Coursebook: Streamlit Web Dashboard

- Part 3 of Data Analytics for Lembaga Penjamin Simpanan (LPS)
- Course Length: 10.5 hours
- Last Updated: August 2024

- Developed by Algoritma's product division and instructors team

1 Background

1.1 Top-Down Approach

The coursebook is part of the **Data Science in Python Specialization** prepared by Algoritma. The coursebook is intended for a restricted audience only, i.e. the individuals and organizations having received this coursebook directly from the training organization. It may not be reproduced, distributed, translated or adapted in any form outside these individuals and organizations without permission.

Algoritma is a data science education center based in Jakarta. We organize workshops and training programs to help working professionals and students gain mastery in various data science sub-fields: data visualization, machine learning, data modeling, statistical inference etc.

1.2 Training Objectives

This coursebook is intended for participants who have completed the preceding courses offered in the **Data Science in Python Specialization**. This is the third course, **Streamlit Web Dashboard**

The coursebook focuses on:

Basic Python Programming for Dashboarding

- Using if-else statement
- Building loops using for
- Defining and calling functions.

Introduction to Streamlit

- Introduction to Streamlit for interactive web applications development
- Exploring Streamlit Features for building dynamic apps

Building Dashboard and Deployment

- Developing interactive visualization dashboards
- Deploying Streamlit apps for production
- Best practices for maintaining and updating dynamic dashboards

2 Basic Python Programming

In this section you will learn basic programming using python which is a provision for us in creating a web dashboard. In some parts you may have studied it, let's review that part.

We will start with objects and data structures in python.

2.1 Python Object and Data Structures

2.1.1 Number

To store numbers, python has two native data types called `int` and `float`.

- `int` is used to store integers (ie: 1,2,-3)
- `float` is used to store real numbers (ie: 0.7, -1.8, -1000.0)

Number Operations

Arithmetic Operators:

- `+` - Addition
- `-` - Subtraction
- `*` - Multiplication
- `/` - Division
- `//` - Round division
- `%` - Module
- `**` - Exponent

Comparison Operators:

- `<` - Less than (ie : a < b)
- `<=` - Less than or equal to (ie : a <= b)
- `>` - Greater than (ie: a > b)
- `>=` - Greater than or equal to (ie: a >= b)
- `==` - Equals (ie: a == b)
- `!=` - Not Equal (ie: a != b)

2.1.2 Strings

Strings are used in Python to record text information, such as names. Strings in Python are actually a sequence, which basically means Python keeps track of every element in the string as a sequence. For example, Python understands the string "hello' to be a sequence of letters in a specific order. This means we will be able to use indexing to grab particular letters (like the first letter, or the last letter).

Python represents any string as a `str` object. There are several ways to create a string value:

- using `''` (ie: "cyber punk 2077'")
- using `""` (ie : `"Hari Jum'at"`)
- using `'''` or `"""` (ie: `'''Andi berkata "Jum'at Bersih"'''`)

2.1.3 Boolean

Boolean stores a very simple value in computers and programming, `True` or `False`.

Boolean operations

Python provides logical operators such as:

- and (ie: a and b)

- or (ie: a or c)
- not (ie: not a)

2.1.4 List

Earlier when discussing strings we introduced the concept of a sequence in Python. Lists can be thought of the most general version of a sequence in Python. Unlike strings, they are mutable, meaning the elements inside a list can be changed!

Lists are constructed with brackets `[]` and commas separating every element in the list.

[1]:
```
my_list = [1,2,3]
```

[2]:
```
my_list = ['A string',23,100.232,'o']
```

Operation List

- `x.append(a)` : add a to x
- `x.remove(a)` : remove a from x

In addition to the previously known operators, one of the most useful lists is to implement an aggregation function such as:

- `len(x)` : extract the length of the list
- `a in b` : checks if the value `a` exists in the list object `b`
- `max(x)` : get the highest value in x
- `sum(x)` : get the number of values in x

Another operation to be aware of in lists is indexing:

- `x[i]` : access the i-th element of x

2.1.5 Dictionaries

We've been learning about sequences in Python but now we're going to switch gears and learn about mappings in Python. If you're familiar with other languages you can think of these Dictionaries as hash tables.

A Python dictionary consists of a **key** and then an associated **value**. That value can be almost any Python object.

Constructing a Dictionary

Let's see how we can construct dictionaries to get a better understanding of how they work!

[3]:
```
# Make a dictionary with {} and : to signify a key and a value
my_dict = {'key1':'value1',
           'key2':'value2'}
```

[4]:
```
# Call values by their key
my_dict['key2']
```

[4]:
```
'value2'
```

Its important to note that dictionaries are very flexible in the data types they can hold. For example:

[5]:
```
my_dict = {'key1':345,'key2':[34,45,56],'key3':['item3','item4','item5']}
```

[6]:
```
# Let's call items from the dictionary
my_dict['key3']
```

[6]:
```
['item3', 'item4', 'item5']
```

2.2 Python Statement

Instructions that a Python interpreter can execute are called statements. For example, `a = 1` is an assignment statement. `if` statement, `for` statement, etc.

if and else Statements

`if` Statements in Python allows us to tell the computer to perform alternative actions based on a certain set of results.

Verbally, we can imagine we are telling the computer:

"Hey if this case happens, perform some action"

We can then expand the idea further with `elif` and `else` statements, which allow us to tell the computer:

"Hey if this case happens, perform some action. Else, if another case happens, perform some other action. Else, if none of the above cases happened, perform this action."

Let's go ahead and look at the syntax format for if statements to get a better idea of this:

```
if case1:
    perform action1
elif case2:
    perform action2
else:
    perform action3
```

[7]:
```
if True:
    print('Yes, it was true')
```

```
Yes, it was true
```

[8]:
```
place = 'Algoritma'
if place == 'Algoritma':
    print('You are in Algoritma')
else:
    print('You are not in Algoritma')
```

```
You are in Algoritma
```

Notes:

Indentation is important to keep a good understanding of how indentation works in Python to maintain the structure and order of your code. We will touch on this topic again when we start building out functions!

for Loops

A `for` loop acts as an *iterator* in Python; it goes through items that are in a sequence or any other iterable item. Objects that we've learned about that we can iterate over include strings, lists, tuples, and even built-in iterables for dictionaries, such as keys or values.

We've already seen the `for` statement a little bit in past lectures but now let's formalize our understanding.

Here's the general format for a `for` loop in Python:

```
for item in object:
    statements to do stuff
```

The variable name used for the item is completely up to the coder, so use your best judgment for choosing a name that makes sense and you will be able to understand when revisiting your code. This item name can then be referenced inside your loop, for example if you wanted to use `if` statements to perform checks.

[9]:
```
my_list1 = [1,2,3,4,5,6,7,8,9,10]
```

[10]:
```
for num in my_list1:
    print(num)
```

```
1
2
3
4
5
6
7
8
9
10
```

We could have also put an `if else` statement in there:

[11]:
```
for num in my_list1:
    if num % 2 == 0:
        print(num)
    else:
        print('Odd number')
```

```
Odd number
2
Odd number
4
Odd number
```

```
6
Odd number
8
Odd number
10
```

2.2.1 Functions

A function is a useful device that groups together a set of statements so they can be run more than once. They can also let us specify parameters that can serve as inputs to the functions.

On a more fundamental level, functions allow us to not have to repeatedly write the same code again and again. If you remember back to the lessons on strings and lists, remember that we used a function len() to get the length of a string. Since checking the length of a sequence is a common task you would want to write a function that can do this repeatedly at command.

Why even use functions?

Put simply, you should use functions when you plan on using a block of code multiple times. The function will allow you to call the same block of code without having to write it multiple times. This in turn will allow you to create more complex Python scripts. To really understand this though, we should actually write our own functions!

Creating a function

In Python a function is defined using the `def` keyword, and follow by function name.

[12]:
```
def my_function():
  print("Hello from a function")
```

Calling a function

To call a function, use the function name followed by parenthesis:

[13]:
```
my_function()
```

```
Hello from a function
```

Arguments

Information can be passed into functions as arguments.

Arguments are specified after the function name, inside the parentheses. You can add as many arguments as you want, just separate them with a comma.

The following example has a function with one argument (`name`). When the function is called, we pass along a first name, which is used inside the function to print the full name:

[14]:
```
def my_function(name):
  print(name + " from Algoritma")

my_function('Dwi')
my_function('Handoyo')
my_function('Wulan')
```

```
Dwi from Algoritma
Handoyo from Algoritma
Wulan from Algoritma
```

Using return

So far we've only seen `print()` used, but if we actually want to save the resulting variable we need to use the **return** keyword.

Let's see some example that use a `return` statement. `return` allows a function to *return* a result that can then be stored as a variable, or used in whatever manner a user wants.

[15]:
```
def area(width,length):
    return width*length
```

[16]:
```
area(4,5)
```

[16]: 20

A Very Common Question: "What is the difference between `return` and `print`?"

The `return` keyword allows you to actually save the result of the output of a function as a variable. The `print()` function simply displays the output to you, but doesn't save it for future use. Let's explore this in more detail

[17]:
```
def print_result(a,b):
    print(a+b)
```

[18]:
```
def return_result(a,b):
    return a+b
```

[19]:
```
print_result(10,5)
```

```
15
```

[20]:
```
# You won't see any output if you run this in a .py script
return_result(10,5)
```

[20]: 15

But what happens if we actually want to save this result for later use?

[21]:
```
my_result = print_result(20,20)
```

```
40
```

[22]:
```
my_result
```

[23]:
```
type(my_result)
```

[23]: NoneType

Be careful! Notice how `print_result()` doesn't let you actually save the result to a variable! It only prints it out, with `print()` returning `None` for the assignment!

3 Introduction to Streamlit

Challenge for Data Scientists

As a data scientist, you often face the challenge of presenting data analysis results in a way that is easily understood by various stakeholders. The large amounts of data processed and the insights you want to share require an effective and efficient medium. However, developing web applications from scratch to display visualizations and data analysis requires time and specialized web development skills that not all data scientists possess.

Solution

Streamlit emerges as a solution to these challenges. With Streamlit, you can quickly and easily create interactive web applications using only Python. Streamlit allows you to focus on the logic of data analysis without worrying about the technical details of web development.

3.1 What is Streamlit

Streamlit is a Python framework used to easily create interactive web applications. With Streamlit, users can develop web applications without requiring deep knowledge of traditional web development.

3.2 Advantages

1. **Ease of Use**: Streamlit is very easy to use and does not require deep knowledge of HTML, CSS, or JavaScript. You only need to write Python code.
2. **Rapid Development**: With Streamlit, you can create web applications in minutes. Changes to the code can be immediately seen in the running application.
3. **High Interactivity**: Streamlit allows for the creation of applications with interactive controls such as sliders, buttons, and text inputs very easily.
4. **Easy Integration**: Streamlit supports integration with various commonly used Python libraries in data science such as Pandas, Matplotlib, and Plotly.
5. **Good Community and Documentation**: Streamlit has comprehensive documentation and an active community, making it easy to find help and inspiration.

4 Get Started

4.1 Tools Preparation

To use Streamlit, you need to have a few tools set up on your computer. These include:

1. **Code Editor**: A good code editor like Visual Studio Code (VSCode) is essential for writing and managing your Streamlit applications. VSCode offers many features such as syntax highlighting, code completion, and extensions that can enhance your development experience.
2. **Python**: Streamlit is built on Python, so you need to have Python installed on your computer. It's recommended to use the latest stable version of Python. You can download Python from the official Python website.

3. **Required Python Libraries**: Apart from Streamlit, you might need other Python libraries for your data processing and visualization tasks. Common libraries include Pandas for data manipulation, Matplotlib and Plotly for data visualization, and NumPy for numerical operations.

4. **Virtual Environment**: Using a virtual environment is a good practice for managing project dependencies separately from your global Python installation. This helps avoid conflicts between different projects and ensures that your application runs in a consistent environment. You can create a virtual environment using **conda evironment**

4.2 Create New Project

After installing all the necessary libraries in your working environment, the next step is to create a new Streamlit project. This is done by creating a new Python script file. For example, let's create a new file named `app.py`.

4.2.1 Steps to Create a New Project

1. **Open your code editor**: Launch Visual Studio Code or your preferred code editor.

2. **Create a new file**: In your project directory, create a new file and name it `app.py`.

3. **Add Streamlit import statement**: Open the `app.py` file and start by importing Streamlit:

   ```
   import streamlit as st
   ```

4. **Write your first Streamlit code**: Add a simple Streamlit command to display a title on the web app:

   ```
   st.title('Hello, Streamlit!')
   ```

Your `app.py` file should now look like this:

```
import streamlit as st

st.title('Hello, Streamlit!')
```

4.3 Run App

Once you have added the `import streamlit` command to your new file, you can run your Streamlit application using the following command in the terminal:

```
streamlit run app.py
```

4.3.1 Steps to Run Your Streamlit App

1. **Open the terminal**: Open your terminal or command prompt.

2. **Navigate to your project directory**: Ensure you are in the directory where your `app.py` file is located.

3. **Activate the virtual environment**: If you have created a virtual environment, activate it.
   `sh conda activate env_name`

4. **Run the Streamlit app**: Execute the command to run your Streamlit application:

```
streamlit run app.py
```

This command will start the Streamlit server and open your default web browser to display the application. You should see the title "Hello, Streamlit!" displayed on the page.

4.4 Features

Streamlit offers a variety of features that make it easy to build interactive and dynamic web applications. In this section, we'll explore some of the key features of Streamlit and how to use them.

4.4.1 Write

`st.write()` The `st.write()` function is a versatile command that allows you to display various types of content in your Streamlit app. You can use it to display text, data, and even custom objects.

Example:

```
st.write("Hello, Streamlit!")
st.write(1234)
st.write({"key": "value"})
```

4.4.2 Text Elements

Streamlit provides several functions to display text in different formats. These functions help you structure your content and make it visually appealing.

`st.markdown` Displays text formatted using Markdown. This is useful for adding styled text, links, and other elements.

Example:

```
st.markdown("# This is a Markdown header")
st.markdown("**Bold text** and _italic text_")
```

`st.title` Displays a title in your Streamlit app.

Example:

```
st.title("This is a Title")
```

`st.header` Displays a header in your Streamlit app.

Example:

```
st.header("This is a Header")
```

`st.subheader` Displays a subheader in your Streamlit app.

Example:

```
st.subheader("This is a Subheader")
```

4.4.3 Data Elements

Streamlit makes it easy to display data in tabular formats, making it easier to visualize and analyze data directly in your app.

st.dataframe Displays a Pandas DataFrame as an interactive table.

Example:

```
import pandas as pd

data = {'Column 1': [1, 2, 3], 'Column 2': [4, 5, 6]}
df = pd.DataFrame(data)
st.dataframe(df)
```

st.table Displays a static table.

Example:

```
st.table(df)
```

4.4.4 Layouts and Containers

Streamlit allows you to organize your app's layout using various layout and container elements. These elements help you create a more structured and user-friendly interface.

st.columns Creates a multi-column layout.

Example:

```
col1, col2, col3 = st.columns(3)
col1.write("Column 1")
col2.write("Column 2")
col3.write("Column 3")
```

st.container Creates a container that can hold multiple elements.

Example:

```
with st.container():
    st.write("This is inside a container")
```

st.expander Creates an expandable/collapsible container.

Example:

```
with st.expander("Expand to see more"):
    st.write("This is inside an expander")
```

st.sidebar Adds elements to the sidebar.

Example:

```
st.sidebar.write("This is in the sidebar")
```

st.tabs Creates a tabbed layout.

Example:

```
tab1, tab2 = st.tabs(["Tab 1", "Tab 2"])
tab1.write("This is Tab 1")
tab2.write("This is Tab 2")
```

4.4.5 Chart Element

Streamlit supports displaying charts and graphs using various plotting libraries.

st.plotly_chart Displays a Plotly chart.

Example:

```
import plotly.express as px

df = px.data.iris()
fig = px.scatter(df, x='sepal_width', y='sepal_length', color='species')
st.plotly_chart(fig)
```

4.5 Input Widgets

Streamlit allows us to create interactive dashboards by providing a variety of input widgets such as buttons, sliders, text inputs, file uploaders, and more.

Implementing these widgets requires an understanding of their syntax and logic, as their usage will affect the interaction between the user and the application we build. For example, a button will execute a specific action when pressed, while a slider will allow users to select a desired value within a specified range.

Here are some commonly used input widgets and their purposes:

4.5.1 Button Elements

- **st.button**
  - Adds a button that can be pressed to execute an action.

```
import streamlit as st

# Example of using st.button
st.button("Reset", type="primary")
if st.button("Say hello"):
    st.write("Why hello there")
else:
    st.write("Goodbye")
```

4.5.2 Slider Elements

- **st.slider**

- Adds a slider that allows users to select a value from a range.

```
import streamlit as st

# Example of using st.slider
age = st.slider("Select your age", 0, 100, 25)
st.write("Your age is:", age)
```

4.5.3 Text Input Elements

- **st.text_input**
  - Adds a text input box for users to type in text.

```
import streamlit as st

# Example of using st.text_input
name = st.text_input("Enter your name")
st.write("Hello,", name)
```

4.5.4 File Upload Elements

- **st.file_uploader**
  - Adds a file uploader that allows users to upload files.

```
import streamlit as st

# Example of using st.file_uploader
uploaded_file = st.file_uploader("Choose a file")
if uploaded_file is not None:
    st.write("File uploaded successfully")
```

4.5.5 Select Box Elements

- **st.selectbox**
  - Adds a dropdown select box for users to choose from a list of options.

```
import streamlit as st

# Example of using st.selectbox
option = st.selectbox("Choose an option", ["Option 1", "Option 2", "Option 3"])
st.write("You selected:", option)
```

4.5.6 Multi-select Elements

- **st.multiselect**
  - Adds a multi-select box that allows users to select multiple options.

```
import streamlit as st

# Example of using st.multiselect
```

```
options = st.multiselect("Select multiple options", ["Option 1", "Option 2", "Option 3"])
st.write("You selected:", options)
```

4.5.7 Radio Button Elements

- **st.radio**
  - Adds a set of radio buttons for users to select a single option.

```
import streamlit as st

# Example of using st.radio
radio_option = st.radio("Choose one", ["Option 1", "Option 2", "Option 3"])
st.write("You selected:", radio_option)
```

4.6 Configuration

Streamlit provides configuration features that help us customize the behavior and appearance of the web applications we create. One useful configuration function is:

4.6.1 st.set_page_config

st.set_page_config is used to set the page configuration for a Streamlit app. This function allows you to adjust various parameters, including:

1. `page_title`
   - **Description**: The title of the page that will appear in the browser tab.
   - **Type**: String
   - **Example**:
     `st.set_page_config(page_title="My Streamlit App")`
2. `page_icon`
   - **Description**: The URL or emoji for the favicon displayed in the browser tab.
   - **Type**: String (URL or Emoji)
   - **Example**:
     `st.set_page_config(page_icon=" ")`
3. `layout`
   - **Description**: The layout type of the page, either "centered" or "wide".
   - **Type**: String
   - **Example**:
     `st.set_page_config(layout="wide")`
4. `initial_sidebar_state`
   - **Description**: The initial state of the sidebar, which can be "auto", "expanded", or "collapsed".
   - **Type**: String
   - **Example**:
     `st.set_page_config(initial_sidebar_state="expanded")`
5. `menu_items`
   - **Description**: A dictionary of menu items to add to the sidebar of the app.
   - **Type**: Dictionary
   - **Example**:

```
st.set_page_config(menu_items={"About": "#", "Help": "#"})
```

4.6.2 Example Usage

Here's an example of how to use `st.set_page_config` in your Streamlit app:

```
import streamlit as st

st.set_page_config(
    page_title="My Awesome App",
    page_icon=" ",
    layout="wide",
    initial_sidebar_state="collapsed",
    menu_items={
        "About": "This is a demo app",
        "Help": "Help page URL"
    }
)

st.title("Welcome to My Awesome App!")
```

This configuration sets the page title to "My Awesome App", uses a rocket emoji for the favicon, sets the layout to wide, collapses the sidebar by default, and adds custom menu items.

5 Deploy

Once we have created a dashboard with features that meet our needs, we naturally want it to be accessible publicly. This means our web dashboard will no longer run locally but on a cloud server. The process of moving the application from a local environment to production or live is known as deployment.

5.1 Deployment Concept

Essentially, the deployment process involves transferring all the necessary components to a server to continuously display and run our application. We need to provide a server environment that can effectively run our application. Here are some key aspects to prepare:

1. **Server or Cloud Provider**: A service provider that will host and run your application. Examples include Heroku, AWS, Google Cloud, or Streamlit Cloud.
2. **Dependencies**: All libraries and packages used in your application must be installed on the server. This includes creating a file like `requirements.txt` to specify these dependencies.
3. **Configuration Files**: Files that describe the application's setup and dependencies. For Python projects, this typically includes `requirements.txt` for Python packages and other configuration files specific to your deployment environment.
4. **Environment Variables**: Environment variables used to store sensitive information such as API keys, database URLs, and other configurations. These should be set on the server and not hardcoded in your application code.

5. **Version Control**: Using a version control system like Git to track changes in your codebase and facilitate the deployment process. This helps in managing different versions of your application and simplifies collaboration.

5.2 Deployment Methods

There are several methods for deploying Streamlit applications:

- **Streamlit Community Cloud**: A free service provided by Streamlit specifically for hosting Streamlit applications. It offers an easy setup and deployment process with a user-friendly interface.
- **Heroku**: A cloud platform that supports a wide range of programming languages and frameworks, including Streamlit. Heroku provides a straightforward deployment process with scalable options.
- **Docker**: Containerization technology that allows applications to run in isolated environments called containers. Docker helps ensure consistency across different environments and simplifies deployment.
- **AWS, Azure, and GCP**: Major cloud providers that offer extensive support for deploying applications with high flexibility and scalability. Each platform has its own set of tools and services to manage, scale, and monitor your Streamlit application.

5.3 Example Deployment of Streamlit on Streamlit Community Cloud

Here are the steps to deploy a Streamlit application on Streamlit Community Cloud:

1. **Push Code to GitHub**:
   - Create a new repository on GitHub and push your Streamlit application code to that repository.
     ```
     your-repository/
        your_app.py
        requirements.txt
     ```
   - Do not push any secret files such as `secret.toml`, `.env`, etc.
2. **Login to Streamlit Community Cloud**:
   - Go to Streamlit Community Cloud and log in using your GitHub account.
3. **Create App**:
   - Click on "Create app".
   - Select the deployment option from GitHub repository.
   - Tip: Use the *"Paste GitHub URL"* menu and paste the URL of your Python file in GitHub.
4. **Advance Settings**:
   - Choose the "Advanced settings" section.
   - Select the Python version that matches the version used in your local environment.
5. **Deploy Application**:
   - Set the desired URL for your app.
   - Click "Deploy!".
6. **Monitor and Update**:
   - After deployment is complete, you can access your application through the provided URL.

- To update the application, simply push changes to your GitHub repository, and the application will automatically redeploy.