Modules and Packages

A module is simply a Python file (.py) that contains Python code, such as functions, classes, or variables. Modules help organize code into reusable components, promoting maintainability and modularity.

Key Concepts of Modules:

1. **Creating a Module**: Any Python file with .py extension is a module.

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

2. **Importing Modules**: You can import and use functions from a module.

```python
import mymodule
print(mymodule.greet("Anand"))  # Output: Hello, Anand!
```

3. **Selective Imports**: Import only specific functions or variables from a module.

```
from mymodule import greet
print(greet("Anand"))  # Output: Hello, Anand!
```

4. **Alias for Modules**: You can provide an alias for a module name using as

```
import mymodule as mm
print(mm.greet("Anand"))  # Output: Hello, Anand!
```

5. **Import All (*)**: Import all functions or variables from a module (not recommended due to possible name clashes).

```
from mymodule import *
print(greet("Anand"))  # Output: Hello, Anand!
```

6. **Checking Module Content**: You can use dir() to list all functions and variables defined in a module.

```
import math
print(dir(math))  # Output: ['__doc__', '__loader__', '__name__', ..., 'sqrt', 'tan'
```

7. **Built-in Modules**: Python has several built-in modules, such as math, datetime, os, and sys. These are available without needing to install them.

```
import math
print(math.sqrt(16))  # Output: 4.0
```

8. **Third-party Modules**: You can install external libraries/modules using pip. For example, installing and using the requests module:

```
pip install requests
```

```python
import requests
response = requests.get("https://api.github.com")
print(response.status_code)  # Output: 200
```

Packages in Python

A **package** is a collection of related modules grouped in a directory. Packages are directories containing a special __init__.py file (which can be empty) and one or more modules.

Key Concepts of Packages:

1. **Creating a Package**:
   - A package is simply a directory that contains modules and an __init__.py file.
   - The __init__.py file tells Python that the directory should be treated as a package.

```
mypackage/
├── __init__.py
├── module1.py
└── module2.py
```

Example of `module1.py`:

```python
def hello():
    return "Hello from module1"
```

2. **Importing from a Package**:

- Once you have a package, you can import specific modules from it.

```
import mypackage.module1
print(mypackage.module1.hello())  # Output: Hello from module1
```

3. **Selective Import from a Package**:

- You can import specific functions or classes from a module within a package.

```
from mypackage.module1 import hello
print(hello())  # Output: Hello from module1
```

4. **Nested Packages**:

- Packages can contain sub-packages (directories with their own __init__.py file), enabling a deep hierarchical structure.

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

Importing from a subpackage:

```python
import mypackage.subpackage.module2
```

5. **Installing Packages**:

- Packages are often installed from the Python Package Index (PyPI) using pip. Example of installing a third-party package:

```
pip install numpy
```

Example of using a package after installation:

```python
import numpy as np
arr = np.array([1, 2, 3, 4])
print(arr)  # Output: [1 2 3 4]
```

6. **Absolute vs Relative Imports**:

- **Absolute Imports**: Import modules by specifying their full path.

```
# module1.py
from mypackage.subpackage.module2 import func
```

- **Relative Imports**: Use dots (.) to import relative to the current module.

```
# module1.py
from .subpackage import module2
```

Modules vs. Packages

- **Modules**: Individual Python files containing functions, classes, or variables.
- **Packages**: Directories containing multiple modules (and potentially sub-packages) organized for better structure.