

✓ Modules, Error and Exceptions 1

Agenda

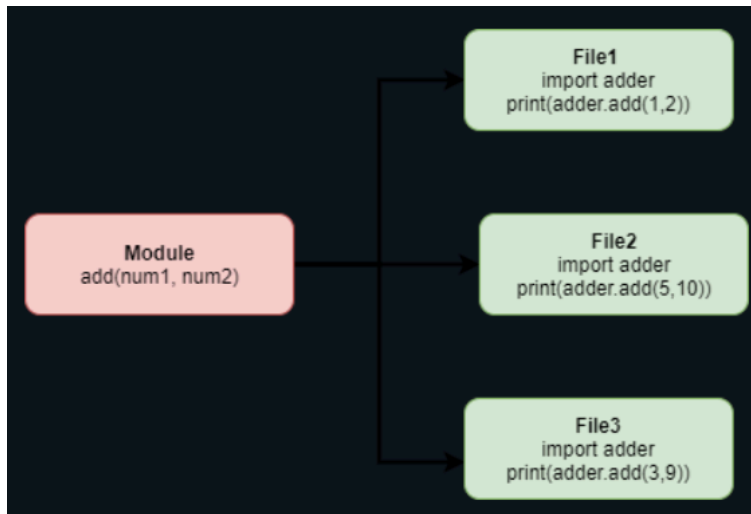
1. Modules

- Importing Modules
 - Basic Import
 - Importing Specific Functions/Variables
 - Renaming Modules or Functions
- Creating Custom Modules
 - init.py
- Exploring Built-in Modules
- Namespaces and Scoping

✓ Modules

✓ What is Python Module?

- A document with definitions of functions and various statements written in python is called a Python module.
- It encapsulates functions, classes, and variables.
- Standard library modules provide built-in functionalities (e.g., math, os, random).



- In Python, we can define a module in one of 3 ways:
 - Python itself allows for the creation of modules.
 - Similar to the re (regular expression) module, a module can be primarily written in C programming language and then dynamically inserted at run-time.
 - A built-in module, such as the itertools module, is inherently included in the interpreter.

italicized text### Why modules are useful:-

- Modules provide a structured and scalable way to manage and organize code in large projects. They facilitate reusability, maintainability, and collaboration, making the development process more efficient and the codebase more robust.
- Reusability: Once a module is created, it can be reused across different parts of the project or even in other projects. This reduces duplication of code and promotes the DRY (Don't Repeat Yourself) principle.

- Scalability: As projects grow, maintaining a monolithic codebase becomes challenging. By using modules, you can scale your project more effectively, as changes to one module won't necessarily affect others.
- Namespace Management: Modules create separate namespaces, which help prevent name collisions in large projects. This means you can have functions, variables, or classes with the same name in different modules without conflict.
- Dependency Management: Modules allow you to clearly define dependencies between different parts of the code. This makes it easier to manage and update dependencies without affecting unrelated parts of the project.

```
# Here, we are creating a simple Python program to show how to create a module.
# defining a function in the module to reuse it
def square( number ):
    # here, the above function will square the number passed as the input
    result = number ** 2
    return result    # here, we are returning the result of the function
```

✓ Creating Custom Modules

- Creating custom modules in Python allows you to organize and reuse code effectively. A module is simply a Python file (.py) that contains functions, classes, and variables. Here's how you can create and use your own custom modules:

1. Create a Python File:

- Create a new file with a .py extension. This file will contain your module code.

```
#Example: Creating my_module.py
def greet(name):
    return f"Hello, {name}!"

def add(a, b):
    return a + b

PI = 3.14159
```

2. Save the File:

- Save my_module.py in the same directory as your main script or in a directory that's included in the Python path.

3. Import the Module:

- You can import your custom module into another script using the import statement.

```
import my_module

# Use the functions and variables from the module
print(my_module.greet("Alice")) # Outputs: Hello, Alice!
print(my_module.add(5, 3))      # Outputs: 8
print(my_module.PI)             # Outputs: 3.14159
```

```
-----
ModuleNotFoundError                                Traceback (most recent call last)
/tmp/ipython-input-3-3532429425.py in <cell line: 0>()
----> 1 import my_module
      2
      3 # Use the functions and variables from the module
      4 print(my_module.greet("Alice")) # Outputs: Hello, Alice!
      5 print(my_module.add(5, 3))      # Outputs: 8

ModuleNotFoundError: No module named 'my_module'
```

NOTE: If your import is failing due to a missing package, you can manually install dependencies using either !pip or !apt.

To view examples of installing some common dependencies, click the "Open Examples" button below.

OPEN EXAMPLES

✓ init.py File

- **init.py**: This file is crucial for turning a directory into a Python package. It can be used to set up package-level variables, import specific functions or classes to make them accessible directly from the package, or initialize the package.
- When you have a package in Python, you can organize your code into multiple modules and sub-packages. The presence of an **init.py** file in a directory indicates that the directory is a package.
- Consider the following directory structure for a package:

```
my_package/
|
├── __init__.py
├── module1.py
├── module2.py
|
└── sub_package/
    ├── __init__.py
    ├── sub_module1.py
    └── sub_module2.py
```

- if **name == "main"**: Construct in Python is crucial for controlling how a module behaves when it's run as a script versus when it's imported as a module in another script. Understanding its importance is key to writing modular, reusable, and maintainable code.

✓ How it works:

- **name** Variable: Every Python module has a built-in attribute called **name**. When a module is run as the main program, the **name** variable is set to **'main'**. If the module is imported into another module, **name** is set to the module's name.
- if **name == "main"**:: This condition checks whether the module is being run as the main program. If it is, the code block under this condition will execute. If the module is imported elsewhere, this code block will not run.

Start coding or [generate](#) with AI.

Imagine you have a file calculator.py:

```
def add(a, b):
    return a + b
```

```
def subtract(a, b):
    return a - b
```

```
if __name__ == "__main__":
    print("Running calculator as a standalone script.")
    # Example usage
    print("2 + 3 =", add(2, 3))
    print("5 - 2 =", subtract(5, 2))
```

```
➦ Running calculator as a standalone script.
2 + 3 = 5
5 - 2 = 3
```

✓ Exploring Built-in Modules

- Python comes with a rich standard library that includes many built-in modules, which provide a wide range of functionalities without the need for additional installations. These modules cover various aspects of programming, including file I/O, system operations, data manipulation, and more.

- **os:** Interact with the operating system (file operations, environment variables, etc.)
- **sys:** Interact with the Python interpreter (arguments, exit, modules, etc.)
- **math:** Mathematical functions (trigonometry, logarithms, etc.)
- **random:** Generate random numbers and sequences
- **time:** Handle time-related operations (time, date, calendars, etc.)
- **datetime:** More advanced date and time handling
- **json:** Encode and decode JSON data
- **pickle:** Serialize and deserialize Python objects
- **re:** Regular expressions for pattern matching
- **csv:** Read and write CSV files
- **collections:** Specialized container data types (defaultdict, Counter, namedtuple, etc.)

✓ OS Module

- The os module provides a way to interact with the operating system. It allows you to work with directories and files, handle environment variables, and execute system commands.

```
import os

# Get the current working directory
print(os.getcwd())

# List files and directories in the current directory
print(os.listdir('.'))

# Create a new directory
os.mkdir('new_directory')

# Remove a file
# os.remove('some_file.txt')
```

```
↗ /content
[ '.config', 'sample_data' ]
```

✓ SYS Module

*The sys module provides access to system-specific parameters and functions. It's useful for manipulating the Python runtime environment.

```
import sys

# Print command-line arguments
print(sys.argv)

# Exit the program with a specific status code
# sys.exit(1)

# Print the platform
print(sys.platform)

↗ [ '/usr/local/lib/python3.10/dist-packages/colab_kernel_launcher.py', '-f', '/root/.local/share/jupyter/runtime/kernel-bc905b7f-97a4-478linux
```

✓ random Module

- The random module implements pseudo-random number generators for various distributions.

```
import random

# Generate a random integer between 1 and 10
print(random.randint(1, 10))

# Generate a random float between 0.0 and 1.0
print(random.random())

# Choose a random element from a list
print(random.choice(['apple', 'banana', 'cherry']))
```

```
3
0.854060783292562
apple
```

▼ datetime Module

- The datetime module provides classes for manipulating dates and times.

```
from datetime import datetime, timedelta

# Get the current date and time
now = datetime.now()
print(now)

# Add 10 days to the current date
future_date = now + timedelta(days=10)
print(future_date)
```

```
2025-07-05 16:29:24.970808
2025-07-15 16:29:24.970808
```

▼ json Module

- The json module provides methods for parsing JSON data and converting Python objects to JSON format.


```
{
  "user_id": 101,
  "name": "Shantanu Sharma",
  "email": "shantanu@example.com",
  "is_active": true,
  "age": 28,
  "roles": ["admin", "editor"],
  "address": {
    "street": "123 MG Road",
    "city": "Indore",
    "state": "Madhya Pradesh",
    "postal_code": "452001"
  },
  "purchase_history": [
    {
      "order_id": "A001",
      "item": "Laptop",
      "price": 75000,
      "date": "2024-11-22"
    },
    {
      "order_id": "A002",
      "item": "Wireless Mouse",
      "price": 1200,
      "date": "2025-01-15"
    }
  ]
}
```

```
import json

# Convert a Python dictionary to a JSON string
data = {'name': 'Alice', 'age': 30}
json_str = json.dumps(data)
```

```
print(json_str)

# Parse a JSON string into a Python dictionary
parsed_data = json.loads(json_str)
print(parsed_data)
```

 `{"name": "Alice", "age": 30}`
`{'name': 'Alice', 'age': 30}`

✓ Pathlib Module

- pathlib is a module in Python that provides an object-oriented approach to working with filesystem paths.

```
from pathlib import Path

# Define a path
file_path = Path('example.txt')


# Create and write to a file
with file_path.open('w') as file:
    file.write("This is a test file.")

# Read from the file
with file_path.open('r') as file:
    content = file.read()
    print("File Content:", content)

# Check if the file exists
print("File exists:", file_path.exists())

# Create a new directory
dir_path = Path('new_directory')
dir_path.mkdir(exist_ok=True)

# List contents of the current directory
for item in Path('.').iterdir():
    print("Item:", item)
```

 File Content: This is a test file.
File exists: True
Item: .config
Item: example.txt
Item: new_directory
Item: sample_data

✓ Itertools Module

- The itertools module is a powerful tool for handling iterators and performing various operations on sequences. It provides efficient and flexible methods to work with iterables, making it a valuable addition to Python's standard library.

```
import itertools

# Example data
numbers = [1, 2, 3, 4]

# 1. Count
print("Count:")
for num in itertools.count(start=1, step=2):
    if num > 7:
        break
    print(num)

# 2. Cycle
print("\nCycle:")
colors = ['red', 'green', 'blue']
for color in itertools.cycle(colors):
    if color == 'blue':
        break
    print(color)

# 3. Chain
```

```

print("\nChain:")
for item in itertools.chain([1, 2], ['a', 'b']):
    print(item)

# 4. Combinations
print("\nCombinations:")
items = ['A', 'B', 'C']
for comb in itertools.combinations(items, 2):
    print(comb)

# 5. Groupby
print("\nGroupby:")
data = [('a', 1), ('b', 2), ('a', 3), ('b', 4)]
for key, group in itertools.groupby(data, key=lambda x: x[0]):
    print(f'Key: {key}')
    for item in group:
        print(item)

```

```

↔ Count:
1
3
5
7

```

```

Cycle:
red
green

```

```

Chain:
1
2
a
b

```

```

Combinations:
('A', 'B')
('A', 'C')
('B', 'C')

```

```

Groupby:
Key: a
('a', 1)
Key: b
('b', 2)
Key: a
('a', 3)
Key: b
('b', 4)

```

```
type(data)
```

```
↔ list
```

```

# 4. Combinations
print("\nCombinations:")
items = ['A', 'B', 'C']
for comb in itertools.combinations(items,0):
    print(comb)

```

```

↔ Combinations:
()

```

✓ Namespaces and scoping

✓ Namespace

- A namespace in Python is a container that holds a collection of identifiers (names) and their corresponding objects (e.g., variables, functions, classes). Namespaces ensure that names are unique and avoid conflicts between different parts of a program.
- Types of Namespace:-

1. Local Namespace:

- Created within a function or method.
- Contains names that are local to the function or method.
- Exists only during the execution of the function.

```
def my_function():
    local_var = 10 # Local namespace
    print(local_var)

my_function()
# print(local_var) # Error: NameError, local_var is not defined outside the function
```

10

2. Enclosing Namespace:

- Refers to namespaces in enclosing functions (e.g., nested functions).
- The scope of names in the enclosing namespace is accessible to nested functions.

```
def outer_function():
    outer_var = 20

    def inner_function():
        print(outer_var) # Accesses name from enclosing namespace

    inner_function()

outer_function()
```

20

3. Global Namespace:

- Created at the level of the main program or module.
- Contains names that are globally accessible within the module.

```
global_var = 30 # Global namespace

def my_function():
    #global global_var
    #global_var = 50
    print(global_var) # Accesses global name

my_function()
#print(global_var)
```

30

4. Built-in Namespace:

- Contains built-in functions and exceptions.
- Available globally in any Python script.

```
print(len("DSML")) # len is a built-in function
```

4

5. Non-local Namespace:

- The nonlocal keyword is used to work with variables in an outer, but non-global, namespace, which is typically a function scope that is not the global scope. This is important when you have nested functions and want to modify variables defined in an enclosing function

scope.

```
def outer_function():
    x = 'outer' # Variable in the outer (enclosing) scope

    def inner_function():
        nonlocal x # Refers to the variable `x` in the enclosing scope
        x = 'inner' # Modifies the variable `x` in the outer scope
        print(f'Inner function: x = {x}')

    inner_function()
    print(f'Outer function: x = {x}')

outer_function()
```

↩ Inner function: x = inner
Outer function: x = inner

✓ Scoping

- Scoping determines the visibility and lifetime of a variable within different parts of the code. Python uses the LEGB rule to resolve names:

1. Local Scope (L):

- Names defined within the current function or method. Highest priority when resolving names.

2. Enclosing Scope (E):

- Names in the enclosing functions (nested functions). Intermediate priority.

3. Global Scope (G):

- Names defined at the top level of a module or script. Lower priority than local and enclosing scopes.

4. Built-in Scope (B):

- Names provided by Python's built-in namespace (e.g., print, len). Lowest priority.

✓ Example of Scoping:

```
x = 10 # Global scope

def outer_function():
    x = 20 # Enclosing scope

    def inner_function():
        x = 30 # Local scope
        print(x)

    inner_function()
    print(x)

outer_function()
print(x)
```

↩ 30
20
10

✓ Global Keyword

* Used to modify variables in the global scope from within a function.

```
x = 10

def modify_global():
    # global x
    x = 20
```

```
modify_global()  
print(x)
```

 10

```
# Create a dummy my_module.py file with the content from cell Mc8cAaZpHei-  
# This allows the next cell to import it  
with open('my_module.py', 'w') as f:  
    f.write("""  
def greet(name):  
    return f"Hello, {name}!"  
  
def add(a, b):  
    return a + b  
  
PI = 3.14159  
""")
```