




Welcome to the CoGrammar Modules

The session will start shortly...

Questions? Drop them in the chat. We'll have dedicated moderators answering questions.



Software Engineering Session Housekeeping

- The use of disrespectful language is prohibited in the questions, this is a supportive, learning environment for all - please engage accordingly.
(Fundamental British Values: Mutual Respect and Tolerance)
- No question is daft or silly - **ask them!**
- There are **Q&A sessions** midway and at the end of the session, should you wish to ask any follow-up questions. Moderators are going to be answering questions as the session progresses as well.
- If you have any questions outside of this lecture, or that are not answered during this lecture, please do submit these for upcoming Academic Sessions. You can submit these questions here: [Questions](#)

Software Engineering Session Housekeeping cont.

- For all **non-academic questions**, please submit a query:
www.hyperiondev.com/support
- Report a **safeguarding** incident:
www.hyperiondev.com/safeguardreporting
- We would love your **feedback** on lectures: [Feedback on Lectures](#)



Skills Bootcamp 8-Week Progression Overview

Fulfil 4 Criteria to Graduation



Criterion 1: Initial Requirements

- **Timeframe:** First 2 Weeks
- **Guided Learning Hours (GLH):**
Minimum of 15 hours
- **Task Completion:** First four tasks



Criterion 2: Mid-Course Progress

- **Guided Learning Hours (GLH):** 60
- **Task Completion:** 13 tasks

Skills Bootcamp Progression Overview

✓ Criterion 3: Course Progress

- **Completion:** All mandatory tasks, including Build Your Brand and resubmissions by study period end
- **Interview Invitation:** Within 4 weeks post-course
- **Guided Learning Hours:** Minimum of 112 hours by support end date (10.5 hours average, each week)

✓ Criterion 4: Demonstrating Employability

- **Final Job or Apprenticeship Outcome:** Document within 12 weeks post-graduation
- **Relevance:** Progression to employment or related opportunity

CoGrammar Modules

May 2024

Learning Objectives

- Define the purpose and importance of Python **modules, requirements files and virtual environments**.
- Differentiate between **scripts, modules, packages, and libraries** in Python
- Import and use modules from the **Python Standard Library**
- Create **custom Python modules and import and use them into scripts**
- Apply object-oriented principles to modularisation by defining **classes, functions and other variables within modules**
- Get with **Python code style guidelines (PEP 8)**, type hinting (**PEP 484**), and linting tools

Poll

Do you have Python installed and working well? Can you
hello_world.py

- Yes
- No


```
animal.py
1  class Animal:
2      def __init__(self, name, age):
3          self.name = name
4          self.age = age
5
6      def get_name(self):
7          return self.name
8
9      def get_age(self):
10         return self.age
11
12     def sleep(self):
13         return f"{self.name} sleeps"
14
15     def make_sound(self):
16         pass
17
18     def __str__(self):
19         return f"Animal: {self.name}"
```

```
cat.py
1  from animal import Animal
2
3  class Cat(Animal):
4      def __init__(self, name, age):
5          super(Cat, self).__init__(name, age)
6
7      def make_sound(self):
8          print("meow")
9
10     # def sleep(self):
11     #     print("cat sleeps")
```

```
dog.py
1  from animal import Animal
2  from cat import Cat
3
4  class Dog(Animal):
5      def __init__(self, name, age):
6          super(Dog, self).__init__(name, age)
7          self.cat_instance = Cat("", 0)
8
9      def make_sound(self):
10         print("wooo")
11
12     def sleep(self):
13         return self.cat_instance.sleep()
```

Poll

From the picture, why is this a bad design?

- The code demonstrates high coupling between classes due to interdependencies that may require changes in one class to be reflected in others.
- The code exhibits low coupling as each class is designed to be independent, minimising dependencies between them.
- The code shows moderate coupling because it relies on inheritance, which establishes a relationship between classes.
- The code doesn't display any significant coupling issues as it follows standard object-oriented principles.

Poll

How would you assess the level of cohesion in the provided object-oriented code?

- The code demonstrates high cohesion as each class encapsulates related functionality within itself.
- The code exhibits low cohesion because it contains unrelated methods within the same class.
- The code shows moderate cohesion due to its reliance on inheritance, which can lead to scattering of related functionality across multiple classes.
- The code doesn't display any significant cohesion issues as it adheres to standard object-oriented design principles.

Introduction



Analogy

Just as a toolbox organizes various tools into separate compartments, modules in programming organize related **functions**, **classes**, and **variables** into **separate "compartments"** or **"drawers"**. Each module serves a specific purpose, like a drawer containing tools for a particular task.

Just as you wouldn't mix your screwdrivers with your hammers, modules keep related elements separate and organised. When you need a specific function or variable, you can "open the drawer" (import the module) and access the tools (functions and variables) inside.



module_1.py

```
def function_1()  
  
def function_2()  
  
class Class1  
  
class Class2  
  
CONSTANT_1  
  
CONTSTANT_2
```

module_2.py

```
def function_3()  
  
def function_3()  
  
class Class3  
  
class Class4  
  
CONSTANT_3  
  
CONTSTANT_4
```

script.py

```
import module_1  
from module_1 import function_1  
from module_2 import *  
from module_2 import Class3  
  
cls = Class3()  
print(function_1())
```

Differentiating Scripts, Modules, Packages, and Libraries



Scripts


- A **script** is a standalone file containing **executable Python code**. It typically encapsulates a sequence of instructions to perform a specific task or set of tasks. It has the extension **.py**
- A **Jupyter notebook** is an **interactive document that combines code, text, and visualisations** in a browser-based environment, featuring cells for separate code execution and documentation, **making automation challenging** due to its interactive nature. It has the extension **.ipynb**

Scripts

- Scripts are designed to accomplish a particular goal or solve a specific problem
- They often automate repetitive tasks, process data
- Scripts can be standalone programs or part of a larger software system, focusing on a specific functionality or aspect of the application
- Shouldn't be used to implement new classes or functions. Those are for modules.

Scripts

- `"__name__"` variable in Python:
 - Special variable managed by Python
 - Automatically set:
 - To `"__main__"` when script is run directly.
 - To module's name (filename) when executed as part of an import statement.

Polls > animals >  driver.py

```
1 #####
2 #
3 # Please don't modify this script. It works #
4 #
5 # This is the driver script
6 #
7 # Run it as follows: python driver.py
8 #
9 #####
10
11 from cat import Cat
12 from dog import Dog
13
14 dog_1 = Dog("Rocky", 5)
15 cat_1 = Cat("Charlie", 3)
16
17 if __name__ == "__main__":
18
19     print(f"Dog 1: {dog_1}")
20     print(f"Cat 1: {cat_1}")
21
22     dog_1.get_name()
23     cat_1.get_name()
24
25     print(dog_1.sleep())
26     print(cat_1.sleep())
27
28     print(dog_1.make_sound())
29     print(cat_1.make_sound())
```


module_1.py

```
def function_1()  
  
def function_2()  
  
class Class1  
  
class Class2  
  
CONSTANT_1  
  
CONTSTANT_2
```

module_2.py

```
def function_3()  
  
def function_3()  
  
class Class3  
  
class Class4  
  
CONSTANT_3  
  
CONTSTANT_4
```

script.py

```
import module_1  
from module_1 import function_1  
from module_2 import *  
from module_2 import Class3  
  
cls = Class3()  
print(function_1())
```

Module

- A **module** is a Python file (.py) that encapsulates reusable code elements such as functions, classes, and variables.
- They can be accessed by **import**-ing the module into other Python files (modules or scripts), enabling code **reuse**, **maintenance** and organization
- Once imported, the functionalities defined in the module can be accessed and utilised in any script that imports it.

Module

Module-level names are global within the module, but they are not visible outside the module unless explicitly exported:

- Names defined at the module level, such as functions and variables, are accessible globally within the module
- However, these names are not visible to other scripts unless explicitly exported using techniques like the `__all__` list or using the `from module import *` syntax
- This **encapsulation** ensures that module internals remain private unless explicitly exposed, promoting encapsulation and preventing namespace pollution.

```
__all__ = ["addition", "subtraction", "division", "multiplication"]
```

```
def addition(x, y):  
    return _addition(x, y)
```

```
def subtraction(x, y):  
    return _subtraction(x, y)
```

```
def division(x, y):  
    return _division(x, y)
```

```
def multiplication(x, y):  
    return _multiplication(x, y)
```

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

```
def _subtraction(x, y):  
    return x - y
```

```
def _division(x, y):  
    return x / y
```

```
def _multiplication(x, y):  
    return x * y
```

```
from module_ops import *
```

```
x = 10
```

```
y = 20
```

```
if __name__ == "__main__":
```

```
    print(addition(x, y))
```

```
    print(subtraction(x, y))
```

```
    print(division(x, y))
```

```
    print(multiplication(x, y))
```

module_1.py

```
def function_1()  
  
def function_2()  
  
class Class1  
  
class Class2  
  
CONSTANT_1  
  
CONTSTANT_2
```

module_2.py

```
def function_3()  
  
def function_3()  
  
class Class3  
  
class Class4  
  
CONSTANT_3  
  
CONTSTANT_4
```

script.py

```
import module_1  
from module_1 import function_1  
from module_2 import *  
from module_2 import Class3  
  
cls = Class3()  
print(function_1())
```

Package

- A **package** is a directory that contains Python modules, along with a special **`__init__.py`** file that signifies it as a Python package
- Packages are a way of structuring Python's module namespace by using "dotted module names". For example, the module name `A.B` designates a submodule named `B` in a package named `A`.
- The **`__init__.py`** file can be empty or contain initialisation code for the package
- This hierarchical structure aids in managing and navigating larger projects by grouping modules into logical units.

EXPLORER



OPEN EDITORS

X driver.py

PROJECT FOLDER

math_operations

__init__.py

advanced_operations.py

basic_operations.py

string_operations

__init__.py

advanced_string_operations.py

basic_string_operations.py

__init__.py

driver.py

i readme.md

driver.py X

driver.py > ...

```
1  from math_operations.basic_operations import *
2  from string_operations.basic_string_operations import *
3
4  x = 10
5  y = 20
6  word_1 = "Hello"
7  word_2 = "world"
8
9  if __name__ == "__main__":
10
11      print(addition(x, y))
12      print(subtraction(x, y))
13      print(division(x, y))
14      print(multiplication(x, y))
15      print(concatenate_strings(word_1, word_2))
16
```

Package_1

module_1.py

```
def function_1()

def function_2()

class Class1

class Class2

CONSTANT_1

CONSTANT_2
```

module_2.py

```
def function_3()

def function_3()

class Class3

class Class4

CONSTANT_3

CONSTANT_4
```

`__init__.py`

script.py

```
import module_1
from package_1.module_1 import function_1
from package_1.module_2 import *
from package_1.module_2 import Class3
from package_1.module_2 import CONSTANT_3 as pi_value

if __name__ == "__main__":
    cls = Class3()
    print(function_1())
    result = pi_value ** 2
    print(result)
```

Library

- A **library** is fundamentally a collection of packages. Its objective is to offer a collection of ready-to-use features so that users won't need to be concerned about additional packages.



gunicorn

BeautifulSoup



Bandit

Bo**ttle**



seaborn



pandas

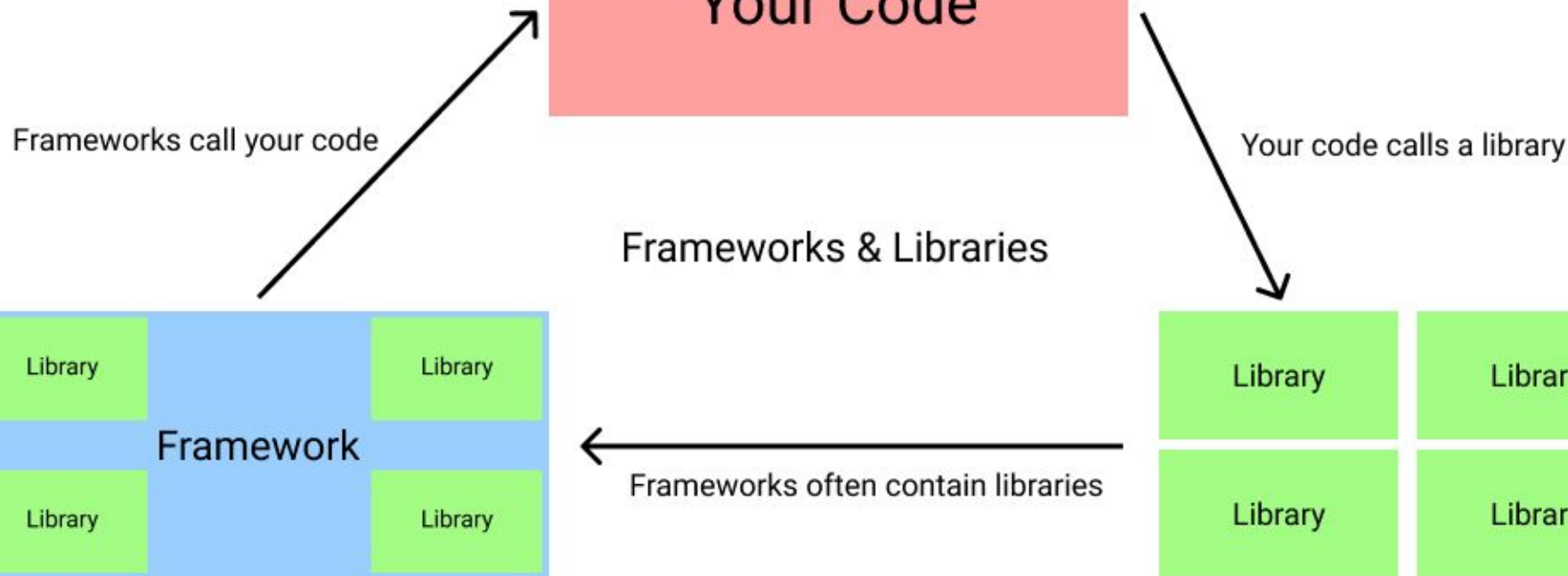


**It's
Dangerous**

Framework

- Frameworks are pre-written code libraries that provide a structure and set of tools to simplify the development of web applications, APIs, and more.
- They offer a foundation for building applications by providing common functionalities and design patterns.

Framework



Framework



Flask

django



FastAPI



TKINTER
GUI FOR PYTHON

Why all that?

1. **Code Organization:** Modules help **organise code** into logical **units**, making it easier to navigate and **manage** as your **project grows**.
2. **Reusability:** By encapsulating code into modules, **you can reuse functions, classes**, and variables across different parts of your program or in other projects, saving time and effort
3. **Maintainability:** Modular code is **easier to maintain** and **update**. Changes or fixes can be made to specific modules without affecting other parts of the codebase, leading to better organisation and collaboration among developers.

Python Standard Library Modules, pip and PyPi



Python Standard Library

- The **Python Standard Library (PSL)** is a collection of modules and packages that come **pre-installed with Python**.
- The PSL contains all the built-in functions commonly used like: **min, max, float, int, eval, print** <- Those do not even need to be imported
- It is considered to be the set of pillars building up the Python language
- The Python Standard Library is comprehensive, providing developers with tools to accomplish common tasks without having to install additional third-party packages.

Python Standard Library: Common Modules

Although, many python keywords do not need to be imported, some need the **import** keyword to be used. Those are built-in modules:

- **print**: Allowing you to perform mathematical operations
- **random**: generating random numbers
- **datetime**: Enables manipulation of dates and times
- **os**: Allows you to interact with the operating system
- **math**: Allowing you to perform mathematical operations

pip, PyPi

- **pip: Preferred Installer Program** is the package installer for Python. It allows you to install, upgrade, and manage Python packages from the Python Package Index (PyPi) or other sources. You can use pip to install third-party packages that are not included in the Python Standard Library.
- To install a new package: **pip install new_package**
- **PyPi:** PyPi is the official Python Package Index, a repository of software packages for Python. It hosts thousands of third-party packages that can be installed using pip.

Requirements File, and Virtual Environment

- **Requirements File**
 - Text file listing required Python packages and versions
 - Usually call **requirements.txt**
 - Ensures **exact dependencies** are installed for your project
 - **Facilitates replicating your environment for others**
- **Virtual Environment:**
 - Self-contained directory with Python interpreter and libraries
 - Isolates project dependencies
 - Prevents conflicts between projects or system-wide installations

Requirements File

<> Code 3.6k Issues Pull requests 131 Actions Projects Security Insights

Files

main

Go to file

- .circleci
- .github
- LICENSES
- asv_bench
- ci
- doc
- gitpod
- pandas
- scripts
- tooling
- typings
- web
- .devcontainer.json
- .gitattributes
- .gitignore

pandas / requirements-dev.txt

twoertwein TYP: register_*_accessor decorators (#58339) ✓

5281e0e · last week History

Code Blame 91 lines (90 loc) · 1.37 KB

Raw Copy Download Edit View Source

```
1 # This file is auto-generated from environment.yml, do not modify.
2 # See that file for comments about the need/usage of each dependency.
3
4 pip
5 versioneer[toml]
6 cython<=3.0.5
7 meson[ninja]==1.2.1
8 meson-python==0.13.1
9 pytest>=7.3.2
10 pytest-cov
11 pytest-xdist>=2.2.0
12 pytest-qt>=4.2.0
13 pytest-localserver
14 PyQt5>=5.15.9
15 coverage
16 python-dateutil
17 numpy<2
18 pytz
19 beautifulsoup4>=4.11.2
20 blocs
21 bottleneck>=1.3.6
22 fastparquet>=2023.10.0
23 fsspec>=2022.11.0
```



BREAK!

Python Code Style and Type Hinting



Examine this code and decide whether it is correct or not. Why?

```
def AreaOfCircle(radius):  
    area = 3.14159 * radius**2  
    return area
```

```
def Print_Area(radius):  
    print("The Area of the Circle with Radius", radius, "is", AreaOfCircle(radius))
```

```
def get_radius():  
    radius = float(input("Enter the Radius of the Circle:"))  
    return radius
```

```
def main():  
    Radius=get_radius()  
    Print_Area(Radius)
```

```
if __name__=="__main__":  
    main()
```

PEP 8 - Python Style Guide

- PEP 8 is the official style guide for Python code, providing guidelines on formatting, naming, and organising Python code.
- It aims to promote consistency and readability in Python code across projects and developers.
- It provides guidelines and best practices on how to write Python code

PEP 8 - Python Style Guide

Key Recommendations from PEP 8 (some)

- **Indentation:** Use 4 spaces (not tabs) per indentation level.
- **Maximum Line Length:** Limit all lines to a maximum of 79 characters.
- **Comments:** Comments that contradict the code are worse than no comments. Always make a priority of keeping the comments up-to-date when the code changes!
- **Class Names:** Class names should normally use the CapWords convention.
- **Method Names and Instance Variables:** Use the function naming rules: lowercase with words separated by underscores as necessary to improve readability.

PEP 8 Law Enforcement: Linting

- **Linting (Lint)** is the automated checking of your source code for programmatic and stylistic errors.
- Flags **unused constructs** such as variables and unreachable code
- Helps standardize code by replacing **tabs** with **spaces** or the other way around so that the codebase is written consistently.
- Makes it easier to review code because it ensures the reviewer that certain standards are already met.

Linting: Common Tools



flake8

The logo for the Ruff linter, featuring the word "RUFF" in a bold, yellow, monospace-style font, with "v0.1.0" in a smaller, yellow, monospace-style font to its right, all on a dark purple background.

RUFF v0.1.0

Type Hinting (PEP 484)

- It is the practice of adding annotations to Python code to indicate the expected types of function parameters, return values, and variables.
- Introduced in PEP 484, type hinting helps improve code clarity and enables static analysis tools to catch type-related errors.
- Improves code clarity by documenting expected types.
- Enables static analysis tools to perform type checking, catching errors early.
- Facilitates code maintenance by making code easier to understand and modify.

Type Hinting (PEP 484)

Type hints can be added using annotations for function parameters, return values, and variable declarations.

Example:

```
from typing import Tuple
```

```
def add(x: int, y: int) -> int
```

```
    list_value: List[int] = [1, 2, 3, 4, 5]
```

```
    person: Dict[str, str] = {"name": "Alice", "age": "30"}
```

```
    learner: Person = Person("Alice", 30)
```

```
    Coordinates = Tuple[int, int]
```

```
    return 9
```

Practical Exercise: Building a GUI Python Calculator



Objectives

I. Design the Calculator GUI:

- a. Design the layout of the calculator interface using tkinter, including buttons for digits, arithmetic operations, and clear/reset functionality.
- b. Organise GUI elements using layout managers (grid, pack, or place) and consider using frames for better organisation.

II. Implement the Calculator Logic:

- a. Create a Calculator class to encapsulate the logic and functionality of the calculator.
- b. Define methods within the Calculator class to perform arithmetic operations (addition, subtraction, multiplication, division) and handle user input.

Objectives

III. Modularization and Inheritance

- a. Organise the code into separate modules for better maintainability and code organisation.
- b. Create a module for the calculator GUI layout and functionality, and another module for the calculator logic
- c. Utilise inheritance to extend functionality, if applicable (e.g., creating specialised calculator classes)

IV. Testing and Debugging:

- a. Test the calculator's arithmetic operations
- b. Debug any errors or issues encountered during testing, ensuring the calculator functions as expected
- c. Use print statements or logging to debug and trace the flow of execution if necessary.

Final Assessment - Poll



Poll

1 - Which of the following accurately describes the purpose of Python modules?

- A. To organise code into reusable units and promote maintainability.
- B. To execute specific tasks within a Python script
- C. To provide graphical user interfaces (GUIs) for Python applications.
- D. To manage dependencies between Python packages

Poll

2 - Given a Python script named `main.py` and a module named `my_module.py` in the same directory, what is the proper way to import the `my_module` module within `main.py`?

- A. `import my_module`
- B. `from my_module import *`
- C. `import my_module.py`
- D. `import .my_module`

Poll

3 - Given a Python module named `my_module.py` containing a function named `my_function`, how would you import and use this function within another Python script?

- A. `import my_module.my_function`
- B. `from my_module import my_function`
- C. `import my_function from my_module`
- D. `from my_function import my_module`

Lesson Conclusion and Recap



Summary

- **Understanding Modules**
 - Modules are used in Python to organise code into reusable units, enhancing maintainability and readability.
- **Exploring Standard Library Modules**
 - Python's Standard Library offers a wide range of modules for common tasks, such as math calculations, file manipulation, and datetime operations.
- **Differentiating Components**
 - Scripts, modules, packages, and libraries serve distinct purposes in Python, with modules acting as reusable units of code.

Summary

- **Creating Custom Modules**
 - Create own modules by encapsulating related code in separate `.py` files, promoting code organisation and reusability.
- **Importing Modules**
 - Modules are imported into Python scripts using the ``import`` statement, providing access to their contents.
- **Understanding Object-Oriented Principles**
 - i. Object-oriented programming principles like encapsulation and inheritance can be applied to modularisation, allowing for the creation of versatile and extensible modules.

Homework/ Follow-up Activities



Homework/ Follow-up Activities

Use the code from the practical and add a factorial operation such that $5! = 120$.

1. Add the factorial (!) button at the position of your choice
2. Add the factorial functionality
3. Make sure that the result comes out on the screen
4. Have a test case for it
5. Do not break the rest of the code

Questions and Answers



References

- <https://docs.python.org/3/library/index.html>
- <https://docs.python.org/3/library/functions.html>
- <https://www.toppr.com/guides/python-guide/references/methods-and-functions/python-standard-library-reference/>
- <https://peps.python.org/pep-0001/#what-is-a-pep>
- <https://peps.python.org/pep-0008/>
- <https://designenterprisestudio.com/2022/05/26/libraries-frameworks/>

Thank you for attending



Department
for Education

CoGrammar

