# Create, Import, and Install Libraries

Kittikun Jitpairod

## Library

`A library in programming is a collection of pre-written code` that provides reusable functionality for common tasks. Libraries extend a programming language's capabilities, allowing developers to utilize well-tested code for various functions without writing everything from scratch. This approach enhances productivity and enables the implementation of complex features more efficiently.

Libraries in Python can be categorized into three main types:

1. `Internal (Standard) Libraries`: These are included with Python's standard distribution. They provide core functionality and are readily available without additional installation. Examples include 'math', 'datetime', and 'os'.
2. `Local Libraries`: These are custom modules or packages created by developers for a specific project or organization. They are typically stored in the project directory or a designated location on the local machine. Local libraries allow for code reuse within a project or across related projects within an organization.
3. `External Libraries`: These are third-party libraries developed independently of the Python core. They offer specialized functionality and require separate installation. Popular examples include 'numpy' for numerical computing, 'pandas' for data analysis, and 'matplotlib' for plotting.

## Creating and Importing Local Library

As your Python projects grow, you'll often want to organize your code into separate files or modules. Let's explore how to create a local library of functions and import them into your main code.

1. First, let's create a new file called `my_functions.py` in the same directory as our main script. This will be our local library.

```python
def calculate_area(length, width):
    """
    Calculate the area of a rectangle.
    """
    return length * width

PI = 3.14159
def calculate_circle_area(radius):
    """
    Calculate the area of a circle.
    """
    return PI * radius ** 2
```

This file defines three functions and a constant that we can use in other parts of our project.

2. Importing functions to your script licated in the same directory:

a. Import the entire module

```python
import my_functions

# Use the functions
print(my_functions.calculate_area(5, 3))
print(my_functions.calculate_circle_area(2))
```

```
15
12.56636
```

When you run `main.py`, it will import `my_functions.py` and you can use its functions by prefixing them with `my_functions.`.

b. If you only need certain functions, you can import them specifically:

```python
from my_functions import calculate_area, calculate_circle_area

print(calculate_area(4, 6))
print(calculate_circle_area(4))
```

```
24
50.26544
```

c. You can also use aliases to avoid naming conflicts or simply for convenience:

```python
from my_functions import calculate_circle_area as circle_area

print(circle_area(3))
```

```
28.27431
```

d. Importing Everything (Use Cautiously)

You can import all `functions` and `variables` from a module using `*`, but this is generally discouraged as it can lead to naming conflicts and make it unclear where things are coming from:

```python
from my_functions import *

print(calculate_area(2, 8))
print(calculate_circle_area(5))
print(PI)
```

```
16
78.53975
3.14159
```

## Installing External Libraries

While Python's standard library provides a robust set of tools for many programming tasks, external libraries often offer additional functionality that can significantly enhance a programmer's capabilities. This section will explore the process of installing and utilizing external libraries in Python.

### Package Managers

Python utilizes package managers to facilitate the installation and management of external libraries. The two primary package managers are:

1. `pip`: The default package installer for Python
2. `conda`: An open-source package management system and environment management system

This discussion will focus on pip, as it is included with Python installations from version 3.4 onward.

### Virtual Environments

When working on multiple projects, it is often beneficial to create isolated Python environments. This practice prevents conflicts between library versions and allows for project-specific dependencies. The `venv` module, included in Python 3.3 and later, provides this functionality.

To create a virtual environment:

```
python -m venv myenv
```

To activate the virtual environment:

- On Windows: `myenv\Scripts\activate`
- On Unix or MacOS: `source myenv/bin/activate`

Once activated, any libraries installed will be specific to this environment.

### Using `pip` to Install Libraries

The basic syntax for installing a library using pip is as follows:

```
pip install library_name
```

For example, to install the popular data analysis library pandas, one would execute:

```
pip install pandas
```

It is also possible to install a specific version of a library:

```
pip install pandas==1.2.0
```

### Requirements Files

For projects with multiple dependencies, it is common practice to list all required libraries in a `requirements.txt` file. This file typically contains a list of libraries and their versions:

```
pandas==1.2.0
numpy==1.19.5
matplotlib==3.3.3
```

To install all libraries listed in a requirements file:

```
pip install -r requirements.txt
```

Once installed, external libraries can be imported and used in Python scripts. For example:

```python
import pandas as pd

data = pd.read_csv('data.csv')
print(data.head())
```