Selective Imports, Building Your Own Modules & the __main__ Guard

AiCE Software Engineering Foundations — Python Programming (Intermediate)

This closing lesson on modules covers a more selective way to import, how to build and organize your own reusable modules, and the guard that keeps test code from running when a file is imported. It closes with a capstone report generator combining everything from this unit.

1. The from...import Syntax

The standard import module_name syntax imports the entire module — every time you use something from it, you prefix it with the module name: math.sqrt(), random.choice(). This is the recommended approach in this program because it makes clear where each function comes from.

Python also supports a more selective import syntax: from module import name. This imports specific items directly into your namespace, so you can use them without the module prefix:

```
# Standard import — prefix required
import math
print(math.sqrt(49))    # math.sqrt
print(math.pi)          # math.pi

# Selective import — no prefix needed
from math import sqrt, pi
print(sqrt(49))         # No prefix
print(pi)               # No prefix
```

Both produce identical results. The difference is clarity and namespace management:

Approach	Syntax	Advantage	Disadvantage
Full import	import math	Clear origin of every function	Longer to type — always need math.

Approach	Syntax	Advantage	Disadvantage
Selective import	from math import sqrt	Shorter, cleaner code	Unclear where sqrt came from without checking imports
Alias import	import numpy as np	Shorter alias for frequently used modules	Non-standard aliases can confuse readers

Never Use from module import *

*import module_name is preferred when you use many things from a module, or when clarity matters. from module import name is preferred when you use just one or two specific items repeatedly. Never use from module import * (import everything) — it pollutes your namespace and makes it impossible to know where names come from.*

2. Creating Your Own Modules

A module is simply a Python file. Any .py file you create can be imported into another Python file — this is how you organize large programs across multiple files, with each file handling one area of responsibility, and the main program importing what it needs.

Step 1 — Create the Module File

Create a file called aice_utils.py — a custom module of reusable utilities:

```
# aice_utils.py – Custom utility module for AiCE programs
PLATFORM_NAME = "AiCE"
PASSING_MARK = 50
DISTINCTION = 70

def get_grade(score):
    """Return the letter grade for a given score."""
    if score >= DISTINCTION:
        return "A"
    elif score >= 60:
        return "B"
    elif score >= PASSING_MARK:
        return "C"
```

```
    else:
        return "F"

def format_name(name):
    """Strip whitespace and apply title case to a name."""
    return name.strip().title()

def calculate_average(scores):
    """Return the average of a list of scores, or 0 if
empty."""
    if not scores:
        return 0
    return sum(scores) / len(scores)

def print_banner(title):
    """Print a formatted AiCE banner with a title."""
    border = "=" * 50
    print(border)
    print(f" {PLATFORM_NAME} | {title}")
    print(border)
```

This file is now a module, containing constants, functions, and a banner printer.

Step 2 — Import and Use the Module

Create a second file in the same directory:

```
# main_program.py — Imports and uses the custom aice_utils
module
import aice_utils

aice_utils.print_banner("Student Results System")

students = [
    {"name": " amara diallo ", "score": 88},
    {"name": "KOFI mensah",    "score": 45},
    {"name": "ngozi ADEYEMI",  "score": 92},
]

for student in students:
```

```
    clean_name = aice_utils.format_name(student["name"])
    grade = aice_utils.get_grade(student["score"])
    status = "Pass" if student["score"] >=
aice_utils.PASSING_MARK else "Fail"
    print(f"{clean_name:<18} | {student["score"]:>3} | {grade}
| {status}")
```

main_program.py imports aice_utils and uses its functions and constants throughout. Notice that aice_utils doesn't know or care about main_program — it's a general-purpose utility. This separation — utility logic in one file, application logic in another — is standard professional Python architecture.

> **Module Files Must Be in the Same Directory**
> *When you import a custom module, Python looks for it in the same directory as the importing file by default. If your files are in different directories, you need to configure the Python path — a topic covered when you start building larger projects. For now, keep all your files in the same folder.*

3. The if __name__ == '__main__' Guard

When you import a module, Python executes all the code in that file from top to bottom. This is normally fine — but what if your module file also contains test code you only want to run when the file is executed directly, not when it's imported?

Every Python file has a built-in variable called __name__. When a file is run directly, __name__ is set to '__main__'. When a file is imported as a module, __name__ is set to the file's own name instead.

```
# aice_utils.py — with the guard added at the bottom
def get_grade(score):
    if score >= 70: return "A"
    elif score >= 50: return "C"
    else: return "F"

# This block ONLY runs when the file is executed directly
# It does NOT run when the file is imported
if __name__ == "__main__":
```

```
    print("Testing aice_utils module...")
    print(get_grade(85))    # A
    print(get_grade(30))    # F
    print("All tests passed.")
```

With this guard in place: running python3 aice_utils.py directly executes the test block; importing import aice_utils in another file loads the functions without running any test code.

Standard Practice

Always add the if __name__ == '__main__' guard to any module file that also contains test or demonstration code. This pattern is standard practice in professional Python.

Wisdom, Not Laziness

You do not have to build everything yourself. The Python Standard Library is one of the most comprehensive collections of pre-built tools in any programming language. Using it is not laziness — it is wisdom. The best engineers know which tools already exist and when to reach for them.

4. Bringing It All Together — The AiCE Daily Report Generator

A capstone combining math, random, datetime, os, and a custom module to produce a daily report:

```
# AiCE Daily Report Generator
import math
import random
import datetime
import os
import aice_utils

def generate_report():
    now = datetime.datetime.now()
    date_str = now.strftime("%A, %d %B %Y")
```

```
    random.seed(42)
    num_students = random.randint(20, 50)
    scores = [random.randint(25, 100) for _ in
range(num_students)]

    avg = aice_utils.calculate_average(scores)
    passed = len([s for s in scores if s >=
aice_utils.PASSING_MARK])
    pass_rate = (passed / num_students) * 100

    variance = sum((s - avg) ** 2 for s in scores) /
num_students
    std_dev = math.sqrt(variance)

    os.makedirs("reports", exist_ok=True)

    aice_utils.print_banner("DAILY COHORT REPORT")
    print(f" Date           : {date_str}")
    print(f" Students       : {num_students}")
    print(f" Average Score  : {avg:.2f}")
    print(f" Std Deviation  : {std_dev:.2f}")
    print(f" Passed         : {passed} ({pass_rate:.1f}%)")

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

This program imports five modules — four from the standard library and one custom — and uses them together to generate a realistic daily report. This is the kind of automated report generation script used in school management systems, HR platforms, and business intelligence tools across Africa.

Key Takeaways

from module import name gives you specific tools without the module prefix, but full import module_name remains the clearer default — and from module import * should never be used. Any .py file is a module, and splitting utility logic into its own file, imported by a separate application file, is standard professional architecture. The if __name__ == '__main__' guard keeps test and demonstration code from running whenever a file is imported rather than executed directly. With modules mastered, the

next unit introduces file handling — reading data from files on disk and writing output that persists between program runs, making your programs genuinely connected to the real world.