Unit-2: File & Exception handling

2.1: User defined Modules and Packages in Python

2.2: Files: File manipulations

2.2.1: File handling (text and CSV files) using CSV module :CSV

module , File modes: Read , write, append

2.2.2: Important Classes and Functions of CSV modules:Open(),

reader(), writer(), writerows(), DictReader(), DictWriter()

2.2.3: File and Directory related methods

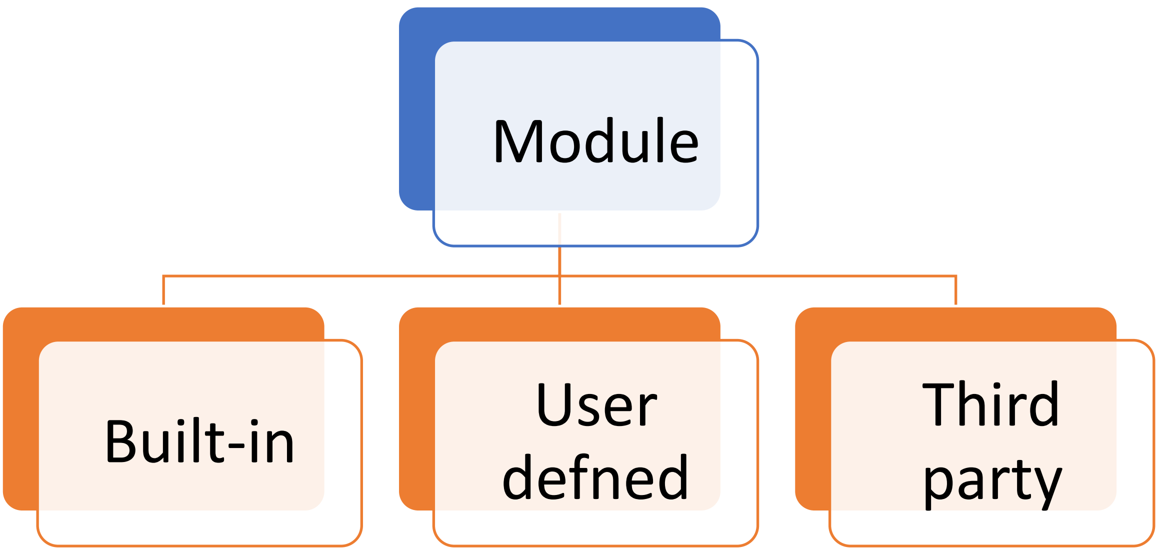
2.3: Python Exception Handling

Modules

- A module is a file containing Python definitions and statements. A module can define functions, classes, and variables. The variables can be of any type (arrays, dictionaries, objects, etc.)
- A module can also include runnable code.
- Grouping related code into a module makes the code easier to understand and use. It also makes the code logically organized.
- Modules provide us with a way to share reusable functions.

In short : A Python module is a Python file with extension .py that contains classes, methods, or variables that you'd like to include in your application.

Types of modules



1. **Built-in modules**: Come with Python (e.g. math, os, sys, random)
2. **User-defined modules**: You create these yourself like math_utils.py
3. **Third-party modules**: Installed via pip (e.g. numpy, pandas, requests)

Note: You do not need to store the module in the same folder as your main file.

Sys

- Sys is a built-in Python module that contains parameters specific to the system i.e. it contains variables and methods that interact with the interpreter and are also governed by it.

sys.path

- sys.path is a built-in variable within the sys module.
- It contains a list of directories that the interpreter will search in for the required module.

How Python Uses sys.path

- When a module(a module is a python file) is imported within a Python file, the interpreter first searches for the specified module among its built-in modules. If not found it looks through the list of directories(a directory is a folder that contains related modules) defined by sys.path.
- By default, the interpreter looks for a module within the current directory.
- To make the interpreter search in some other directory you just simply have to change the current directory by appending the path.

What sys.path Contains

- sys.path is a list that includes:
- The directory containing the input script (or the current directory when no script is specified)
- The PYTHONPATH environment variable directories (if set)
- Installation-dependent default paths (including standard library locations)

Viewing sys.path

- python
- import sys
- print(sys.path)

APPENDING PATH- append() is a built-in function of sys module that can be used with path variable to add a specific path for interpreter to search. The following example shows how this can be done.

- # importing module

- import sys
- # appending a path
- sys.path.append('D:\Notes_2021')
- # printing all paths
- sys.path

Temporary modification:

- **Changes to sys.path last only for the current session**
- **Order matters: Python uses the first matching module it finds in the path list**
- **Security: Be cautious when modifying sys.path as it affects module loading**

Best practice: For permanent additions, use:

- **Virtual environments**
- **PYTHONPATH environment variable**
- **Package installation with pip**

Built in modules

- modues that are the part of the language itself
- To name a few, Python contains modules like "os", "sys", "datetime", "random".
- You can import and use any of the built-in modules whenever you like in your program.

Absolute vs. Relative Imports in Python

- When importing modules in Python, you can use either absolute or relative paths to specify the module's location. This is particularly important when working with packages and complex project structures.

Absolute Imports

- Absolute imports specify the complete path to the module from your project's root directory or Python's environment.

Characteristics:

- Always start from the top-level package or Python path
- More explicit and generally preferred
- More portable if you move files around
- Easier to understand at a glance

Syntax Examples:

python

```python
# Importing standard library modules
import os
from sys import path
# Importing your own modules from package root
from mypackage import module1
from mypackage.subpackage import module2
# Importing specific functions/classes
from mypackage.module1 import my_function
```

When to Use:

- In most production code
- When modules might be run directly (as scripts)
- When clarity is more important than brevity

Relative Imports

- Relative imports specify the path to the module relative to the current module's location.

Characteristics:

- Use dots (.) to indicate relative position
- Shorter syntax within deep package structures
- Only work inside packages
- Can break if you reorganize your package structure

Syntax Examples:

python

```python
# Import from same directory
from . import sibling_module

# Import from parent package
from .. import parent_module
```

```
# Import from subpackage
from .subpackage import module


# Import specific function from sibling
from .sibling_module import some_function
```

When to Use:

- Within large packages with deep hierarchies
- When you want to emphasize intra-package relationships
- For internal package modules that won't be imported directly

User-defined Modules

1. **Create a Python file**: Save it with a .py extension

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


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


PI = 3.14159
```

Using a User-Defined Module

```
# main.py
import mymodule
print(mymodule.greet("Jyoti"))  # Output: Hello, Jyoti!
print(mymodule.add(5, 3))      # Output: 8
print(mymodule.PI)         # Output: 3.14159
```

Different Ways to Import

1. **Import entire module**:

import mymodule

2. **Import specific items**:

from mymodule import greet, add

print(greet("Moksh")) *# No need to use module prefix*

3. **Import with alias**:

import mymodule as mm

print(mm.greet("Preeti"))

4. **Import all names** (not recommended):

from mymodule import *

Advantages of modules

1. **Code Reusability**

- Write once, use many times across different programs
- Avoid rewriting the same code in multiple files
- Example: Create a math_operations.py module and reuse it in various projects

2. **Better Organization**

- Break large programs into logical, manageable components
- Group related functionality together
- Example: Separate database operations into db_utils.py and UI code into gui.py

3. **Namespace Separation**

- Avoid naming conflicts by scoping variables/functions to modules
- Example: math_utils.sqrt() vs. numpy.sqrt()

4. **Collaboration Efficiency**

- Different team members can work on separate modules simultaneously
- Clear boundaries between components
- Example: One developer works on data_processing.py while another works on report_generator.py

5. **Maintainability**

- Fix bugs or improve features in one centralized location
- Changes propagate to all programs using the module
- Example: Update currency conversion rates in finance.py once

6. **Performance Optimization**

- Python caches compiled modules (.pyc files)
- Subsequent imports are faster
- Example: Large modules like numpy load quicker after first import

7. **Information Hiding**

- Expose only what users need through selective imports
- Hide implementation details
- Example: from auth import login without exposing password hashing logic

8. **Testing Advantages**

- Test modules in isolation
- Mock dependencies easily
- Example: Test email_sender.py separately from main application

9. **Distribution/Sharing**

- Package related modules together for distribution
- Share via PyPI or internal repositories
- Example: Create company_utils package for internal use

10. **Documentation**

- Module-level docstrings provide natural documentation
- Tools like Sphinx generate documentation automatically
- Example: help(json) shows module documentation

The __name__ Variable in Python

The __name__ variable is a special built-in variable in Python that helps determine how a Python file is being executed. It serves two primary purposes:

1. Module Identification

When a Python file is imported as a module, __name__ is set to the module's name (the filename without the .py extension).

Example:

```
# mymodule.py
print(f"Module name: {__name__}")
# If imported in another file:
# Output: "Module name: mymodule"
```

2. Execution Context Detection

When a Python file is run directly (as the main program), __name__ is set to "__main__".

Example:

```
# myscript.py
print(f"Execution context: {__name__}")
# When run directly:
# Output: "Execution context: __main__"
```

Common Use Case:

The if __name__ == '__main__':

This pattern allows you to:

- Write code that executes when run directly
- Prevent code execution when imported as a module

```
# calculator.py
def add(a, b):
    return a + b


if __name__ == '__main__':
    # This only runs when executing the file directly
    print("Running in main mode")
    result = add(5, 3)
    print(f"5 + 3 = {result}")
```

Note:

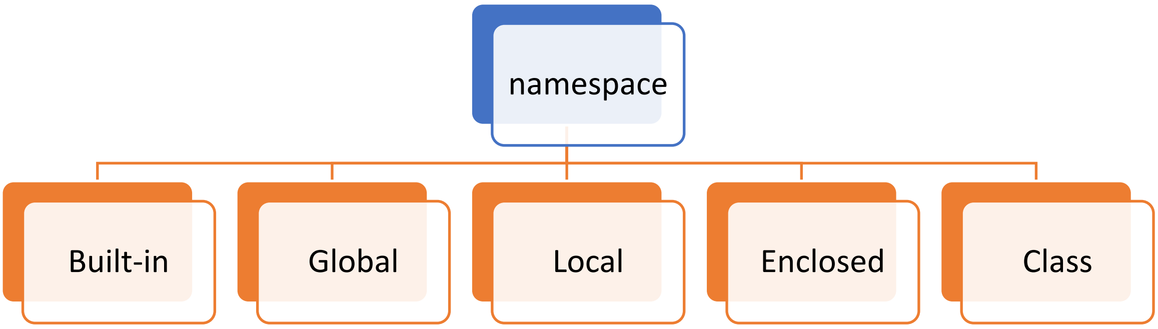
- **__name__ is automatically created for every Python module**

- **It changes value based on how the file is executed:**
  - **"__main__" when run directly**
  - **Module name when imported**
- **The if __name__ == '__main__': pattern is considered a best practice**

Namespaces in Python

- A namespace in Python is a system that ensures names (variables, functions, classes, etc.) are unique and can be used without conflict.
- It acts like a dictionary where names are mapped to their corresponding objects.

Types of Namespaces in Python



Python has different types of namespaces, each with its own scope and lifetime:

1. Built-in Namespace

- Contains all of Python's built-in functions (print(), len(), etc.) and exceptions
- Always available in every Python program
- Lives until the interpreter exits

print(len) *# <built-in function len>*

2. Global (Module) Namespace

- Contains module-level variables, functions, and classes
- Created when a module is imported
- Lasts until the module is unloaded

```
x = 10  # Global variable
def foo():
    pass  # Global function
```

3. Local (Function) Namespace

- Contains variables defined inside a function
- Created when a function is called
- Destroyed when the function exits

```
def bar():
    y = 20  # Local variable
    print(y)
```

4. Enclosed (Nonlocal) Namespace

- Exists in nested functions (closures)
- Contains variables from outer (but non-global) functions
- Accessed using nonlocal keyword

```
def outer():
    z = 30  # Enclosed namespace

    def inner():
        nonlocal z
        z += 1
        print(z)
    return inner
```

5. Class Namespace

- Contains class attributes and methods
- Created when a class is defined
- Exists until the class is garbage collected

```
class MyClass:
```

```
    class_var = 40  # Class namespace


    def method(self):
        return self.class_var
```

Namespace Resolution Order (LEGBC Rule)

Python searches names in this order:

1. **L**ocal (inside current function)
2. **E**nclosed (in nested functions)
3. **G**lobal (module level)
4. **B**uilt-in (Python's built-ins)
5. **C**lass (when in method)

```
x = "global"
class Test:
    x = "class"
        def method(self):
        x = "local"
        def inner():
            print(x)  # Which x?
        inner()
t = Test()
t.method()  # Output: "local"
```

Namespace Lifetime & Scope

- Lifetime: How long a namespace exists.
- Scope: Where a name can be accessed.

Modifying Namespaces

1. global Keyword → Modify a global variable inside a function.

```
count = 0
def increment():
```

```
    global count
    count += 1
increment()
print(count)  # 1
```

2. nonlocal Keyword → Modify a variable in an enclosing (non-global) scope.

```
def outer():
    x = 10
    def inner():
        nonlocal x
        x = 20
        inner()
    print(x)  # 20 (modified by inner)
outer()
```

Namespace	Scope	Lifetime	Access Method
Built-in	Everywhere	Until interpreter exits	Automatic
Global	Module-wide	Until module unloads	global keyword
Local	Function-only	During function execution	Automatic
Enclosed	Nested functions	While outer function runs	nonlocal keyword
Class	Class definition	Until garbage collected	self or ClassName

Packages

- In Python, a package is a way of organizing related modules into a directory hierarchy. It helps in structuring Python's module namespace using dotted notation (e.g., package.module).

Key Features of a Python Package:

Directory with __init__.py:

- A package is a directory that contains a special file named __init__.py (can be empty).
- This file indicates that the directory should be treated as a package.
- (In Python 3.3+, __init__.py is optional due to namespace packages.)

Hierarchical Structure:

- Packages can contain sub-packages and modules.
- Example:

```
my_package/
├── __init__.py
├── module1.py
└── subpackage/
    ├── __init__.py
    └── module2.py
```

Importing from Packages:

- Modules inside a package can be imported using dot notation.
- Example:
- import my_package.module1
- from my_package.subpackage import module2
- Each package in Python is a directory which MUST contain a special file called __init__.py. This file can be empty, and it indicates that the directory it contains is a Python package, so it can be imported the same way a module can be imported.

Purpose of Packages:

- **Avoid Naming Conflicts**: Different packages can have modules with the same name.
- **Improve Code Organization**: Group related functionality together.
- **Enable Reusability**: Distribute and reuse code easily (e.g., via PyPI).

Difference Between a Module and a Package:

- A **module** is a single .py file containing Python code.
- A **package** is a collection of modules in a directory.

In Python, __all__ is a special variable used in modules and packages to define their public interface - the list of names that should be imported when someone uses from module import *

Import the entire package

import package

 # Usage:

package.module.function()

Import specific module

from package import module

 # Usage:

module.function()

Import specific function/class

from package.module import function

 # Usage:

function()

2. Package Structure Example

```
my_package/
├── __init__.py
├── utils.py
└── core/
    ├── __init__.py
    ├── calculations.py
    └── validation.py
```

3. Importing from Subpackages

from my_package.core import calculations

from my_package.core.calculations import add_numbers

4. Using init.py for Cleaner Imports

 # my_package/core/__init__.py

from .calculations import add_numbers, multiply_numbers

from .validation import validate_input

__all__ = ['add_numbers', 'multiply_numbers', 'validate_input']

Now users can:

from my_package.core import add_numbers *# Instead of full path*

5. Relative Imports (within package)

Inside my_package/core/validation.py:

from .calculations import add_numbers *# Import from sibling module*

from ..utils import helper_function *# Import from parent package*

__init__.py file

The package folder contains a special file called __init__.py, which stores the package's content. It serves two purposes:

1. The Python interpreter recognizes a folder as the package if it contains __init__.py file.
2. __init__.py exposes specified resources from its modules to be imported.

An empty __init__.py file makes all functions from the above modules available when this package is imported. Note that __init__.py is essential for the folder to be recognized by Python as a package. You can optionally define functions from individual modules to be made available.

Example

```
#Project structure:
# finance/
# ├── __init__.py
# ├── math/
# │   ├── __init__.py
# │   └── compound.py
# └── utils.py
# finance/math/compound.py
def interest(principal, rate, years):
    return principal * (1 + rate) ** years
# finance/math/__init__.py
from .compound import interest
__all__ = ['interest']
# Usage:
from finance.math import interest
print(interest(1000, 0.05, 3))  # 1157.625
```

```
bank_operations/
├── __init__.py
├── customer_details.py
├── account_summary.py
├── withdraw.py
├── deposit.py
└── main.py
```

1. `customer_details.py`

python
```
# bank_operations/customer_details.py

customers = {}

def add_customer(customer_id, name, email, phone, initial_balance):
    """Add a new customer to the system"""
    if customer_id in customers:
        raise ValueError("Customer ID already exists")

    customers[customer_id] = {
        'name': name,
        'email': email,
        'phone': phone,
        'accounts': {
            'SAVINGS': initial_balance if initial_balance > 0 else 0
        }
    }
    return True

def get_customer_details(customer_id):
    """Retrieve customer details"""
    return customers.get(customer_id, None)

def update_customer_info(customer_id, **kwargs):
    """Update customer information"""
    if customer_id not in customers:
        raise ValueError("Customer not found")

    for key, value in kwargs.items():
        if key in customers[customer_id]:
            customers[customer_id][key] = value
    return True
```

2. `account_summary.py`

python
```
# bank_operations/account_summary.py
from .customer_details import customers

def get_account_balance(customer_id, account_type='SAVINGS'):
    """Get balance for a specific account"""
    customer = customers.get(customer_id)
    if not customer:
```

```python
        raise ValueError("Customer not found")

    return customer['accounts'].get(account_type, 0)

def get_all_accounts(customer_id):
    """Get all accounts for a customer"""
    customer = customers.get(customer_id)
    if not customer:
        raise ValueError("Customer not found")

    return customer['accounts']

def add_account(customer_id, account_type, initial_balance=0):
    """Add a new account for customer"""
    customer = customers.get(customer_id)
    if not customer:
        raise ValueError("Customer not found")

    if account_type in customer['accounts']:
        raise ValueError("Account type already exists")

    customer['accounts'][account_type] = initial_balance
    return True
```

3. `withdraw.py`

python

```python
# bank_operations/withdraw.py
from .account_summary import get_account_balance, update_account_balance

def withdraw_amount(customer_id, amount, account_type='SAVINGS'):
    """Withdraw amount from account"""
    balance = get_account_balance(customer_id, account_type)

    if balance < amount:
        raise ValueError("Insufficient balance")

    new_balance = balance - amount
    update_account_balance(customer_id, account_type, new_balance)

    return {
        'customer_id': customer_id,
        'account_type': account_type,
        'withdrawn_amount': amount,
        'new_balance': new_balance
    }
```

4. `deposit.py`

python
```
# bank_operations/deposit.py
from .account_summary import get_account_balance, update_account_balance

def deposit_amount(customer_id, amount, account_type='SAVINGS'):
    """Deposit amount to account"""
    balance = get_account_balance(customer_id, account_type)
    new_balance = balance + amount

    update_account_balance(customer_id, account_type, new_balance)

    return {
        'customer_id': customer_id,
        'account_type': account_type,
        'deposited_amount': amount,
        'new_balance': new_balance
    }
```

5. `__init__.py`

python
```
# bank_operations/__init__.py
from .customer_details import add_customer, get_customer_details, update_cu
stomer_info
from .account_summary import get_account_balance, get_all_accounts, add_acc
ount
from .withdraw import withdraw_amount
from .deposit import deposit_amount

__all__ = [
    'add_customer',
    'get_customer_details',
    'update_customer_info',
    'get_account_balance',
    'get_all_accounts',
    'add_account',
    'withdraw_amount',
    'deposit_amount'
]
```

6. `main.py` (Example Usage)

python
```
# bank_operations/main.py
```

```python
from bank_operations import *

# Add a new customer
add_customer('CUST001', 'John Doe', 'john@example.com', '1234567890', 5000)

# Deposit money
deposit_result = deposit_amount('CUST001', 2000)
print(f"Deposit successful. New balance: {deposit_result['new_balance']}")

# Withdraw money
withdraw_result = withdraw_amount('CUST001', 1000)
print(f"Withdrawal successful. New balance: {withdraw_result['new_balance']
}")

# Get account summary
balance = get_account_balance('CUST001')
print(f"Current balance: {balance}")

# Get customer details
customer = get_customer_details('CUST001')
print(f"Customer details: {customer}")
```

Short Questions:

1. What is a Python module?
2. What is the file extension of a Python module
3. State two advantages of using modules in Python.
4. Name any two third-party modules and their common uses.
5. What does sys.path contain in Python?
6. Explain the purpose of sys.path.append() with an example.
7. What happens if you modify sys.path temporarily?
8. Write examples of both absolute and relative import syntax.
9. What is the value of __name__ when a module is run directly?
10. Why is if __name__ == "__main__" considered a best practice?
11. What is the purpose of the __init__.py file in a package?
12. How do you import a function from a subpackage in Python

Long question:

1. Give an example of a user-defined module and how to import it.
2. Differentiate between built-in, user-defined, and third-party modules with examples.
3. What is the difference between absolute and relative imports in Python?
4. List any five advantages of using modules in Python.
5. How do modules improve collaboration and maintainability?
6. Explain the concept of namespace separation with an example.
7. What are the different types of namespaces in Python?

8. Explain the LEGB rule of name resolution with an example.
9. How does the nonlocal keyword help in modifying enclosed variables?
10. Create a sample package structure with submodules and demonstrate how to use relative import.
11. What is the role of __all__ in a module or package?