AI Boot Camp

Programming with Functions — Part 2

Module 3 Day 2

Class Objectives

By the end of class, you will be able to:

1. Import and use external Python modules and functions.
2. Refactor complex code into functional units.
3. Understand how to create software applications based on business and user needs.
4. Write and implement your own Python modules.

Instructor **Demonstration**

Importing Modules, Functions, and Methods

Modular Programming

Modular programming involves breaking down large, complex programming tasks using smaller “building blocks”, called modules.

The advantages of using modules include:

Reusability

Modules are able to be called and used over and over again without the need to duplicate the code.

Simplicity and organization

The use of modules helps to organize otherwise cumbersome code into smaller blocks, each of which accomplishes a specific task. Simplicity makes debugging errors easier too.

Maintainability

It is best practice for programmers to write modules with as few dependencies on other modules as possible, since doing so reduces the risk that a change to one module will have a knock-on effect on other modules. This is especially important when collaborating on code.

Scoping

Programmers have to carefully consider which modules will have access to any variables in the code. By using modules, any variables defined inside those modules are only accessible locally which reduces the risk of naming conflict in the larger program.

Importing Modules and Functions

Module contents are made available to the caller with the import statement. The import statement takes many different forms, shown below:

- The statement `import <module_name>` only allows the module to be imported to the programming file. All the functions and methods in the module can be accessed.

Python

```
import <module_name>
```

- An alternate form of the import statement allows individual objects from the module to be imported directly into the programming file.

Python

```
from <module_name> import <function_name>
(or <method_name>)
```

Using the Import Function

The Python **`math`** module contains many methods and constants that can be used to perform mathematical tasks.

```
# Import the sqrt function from the math module.
from math import sqrt

# Calculate the square root of a number.
number = 16
result = sqrt(number)
print(f"The square root of {number} is {result}")
```

The output from running the code is:

```
The square root of 16 is 4.0
```

Using the Import Function

The Python `random` module contains methods that allow you to work with random numbers. We can import specific methods from the `random` module as follows:

```
# Import the randint and choice methods from the
random module.
from random import randint, choice

# Generate a random number between 1 and 10 and
select a random element from a list.
random_number = randint(1, 10)
print(f"The random number is: {random_number}")

my_list = ['apple', 'banana', 'orange', 'grape',
'mango']

# Use the choice method to randomly select an element
from the list.
random_element = choice(my_list)
print(f"A random element from the list is:
{random_element}")
```

The output from running the code is:

```
The random number is: 9
A random element from the list is: banana
```

Using the Import Function

The Python `datetime` module contains **classes** that can be used to manipulate dates and times. We can import the datetime and date classes to print the current datetime, time, and date.

```
# Import the datetime and date classes from the
datetime module.
from datetime import datetime, date

# Get the current datetime using the now function.
current_datetime = datetime.now()
# Get the current time using the strftime function.
current_time = current_datetime.strftime("%H:%M:%S")
# Get the current date using the today function.
current_date = date.today()

print(f"The current datetime is: {current_datetime}")
print(f"The current time is: {current_time}")
print(f"The current date is: {current_date}")
```

The output from running the code is:

```
The current time is: 2023-07-07 10:57:43.724947
The current time is: 10:57:43
The current date is: 2023-07-07
```

Activity:

Importing Car Loan Function

In this activity, you'll import and use a function from a Python file to calculate the future value of a car loan.

Suggested Time:
15 Minutes

Time’s up!

Let’s review

Questions?

Instructor **Demonstration**

Refactoring Code Best Practices

What is Refactoring?

Refactoring is the process of improving the internal structure of code without changing its external behavior. It involves making small changes to the code that enhance its readability, maintainability, and performance without affecting how the code functions while in use.

Some advantages of refactoring are:

1. It improves the design of software applications.
2. It makes code more easily understandable.
3. It makes it easier for team members to understand your code and changes you've made.
4. It broadens your knowledge of the code and its role in the application.

Some Refactoring Best Practices

01

Refactoring is not bug fixing. Refactoring should occur after bugs are fixed.

- However, if part of the code causes unpredictable errors despite trying to fix the problem then refactoring should be considered.

02

Make sure to refactor code that is already working and has been tested.

- If the code has lots of bugs, refactoring may cause more problems.

03

Minimize the amount of refactoring.

- Make small changes and test if the behavior of the code is the same, repeating the process if necessary.

04

Avoid adding new features and functionality until you are done refactoring.

- Refactored code shouldn't change the behavior of the code

Common Tips for Refactoring Python Code

Use the `enumerate()` function instead of the `range()` function.

```
# Code that uses the range() function.
numbers = [10, 20, 30, 40, 50]
for i in range(len(numbers)):
    print(f"Index: {i}, Value:
{numbers[i]}")
```

```
# Refactored the code to use
enumerate()
numbers = [10, 20, 30, 40, 50]
for i, num in enumerate(numbers):
    print(f"Index: {i}, Value: {num}")
```

Common Tips for Refactoring Python Code

Use a function instead of long code blocks and repetitive tasks.

```
# Code without a function.
numbers = [5, 10, 15, 20, 25]
total = 0
count = 0
for num in numbers:
    total += num
    count += 1
average = total / count
print(f"The average is: {average}")
```

```
# Refactored code with a function
def calculate_average(numbers):
    """The function calculates the
average of an array of numbers."""
    total = sum(numbers)
    count = len(numbers)
    average = total / count
    return average

numbers = [5, 10, 15, 20, 25]
average = calculate_average(numbers)
print(f"The average is: {average}")
```

Activity:

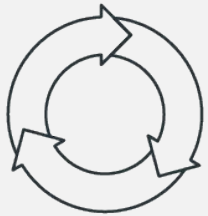
Refactoring Travel Loops

In this activity, you'll code along with the instructor who will prompt you at certain points on what the next step in refactoring the example should be. These probing questions combined with the whole group's participation in the demonstration is meant to reinforce the concept of refactoring.

Suggested Time:
20 Minutes

Instructor **Demonstration**

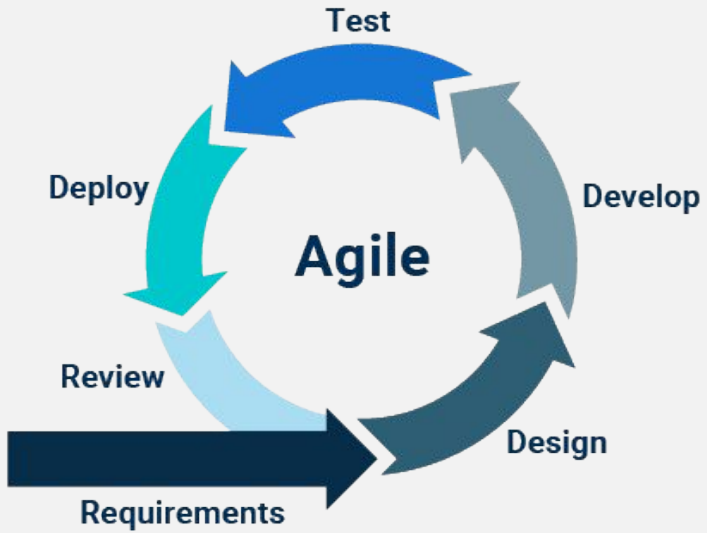
Working with User Stories and Business Requirements

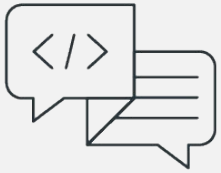


Agile **Methodology**

Agile Methodology

To improve the speed and quality of the software development process, a style of project management was created that focused on cross-team collaboration and feedback.





User **Stories**

User Stories

A **user story** is a short statement that describes how a type of user needs to interact with a feature in a program.

- We write user stories in plain language. This is so that both the technical and the non-technical stakeholders in the software development can understand what the software should be able to do.
- User stories often adhere to the following three-part template:

> ***As a [type of user], I want [some goal] so that [some reason].***

Understanding the Importance **of Business and User Needs**

Understanding the Importance of Business Needs and Requirements

Even a task that seems simple can involve many considerations, which are also known as **business requirements** or **software application requirements**. These requirements specify how to build a program to meet the **business need**—the reason your company is paying you to build the software.

Before you begin to develop an application, it's critical that you understand two things:

1. The purpose that it’s intended to serve (the business need)
2. How it should function to properly meet that need (the business requirements)

Identify Business Needs **and Business Requirements**

Activity:
ATM User Stories

In this activity, you'll work in a group to analyze a set of user stories in order to plan a Python program that meets user needs.

Suggested Time:
10 Minutes

Time’s up!

Let’s review

Break

15 mins

Instructor **Demonstration**

ATM Application Logic

ATM Application Logic

Create code that meets the requirements:

- Create a login function that will take a user's PIN as an argument.
- The function should validate the PIN agains the provided list of accounts.
- If the PIN is valid, the function should return the account balance.

```
# Define the `login` function for the ATM application.
def login(pin):
    """Create a login function for the ATM application.
    Args:
        pin (integer): The users pin number

    Returns:
        If the pin matches one of the pin numbers in the
"accounts"
        the account balance is returned.

    Notes:
        Create a for loop to check to validate the PIN
against this list of `accounts`.
        If the PIN is validated, print the account's
balance formatted to two decimal places and thousandths.
    """
    for account in accounts:
        if int(pin) == account["pin"]:
            print(f"The account balance for PIN
{account['pin']} is: ${account['balance']: ,.2f}.")

if __name__ == "__main__":
    # Set the function call equal to a variable called
account_balance.
    account_balance = login(246802)
```

Activity:

ATM Application Code

In this activity, you'll write the business logic and code associated with three ATM functions.

Suggested Time:
15 Minutes

Time’s up!

Let’s review

Instructor **Demonstration**

A Modular ATM Design

A Modular ATM Design

Modularizing a codebase splits the code into several files for two main reasons:

01 It's easier to maintain.

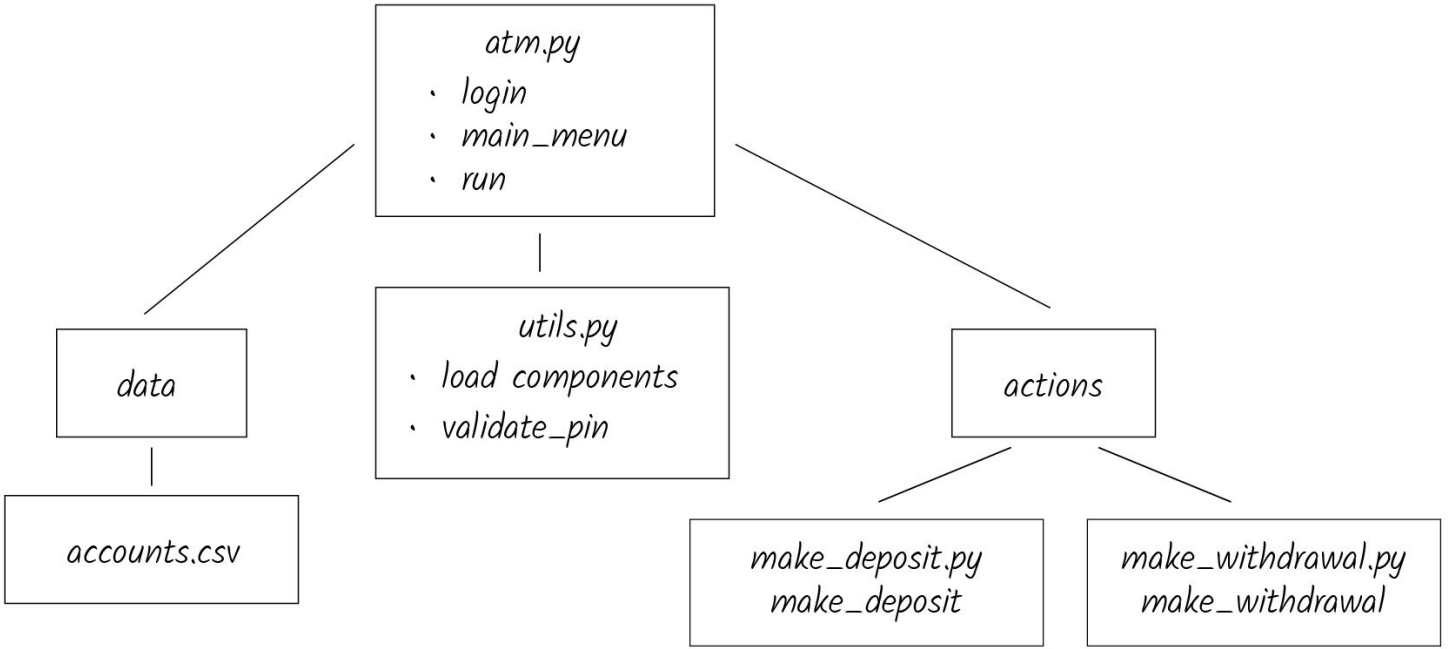
02 It allows us to avoid rewriting a particular piece of common logic multiple times.

Think about the activity you just completed:

- What is the starting point of the application?
- Can we group some of the code by functionality?
- Is any code shared across the codebase?

ATM Application Structure

Use the following working diagram to create the application:



Activity:

ATM Modularization

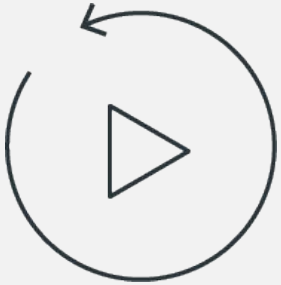
In this activity, you'll modularize the ATM application so that the codebase matches the following layout of the "atm" folder and its files and subfolders:

```
atm (main folder)
├── actions (subfolder)
│   ├── make_deposit.py
│   └── make_withdrawal.py
├── data (subfolder)
│   └── accounts.csv
├── modular_atm.py
└── utils.py
```

Suggested Time:
15 Minutes

Time’s up!

Let’s review



Let’s **recap**

Review the Class Objectives

In this lesson you learned how to:

1. Import and use external Python modules and functions.
2. Refactor complex code into functional units.
3. Understand how to create software applications based on business and user needs.
4. Write and implement your own Python modules.

Next

In the next lesson, you'll learn to make code better by diving into the world of object-oriented programming (OOP), a programming paradigm that uses objects and classes to structure and organize code.

Questions?

The End