

Advanced Programming

Modules in Python

Ramaseshan Ramachandran

IMPORT STATEMENT

- ✦ Gains access to the code of one module in another
- ✦ Finds a Python module with a specified name to access the functions and data types defined within that module
- ✦ Two functions
 - ✦ Searches for the module named in the import statement
 - ✦ Binds the results of the search to the local scope
- ✦ Import programmatically

```
if __name__ == "__main__":  
    import importlib  
    my_math = importlib.import_module('math')  
    print(my_math.__doc__)  
    print(dir(my_math))
```

MODULES

- ✦ If the program gets longer, you may want to logically split it into several files for easier maintenance
- ✦ A function may also be used in several of your programs
- ✦ You want to use it without copying it
- ✦ Modules promote code organization and reusability.
- ✦ They group related functionalities into logical units
- ✦ They prevent global namespace pollution – names are local to the module
- ✦ A **module** is a file containing Python code that acts as a reusable building block
 - ✦ Self-contained library of functions, variables, and classes

Code organisation

Reusability

Maintainability


Namespace management

MODULE STRUCTURE

```
my_module/  
├── __init__.py # (Optional initialisation code)  
├── function1.py  
└── function2.py
```

✓ algorithms

 __init__.py

 recursion.py

`__init__.py`

- ✦ Marks a directory as a package
- ✦ Python interprets it as a package rather than a single module
- ✦ Package-level variables or constants
- ✦ Performing one-time initializations (e.g., connecting to databases)
- ✦ Setting up logging or configuration

Functions in Module “Algorithms”

```
import math
def factorial(n) -> int:
    if not n >= 0:
        raise ValueError("n must be >= 0")
    if math.floor(n) != n:
        raise ValueError("n must be exact integer")
    if n+1 == n: # catch a value like 1e300
        raise OverflowError("n too large")
    result = 1
    factor = 2
    while factor <= n:
        result *= factor
        factor += 1
    return result

def fibonacci(n) -> int:
    if n <= 1:
        return n
    else:
        return(fibonacci(n-1) + fibonacci(n-2))

if __name__ == "__main__":
    print(factorial(5))
    print(fibonacci(15))
```

if `__name__ == "__main__":`

- ✦ `__name__`: A special built-in variable set to `"__main__"`
- ✦ Only executes when the file is run as a script
- ✦ Useful for tasks specific to standalone execution
- ✦ Advantages:
 - ✦ Prevents unintended side effects when imported
 - ✦ Creates cleaner and more modular code

SCOPE

- ✦ The scope determines the visibility and accessibility of variables within different parts code
- ✦ Scopes are implemented as dictionaries
- ✦ These dictionaries are called namespaces.
- ✦ Scope = {name:object}
- ✦ Follows Local, Enclosing, Global and built-in (LEGB) rule
- ✦ Global scope

```
global_var = 40

def my_function():
    local_var = 100
    print(global_var)
if __name__ == '__main__':
    print(local_var)
    my_function()
```

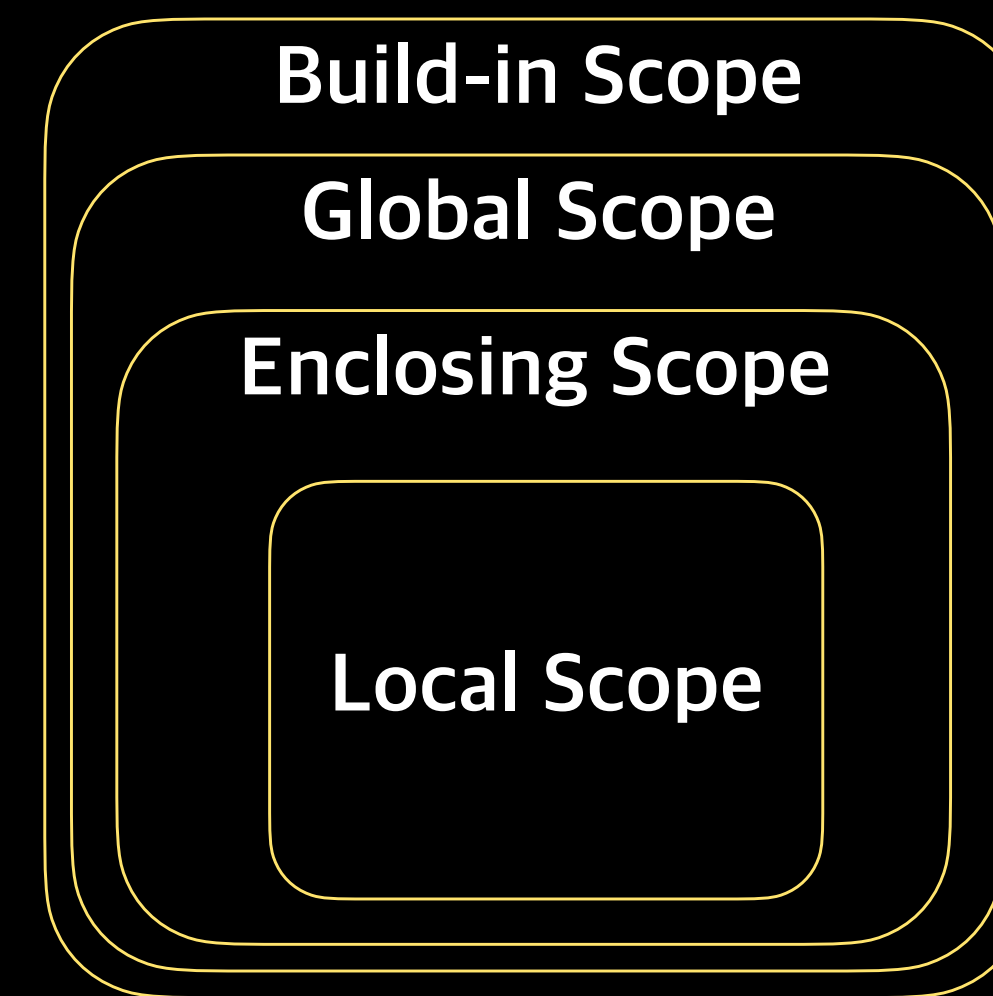
Built-in Scope

Built-in functions and variables that are always accessible

◦ Examples: `print()`, `len()`, `True`, `False`.

LEGB RULE

- ✦ When Python encounters a variable name
 - ✦ Local: It checks if the variable is defined within the current block (function, loop, or conditional).
 - ✦ Enclosing: If not found locally, it looks for it in enclosing blocks (outer functions/blocks).
 - ✦ Global: If still not found, it checks the global scope of the module.
 - ✦ Built-in: If not found globally, it checks the built-in scope
- ✦ Minimise global variable usage
- ✦ Use descriptive variable names
- ✦ Consider using classes to encapsulate related variables and functions, providing better control over their scope



CHECK YOUR UNDERSTANDING

```
def scope_test():
    def do_local():
        spam = "local spam"

    def do_nonlocal():
        nonlocal spam
        spam = "nonlocal spam"

    def do_global():
        global spam
        spam = "global spam"

    spam = "test spam"
    do_local()
    print("After local assignment:", spam)
    do_nonlocal()
    print("After nonlocal assignment:", spam)
    do_global()
    print("After global assignment:", spam)

if __name__ == '__main__':
    scope_test()
    print("In global scope:", spam)
```

After local assignment: test spam

After nonlocal assignment: nonlocal spam

After global assignment: nonlocal spam

In global scope: global spam

DECORATOR

- ✦ Is a design pattern
- ✦ Modifies the behaviour of functions
- ✦ Adds or alters its functionality without permanently changing the original code
- ✦ Promotes code reuse, readability, and maintainability

Prerequisite



```
def outer_function(x):  
    def inner_function(y):  
        return y * 2  
  
    result = inner_function(x)  
    return result  
  
def m_by_2(x):  
    return x * 2  
  
def calculate(func, x):  
    return func(x)  
  
def return_func_as_value(x):  
    def calculate(x):  
        return x*2  
    return calculate  
  
if __name__ == '__main__':  
    print(outer_function(5))  
    print(calculate(m_by_2,5))  
    calc = return_func_as_value(5)  
    print(calc(5))
```

DECORATOR SYNTAX

- ✦ Takes the original function as an argument (called as wrapper).
- ✦ Inside the decorator, one can perform actions before, after, or around the execution of the wrapped function
- ✦ Returns the Wrapper Function
 - ✦ Encapsulates the modified behaviour of the original function
- ✦ Decoration Syntax:
 - ✦ The @ symbol is used to apply the decorator to a function.
 - ✦ The decorator function is placed above a function that needs to be decorated

```
@greeting_decorator  
def say_hello(name, cmi="CMI"):
```

```
#efficiently computing  
#Fibonacci numbers using a  
#least recently used cache  
@lru_cache(maxsize=None)  
def fibonacci(n):
```

```
@requires_role("admin")  
def update_settings(config):
```

DECORATOR - EXAMPLE 1

```
import datetime

def time_of_day():
    # Get the current hour
    current_hour = datetime.time.hour

    # time ranges
    morning_end = 12
    evening_end = 21

    if morning_start < (current_hour and 12):
        return "Good morning"

    elif evening_start <= (current_hour and 21):
        return "Good evening"

    else:
        return "Good night"
```

```
# Define a decorator function
def greetings_decorator(func):
    def wrapper(*args, **kwargs):
        # Call the original function
        print(f"{func(*args, **kwargs)}, \
              {time_of_day()}")
    return wrapper

# Decorator
@greeting_decorator
def say_hello(name, cmi="CMI"):
    return f"Hello, {name} {cmi}!"

if __name__ == "__main__":
    say_hello("Students of ", cmi="ADPG")
```

DECORATOR - EXAMPLE 2

```
import time

def measure_time(func):
    def wrapper(*args, **kwargs):
        start_time = time.time()
        result = func(*args, **kwargs)
        end_time = time.time()
        print(f"Execution time of {func.__name__}: {end_time - start_time:0.2e} seconds")
        return result
    return wrapper

# Decorate
@measure_time
def example_function(n):
    sum = 0
    for i in range(1, n+1):
        sum += i
    return sum

if __name__ == '__main__':
    result = example_function(1000000)
    print(result)
```

USE CASES OF DECORATORS

- ✦ Logging
 - ✦ Track function calls and their results for debugging purposes
- ✦ Authentication and Authorisation
 - ✦ Control access to functions based on user permissions
- ✦ Caching
 - ✦ Store function results to avoid redundant calculations
- ✦ Error Handling
 - ✦ Handle exceptions gracefully and provide informative error messages
- ✦ Performance Measurement
 - ✦ Time how long functions take to execute for optimization

EXERCISE

- ✦ Count calls decorator:
 - ✦ Create a decorator named `count_calls` that keeps track of how many times a factorial function is called and prints the count before each call
- ✦ Retry_decorator:
 - ✦ Create a decorator named `retry_on_wrong_answer` that allows a student to retake a simple quiz function (simulated with multiple-choice questions) a certain number of times before failing. The decorator should display the current question, available choices, and track the student's answer and number of attempts
- ✦ Rate Limiting Decorator:
 - ✦ Implement a rate-limiting decorator `rate_limit` that restricts the number of times a function can be called within a certain time period