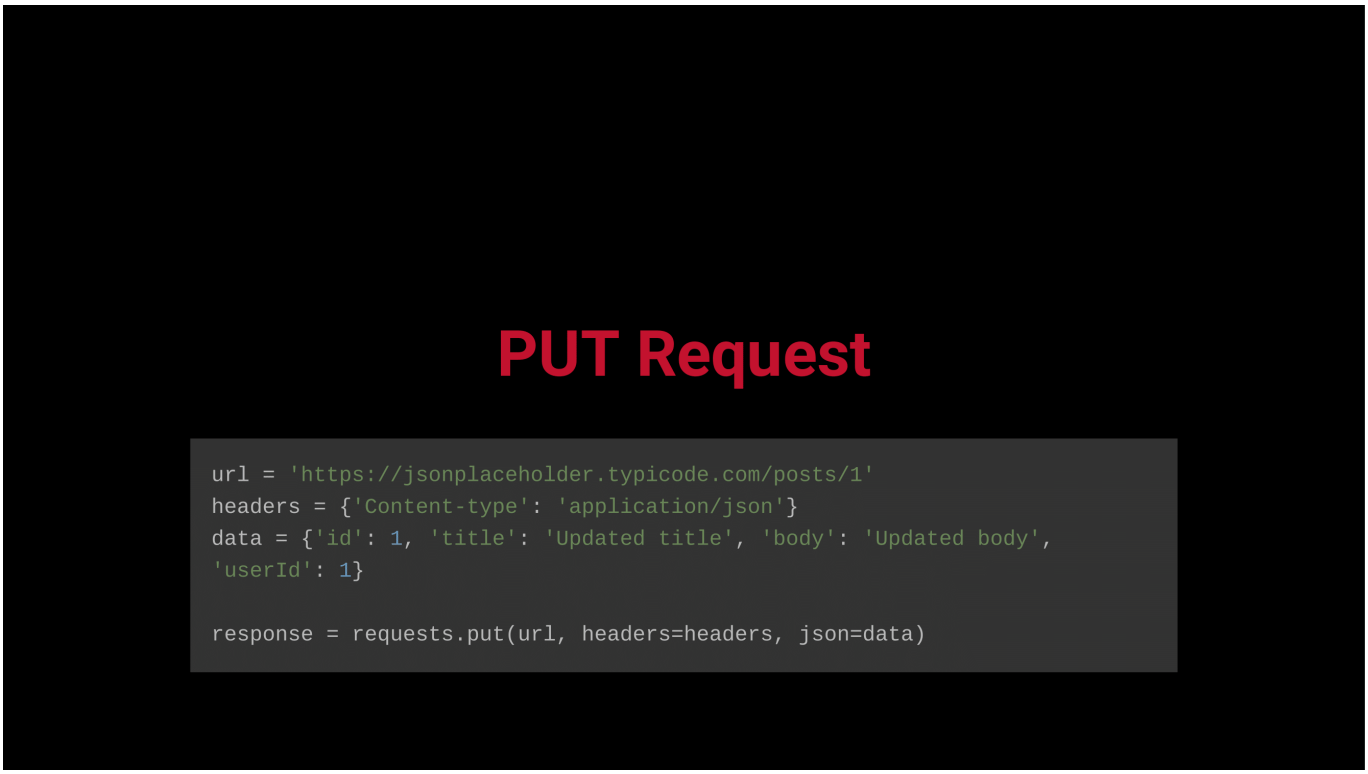
This snippet shows how to make a POST request using the `requests` library in Python.

```
url = 'https://jsonplaceholder.typicode.com/posts'
headers = {
    'Content-type': 'application/json; charset=UTF-8'  # Header for sending JSON
data
}
data = {
    'title': 'foo',
    'body': 'bar',
    'userId': 1
}

response_post = requests.post(url, headers=headers, json=data)
```

1. `url =`: Specifies the web address the data will be sent to as a string value containing the base URL and the endpoint.
2. `headers = {...}`: Defines a dictionary that contains information about the content type, telling the server that the content being sent is in JSON format.
3. `data = {...}`: Represents the content being sent. It's a dictionary containing a title, body, and a userId.
4. `response_post = requests.post(...): Sends a POST request to the URL, with the defined headers and data. The json=data argument converts the Python dictionary into a JSON string for the request. There are other methods to handle this conversion which we will see later!

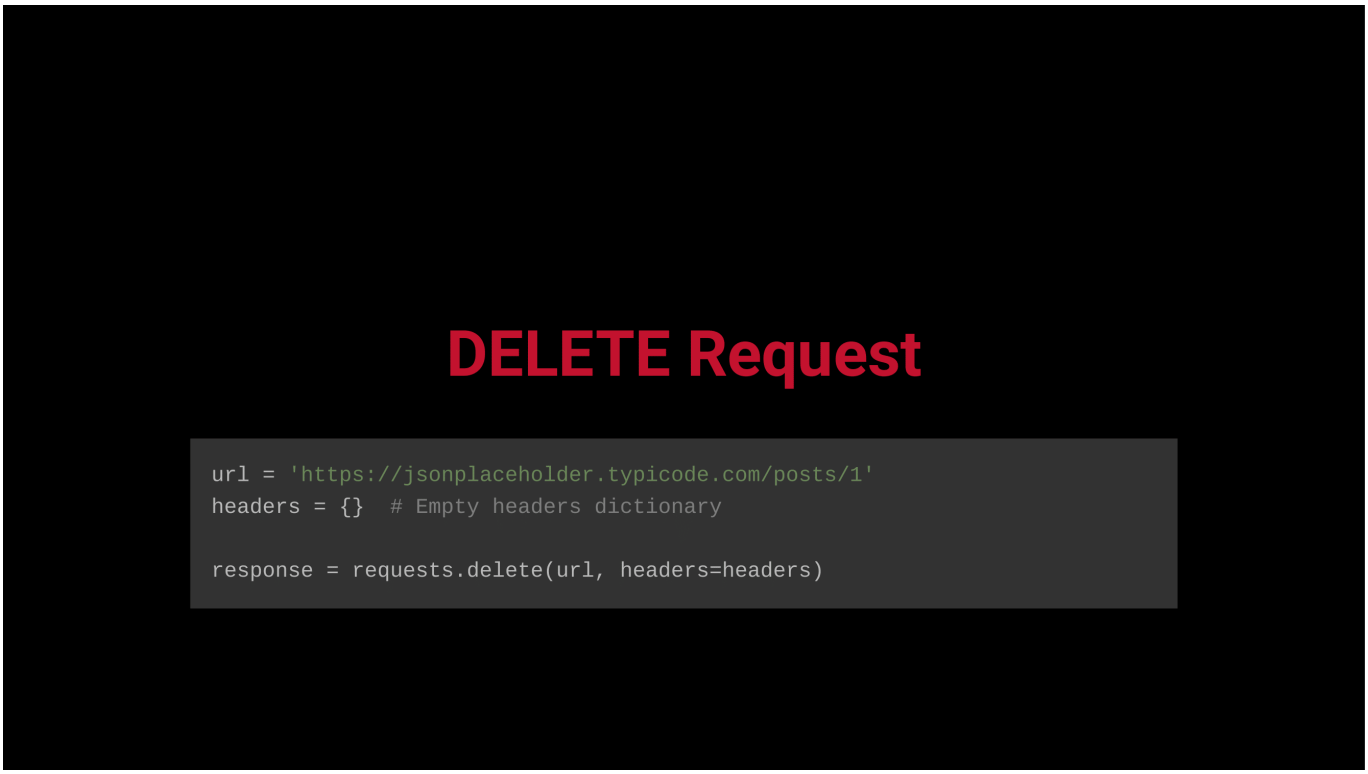
Slide 31: PUT Request



This snippet shows how to handle a PUT request in python

```
url = 'https://jsonplaceholder.typicode.com/posts/1'
headers = {
    'Content-type': 'application/json; charset=UTF-8'
}
data = {
    'id': 1,
    'title': 'Updated title',
    'body': 'Updated body',
    'userId': 1
}
```

Slide 32: DELETE Request



Lastly, this snippet shows how to create a DELETE request in Python.

```
url = 'https://jsonplaceholder.typicode.com/posts/1'
headers = {}  # Empty headers dictionary

response = requests.delete(url, headers=headers)
```

1. `headers = {}`: we still define a headers variable even though we don't need one. This just helps to maintain clarity and consistency, but we could have simply done the following and gotten the same result.

```
url = 'https://jsonplaceholder.typicode.com/posts/1'

response = requests.delete(url)
```

2. `requests.delete(...)`: execute a DELETE request with the `delete()` method.

Break (10 minutes)

Hands-On: Procore API in Python

▶ Click to Expand

Slide 34: Procore API in Python

During this part of the Hands-On Session we will be:

1. Compare Postman requests to requests crafted in Python
2. Craft Procore Requests Together
3. Craft Your Own Requests!

Slide 35: POST Access Token in Procore

This snippet shows you how to create and access token for your Procore App

```
client_id = os.getenv("CLIENT_ID")
client_secret = os.getenv("CLIENT_SECRET")

endpoint = "/oauth/token"

headers = {"Content-Type": "application/json"}

body = {
    "grant_type": "client_credentials",
    "client_id": client_id,
    "client_secret": client_secret
}

response = requests.post(
    url=f"{BASE_URL}{endpoint}",
    headers=headers,
    data=json.dumps(body)  # Convert the dictionary to a JSON string
)

access_token_data = response.json()
access_token = access_token_data["access_token"]
```

1. `client_id = os.getenv("CLIENT_ID")`: Retrieves the value of the environment variable "CLIENT_ID" and assigns it to the client_id variable.
2. `client_secret = os.getenv("CLIENT_SECRET")`: Retrieves the value of the environment variable "CLIENT_SECRET" and assigns it to the client_secret variable.
3. `endpoint = "/oauth/token"`: Specifies the endpoint for obtaining an OAuth token.
4. `headers = {...}`: Defines a dictionary that indicates the content being sent is in JSON format.
5. `body = {...}`: Constructs the data payload, including the grant type and the client's ID and secret credentials.
6. `response = requests.post(...)`: Sends a POST request to the composed URL (BASE_URL + endpoint), with the defined headers and the body converted to a JSON string.
7. `access_token_data = response.json()`: Parses the response, which is expected to be in JSON format, into a Python dictionary. 8 `access_token = access_token_data["access_token"]`: Extracts the value associated with the key "access_token" from the parsed response and assigns it to the access_token variable.

Slide 36: GET All Companies

This snippet gets the companies that the app has been downloaded to.

```
endpoint = "/rest/v1.0/companies"

headers = {"Authorization": f"Bearer {access_token}"}

response = requests.get(
    url=f"{BASE_URL}{endpoint}",
    headers=headers
)

company_data = response.json()

# save first (0th) "id" from the list which will correspond to RO
company_id = company_data[0]["id"]
```

1. `endpoint = "/rest/v1.0/companies"`: Specifies the endpoint for fetching company data.
2. `headers = {...}`: Defines a dictionary containing an Authorization header, which uses the previously retrieved access_token.
3. `response = requests.get(...)`: Sends a GET request to the composed URL (BASE_URL + endpoint), with the defined headers.
4. `company_data = response.json()`: Parses the response, which is expected to be in JSON format, into a Python dictionary or list (based on the response structure).
5. `company_id = company_data[0]["id"]`: Extracts the "id" of the first (0th) company in the retrieved data list and assigns it to the company_id variable.

Slide 37: GET All Projects within a Company

This snippet pulls all the projects that have granted access to your app.

```
endpoint = "/rest/v1.1/projects"

headers = {
    "Authorization": f"Bearer {access_token}",
    "Procore-Company-Id": f"{company_id}"
}

params = {
    "company_id": f"{company_id}"
}

response = requests.get(
    url=f"{BASE_URL}{endpoint}",
    headers=headers,
    params=params
)

project_data = response.json()
```

1. `endpoint = "/rest/v1.1/projects"`: Specifies the endpoint for fetching project data.
2. `headers = {...}`: Defines a dictionary for headers, including the previously retrieved access_token for authentication and specifying which company's projects to fetch.
3. `params = {"company_id": f"{company_id}"}`: Sets up query parameters to include the company_id in the request.
4. `response = requests.get(...)`: Sends a GET request to the composed URL (BASE_URL + endpoint), using the headers and parameters defined.

5. `project_data = response.json()`: Parses the response, expected to be in JSON format, into a Python dictionary or list (based on the response structure).

Slide 38: GET All Root Folders

This snippet gets all the folders and their children at the root of the Project's Documents directory

```
endpoint = "/rest/v1.0/folders"

headers = {
    "Authorization": f"Bearer {access_token}",
    "Procore-Company-Id": f"{company_id}"
}

params = {
    "project_id": "1668030" # hard-coded South Lamar project ID
}

response = requests.get(
    url=f"{BASE_URL}{endpoint}",
    headers=headers,
    params=params
)

folders_data = response.json()
```

1. `endpoint = "/rest/v1.0/folders"`: Specifies the endpoint for fetching folder data.
2. `headers = {...}`: Defines a dictionary for headers including the previously retrieved access_token for authentication and the company ID

3. `params = {"project_id": "1668030"}`: Sets up query parameters with a hard-coded `project_id` to specify which project's folders to fetch.
4. `esponse = requests.get(...)`: Sends a GET request to the composed URL (BASE_URL + endpoint), using the headers and parameters defined.
5. `folders_data = response.json()`: Parses the response, expected to be in JSON format, into a Python dictionary or list (depending on the response structure).

Slide 39: GET All Submittals

We will craft the Python request to get all submittals together.

Challenge: GET All RFIs (Slide 40)

Use the structure from the submittals and the documentation to list all of the RFIs.

- General Documentation
