Prepared by: **Dhananjay Kumar**
Trainer | Java Full-Stack & Python | Contact: 7023577968
LinkedIn: Dhananjay Kumar

Python Modules — Notes Examples

Table of Contents

---

1. Creating Modules

Concept (simple): A *module* in Python is a file containing Python definitions and statements (functions, classes, variables). Any `.py` file is a module. We create modules to organize code and reuse functionality across projects.

Why modules?

- Reusability: write once, import many times.
- Readability: split large programs into smaller, logical files.
- Namespace separation: prevents name collisions.

Example 1 — `math_utils.py` (simple function module)

```python
# math_utils.py

def add(a, b):
    """Return sum of a and b."""
    return a + b

def multiply(a, b):
    """Return product of a and b."""
    return a * b
```

Explanation: Save this file as `math_utils.py`. It exposes two functions you can import in other scripts.

Example 2 — `greetings.py` (module with constants)

```python
# greetings.py

WELCOME = "Hello"

def greet(name):
    return f"{WELCOME}, {name}!"
```

Explanation: This module shows a constant and a function. Constants are just variables by convention (caps).

Example 3 — Module with a small class: `counter.py`

```python
# counter.py

class Counter:
    def __init__(self, start=0):
        self.value = start

    def inc(self):
        self.value += 1
        return self.value

    def reset(self):
        self.value = 0
```

Explanation: Modules can export classes. You can import `Counter` where needed.

Example 4 — Module with internal helper (private by convention)

```
# shapes.py

def _area_square(side):
    return side * side

def area(shape, size):
    if shape == 'square':
        return _area_square(size)
    raise ValueError('unsupported shape')
```

Explanation: Names starting with `_` are considered internal implementation details (not enforced but conventional).

Example 5 — Packaging small utilities in `__all__`

```
# colours.py
__all__ = ['primary', 'secondary']

primary = ['red', 'green', 'blue']
secondary = ['orange', 'violet', 'cyan']
_hidden = 'not exported'
```

Explanation: `__all__` controls what `from colours import *` will import. It is a good practice in public modules.

---

2. Variables in Modules

Concept: Module-level variables are created when the module is first imported. They persist as attributes of the module object and can act like singletons (shared state).

Example 1 — Module-level counter

```
# modstate.py
COUNTER = 0

def inc():
    global COUNTER
    COUNTER += 1
    return COUNTER
```

Explanation: `COUNTER` lives inside the module; every import of `modstate` refers to the same `COUNTER` object.

Example 2 — Mutable default (list) as a module variable

```
# favourites.py
favs = []

def add(item):
    favs.append(item)

def show():
    return list(favs)
```

Explanation: Be careful: mutable module variables are shared and can be changed by any code importing the module.

Example 3 — Constants and naming convention

```
# config.py
DB_HOST = 'localhost'
DB_PORT = 3306

# Usage: from config import DB_HOST
```

Explanation: Using ALL_CAPS indicates the value is a constant (by convention).

Example 4 — Lazy initialization inside module

```
# db.py
_connection = None

def get_connection():
    global _connection
    if _connection is None:
        # pretend to create a connection
        _connection = {'connected': True}
    return _connection
```

Explanation: Module variables are useful for caching resources like DB connections.

Example 5 — Using module variables to track state across imports

```
# visits.py
visits = {
    'home': 0,
    'about': 0,
}

def visit(page):
    if page in visits:
        visits[page] += 1
    else:
        visits[page] = 1

def stats():
    return dict(visits)
```

Explanation: Modules are often used to collect simple shared state; thread-safety is a separate concern.

---

3. Imports and Attributes

Concept: Import statements bring module objects or attributes into the current namespace. Common forms:

- `import module`
- `import module as m`
- `from module import name`
- `from module import *` (not recommended for large code)

Access attributes using dot notation: `module.attr`.

Example 1 — Basic import

```
# assume math_utils.py exists
import math_utils

print(math_utils.add(2, 3))  # 5
```

Explanation: `math_utils` is a module object. Functions inside are attributes.

Example 2 — Import specific names

```
from math_utils import multiply

print(multiply(4, 5))
```

Explanation: Only `multiply` is imported into local namespace.

Example 3 — Import with alias

```
import math_utils as mu
print(mu.multiply(3, 3))
```

Explanation: Aliasing keeps things short and readable.

Example 4 — Inspecting module attributes

```
import greetings
print(dir(greetings))
print(greetings.WELCOME)
print(greetings.greet('Ankita'))
```

Explanation: `dir()` helps you see names defined in a module. `module.__dict__` contains the actual namespace mapping.

Example 5 — Avoid `from module import *`

```
# bad_practice.py
from colours import *  # brings primary, secondary into current namespace

print(primary)
```

Explanation: Using `*` may cause name clashes and makes the origin unclear. Prefer explicit imports.

---

4. Namespaces

Concept (simple): A *namespace* is a mapping from names to objects. Modules provide their own namespaces. Local, global, builtins are other namespaces. Understanding namespaces avoids confusion about where a name is resolved.

Example 1 — Module vs local namespace

```
# names.py
x = 'module x'

def show():
    x = 'local x'
    print('inside function:', x)

show()
print('module level:', x)
```

Explanation: Local `x` shadows module-level `x` inside function.

Example 2 — Using `global` to modify module variable

```
# names2.py
count = 0

def increase():
    global count
    count += 1
```

Explanation: `global` refers to the module-level name, not the builtins.

Example 3 — Builtins namespace

```
# builtins_example.py
import builtins
print('has print?', hasattr(builtins, 'print'))
```

Explanation: Python looks for names in local, then global (module), then builtins.

Example 4 — Avoiding namespace collision with aliases

```
import datetime as dt
from datetime import datetime as dt_now

print(dt.date.today())
print(dt_now.now())
```

Explanation: Good aliasing clarifies which object you refer to.

Example 5 — Nested namespaces with classes

```
# person.py
class Person:
    species = 'Homo sapiens'  # class namespace

    def __init__(self, name):
        self.name = name  # instance namespace

p = Person('Dhananjay')
print(Person.species)
print(p.name)
```

Explanation: Class, instance, module are different namespaces.

---

5. Reloading Modules

Concept: When you import a module, Python caches it in `sys.modules`. During development, if you modify the module file and want to pick up changes in the same interpreter session, you must reload it. Use `importlib.reload()` in Python 3.

Warning: Reloading has limitations: existing references to objects from the old module remain unchanged. Prefer restarting the interpreter for complex changes.

Example 1 — Basic reload

```
# interactive or notebook cell
import importlib
import greetings

# change greetings.py on disk now (manually), then:
importlib.reload(greetings)
```

Explanation: `reload()` re-executes the module code and updates the module object.

Example 2 — Demonstration with state

```
# modstate.py
COUNTER = 0

def inc():
    global COUNTER
    COUNTER += 1
    return COUNTER
import importlib
import modstate
print(modstate.inc())  # 1
# edit modstate.py to change starting COUNTER or functions, then:
importlib.reload(modstate)
```

Explanation: Reloading updates module attributes but **existing references** to `modstate.inc` from before reload may still point to the old function.

Example 3 — Reloading specific submodule in a package

```
# package structure:
# mypkg/
#   __init__.py
#   a.py

import importlib
import mypkg.a
importlib.reload(mypkg.a)
```

Explanation: Reload the exact module you imported; reloading package doesn't necessarily reload submodules.

Example 4 — Practical pattern in notebooks

```
# In notebook during development:
import importlib
import myutils
# After editing myutils.py on disk:
myutils = importlib.reload(myutils)
```

Explanation: Assigning the return value of `reload()` to the name helps ensure the top-level reference points to the updated module.

Example 5 — Limitations example

```
# Suppose:
from mylib import helper
# later file changed and reloaded via importlib.reload(mylib),
# but `helper` still binds to old function object.
```

Explanation: `from module import name` copies the object into current namespace; reloading the module doesn't update that copied reference.

---

6. Built-in Modules

Concept: Python ships with a rich standard library. Built-in modules include `os`, `sys`, `math`, `datetime`, `json`, and many more.

Example 1 — `os` for file operations

```
import os
print('current dir:', os.getcwd())
print('list files:', os.listdir('.'))
```

Explanation: `os` helps interact with the operating system in a portable way.

Example 2 — `sys` for interpreter info

```
import sys
print('python version:', sys.version)
print('module search path:', sys.path)
```

Explanation: `sys.path` determines where Python looks for modules.

Example 3 — `math` for math helpers

```
import math
print(math.sqrt(16))
```

```
print(math.pi)
```
Explanation: Use `math` when you need numerical functions.

Example 4 — `datetime` for dates and times

```
from datetime import datetime, timedelta
now = datetime.now()
print('now:', now)
print('tomorrow:', now + timedelta(days=1))
```
Explanation: `datetime` is the go-to for date/time operations.

Example 5 — `json` for serialization

```
import json
obj = {'name': 'Dhananjay', 'items': [1, 2, 3]}
text = json.dumps(obj)
print(text)
print(json.loads(text))
```
Explanation: `json` converts Python objects to JSON strings and back — useful for APIs and storage.

---

7. Generating Random Values

Concept: The `random` module generates pseudo-random numbers. For cryptographic randomness, use `secrets` (not covered deeply here). Typical functions: `random()`, `randint()`, `choice()`, `shuffle()`, `sample()`.

Example 1 — `random.random()` and `random.randint()`

```
import random
print(random.random())  # float in [0.0, 1.0)
print(random.randint(1, 6))  # simulated dice: 1..6 inclusive
```
Explanation: `random()` gives a float; `randint(a,b)` includes both endpoints.

Example 2 — Choosing randomly from a list

```
names = ['Ankita', 'Mona', 'Shilpa', 'Ravi']
print(random.choice(names))
```
Explanation: `choice()` picks a single random element.

Example 3 — Shuffling and sampling

```
cards = list(range(1, 53))
random.shuffle(cards)
hand = random.sample(cards, 5)
print('hand:', hand)
```
Explanation: `shuffle()` mutates the list, `sample()` returns k unique choices without replacement.

Example 4 — Seeding for reproducibility

```
random.seed(42)
print([random.randint(1, 100) for _ in range(5)])
# reseed to get same sequence
random.seed(42)
print([random.randint(1, 100) for _ in range(5)])
```
Explanation: `seed()` makes pseudo-random sequences reproducible — useful in testing and examples.

Example 5 — Using `secrets` for secure tokens

```
import secrets
token = secrets.token_hex(8)  # 16 hex chars
print('secure token:', token)
```
Explanation: For anything security-sensitive (passwords, tokens), use `secrets`, not `random`.

Exercises (practice for students)

1. Create a module `string_stats.py` with functions: `count_vowels(s)`, `count_consonants(s)`, `most_common_char(s)`; then import and test.
2. Build a module `simpledb.py` that simulates a tiny in-memory DB using a module-level dict with `insert`, `get`, `delete` functions.
3. Experiment: create `a.py` with a variable `X=1`; import `X` into another script using `from a import X`; then change `a.X` and

reload — observe what happens.

4. Write a small script using `random.seed()` to generate reproducible sequences for unit tests.
5. Use `os` and `json` to write module results to a file and read them back.

End of notes.

Python Modules: Practice Questions & MCQs

Section 1 — Quick Reference

Creating a module

Create a file `mymodule.py` and put functions or variables there. Example:

```python
# mymodule.py
PI = 3.14159

def area_circle(r):
    return PI * r * r

class Greeter:
    def greet(self, name):
        return f"Hello, {name}!"
```

Importing

```python
import mymodule
from mymodule import area_circle, PI
import mymodule as mm
```

Reloading (during development)

```python
import importlib
importlib.reload(mymodule)
```

Namespaces

- Each module has its own global namespace. `mymodule.PI` and `PI` in your current script are different unless you `from mymodule import PI`.

Useful built-in modules

- `math` — math functions and constants
- `sys` — interpreter info
- `random` — random numbers
- `os` — filesystem operations

Section 2 — Practice Questions (10)

> Each practice question contains: Scenario, Task, Code, and Explanation.

Question 1 — Create & Use a Simple Module

Scenario: You want a reusable utility that calculates area and circumference of a circle.

Task: Create a module `circle_utils.py` with constants and functions. Then import and use it in another script.

Code:

```python
# File: circle_utils.py
PI = 3.141592653589793

def area(radius):
    return PI * radius * radius

def circumference(radius):
    return 2 * PI * radius
```

```python
# File: use_circle.py
import circle_utils

r = 5
print("Area:", circle_utils.area(r))
print("Circumference:", circle_utils.circumference(r))
```

Explanation:

- `circle_utils.py` defines functions and a constant. When you `import circle_utils` in `use_circle.py`, you access them as attributes of the module: `circle_utils.area`.

Question 2 — Module Variables & Mutability

Scenario: You have a module `config.py` with a default setting. You want to update it at runtime.

Task: Show that modifying a module variable in one module affects other modules that import the module object (but not those that used `from ... import ...`).

Code:

```python
# config.py
DEFAULT_LANG = 'en'
# app1.py
import config
config.DEFAULT_LANG = 'hi'
print("app1 sees", config.DEFAULT_LANG)

# app2.py
from config import DEFAULT_LANG
print("app2 sees", DEFAULT_LANG)
```

Explanation:

- `app1.py` imports the module object `config` and mutates `config.DEFAULT_LANG`. Any module that imports `config` and accesses `config.DEFAULT_LANG` will see the updated value.
- `app2.py` used `from config import DEFAULT_LANG` which copies the value into the local namespace at import time — it won't reflect later changes to `config.DEFAULT_LANG`.

Question 3 — `from ... import` vs `import` namespace clarity

Scenario: Two functions named `connect` exist in different modules `db_mysql.py` and `db_sqlite.py`. You must avoid name collisions.

Task: Demonstrate safe importing so names don't clash.

Code:

```python
# db_mysql.py

def connect():
    return "Connected to MySQL"

# db_sqlite.py

def connect():
    return "Connected to SQLite"
# main.py
import db_mysql
import db_sqlite

print(db_mysql.connect())
print(db_sqlite.connect())

# OR using aliasing
from db_mysql import connect as mysql_connect
from db_sqlite import connect as sqlite_connect
print(mysql_connect())
print(sqlite_connect())
```

Explanation:

- Using `import module` keeps functions under the module namespace. Use `as` to alias imported names to avoid collisions.

Question 4 — Building a Package (simple)

Scenario: You want to group related modules under a package `shapes`.

Task: Create a package with `__init__.py`, `circle.py`, and `rectangle.py`. Import `shapes.circle`.

Code:

```
   # Directory structure:
   # shapes/
   #   __init__.py
   #   circle.py
   #   rectangle.py

# shapes/circle.py
PI = 3.14159

def area(r):
    return PI * r * r

# shapes/__init__.py
from .circle import area as circle_area

# usage
import shapes
print(shapes.circle_area(3))
```

Explanation:

- Packages let you organize modules. `__init__.py` can expose convenient names for users.

Question 5 — Reload a module while developing

Scenario: You're testing functions in `calc.py` interactively. You change a function and want Python to pick up the new code without restarting the interpreter.

Task: Show how to use `importlib.reload`.

Code:

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

# interactive session or script
import calc
print(calc.add(2,3))  # 5

# edit calc.py, change add to return a + b + 1

import importlib
importlib.reload(calc)
print(calc.add(2,3))  # now picks up new behavior
```

Explanation:

- `importlib.reload(module)` re-executes the module code and updates the module object. Be careful: existing references to old functions or objects need to be refreshed.

Question 6 — Using built-in `random` to simulate dice rolls

Scenario: Create a small utility module `dice.py` to simulate rolling N six-sided dice and returning the sum.

Task: Implement the function and show example use.

Code:

```
# dice.py
import random

def roll_one_die():
    return random.randint(1, 6)

def roll_n_dice(n):
    return sum(roll_one_die() for _ in range(n))

# use_dice.py
import dice
print("Roll 3 dice:", dice.roll_n_dice(3))
```

Explanation:

- `random.randint(a,b)` returns a random integer in `[a, b]` inclusive. The module encapsulates the behavior so it can be reused.

Question 7 — Module attributes and `__name__` guard

Scenario: Make a module `greet.py` that can be used as a script or imported as a module.

Task: Use `if __name__ == '__main__'` to provide example behavior when run directly.

Code:

```
# greet.py

def say_hi(name):
    return f"Hi, {name}!"

if __name__ == '__main__':
    # this block runs only when greet.py is executed directly
    print(say_hi('Student'))
# another file
import greet
print(greet.say_hi('Dhananjay'))
```

Explanation:

- `__name__` is set to `'__main__'` when a module is executed as a script; otherwise it's the module name. This idiom keeps demo code out of imports.

Question 8 — Namespaces: Demonstrate local vs global vs module-level

Scenario: Illustrate variable lookup order and show that module-level variables are different from local function variables.

Task: Create `ns_demo.py` and show examples.

Code:

```
# ns_demo.py
val = 'module-level'

def show():
    val = 'local'
    print('inside function val =', val)    # local
    print('module val =', globals()['val'])

show()
print('outside function val =', val)
```

Explanation:

- `globals()` returns the module's global namespace. Function-local variables shadow module globals.

Question 9 — Use `math` and `random` together: random points in a circle

Scenario: You want to randomly sample a point uniformly inside a circle of radius R.

Task: Write a function `random_point_in_circle(R)` that returns `(x,y)`.

Code:

```
import math
import random

def random_point_in_circle(R):
    # polar coordinates with sqrt for uniform radius distribution
    r = R * math.sqrt(random.random())
    theta = 2 * math.pi * random.random()
    x = r * math.cos(theta)
    y = r * math.sin(theta)
    return x, y

# example
print(random_point_in_circle(5))
```

Explanation:

- To sample uniformly in a disk, radius must be sampled proportional to sqrt(U) where U~Uniform(0,1). Sampling r uniformly would bias towards the edge.

Question 10 — Build a small CLI module and import it

Scenario: You want a helper module `cli_tools.py` that provides a `confirm()` function to ask yes/no from the user; you want to import and reuse it across scripts.

Task: Implement `confirm()` and show safe importing.

Code:

```
# cli_tools.py

def confirm(prompt='Are you sure? (y/n): '):
    ans = input(prompt).strip().lower()
    return ans in ('y', 'yes')

# main_app.py
from cli_tools import confirm

if confirm('Delete all data? (y/n): '):
    print('Proceeding with deletion...')
else:
    print('Aborted.')
```

Explanation:

- Packaging common interactive helpers into modules avoids code duplication and improves testability (you can mock `input` in tests).

Section 3 — 20 MCQs (Multiple Choice Questions)

Instructions: For each question choose the best option. Answers and short explanations follow each question.

1. **Which statement creates a module object and does not copy names into the local namespace?**

A. `from math import sqrt` B. `import math` C. `from math import *` D. `import sqrt from math`

Answer: B

Explanation: `import math` binds the module object `math` in the local namespace; names stay under `math.` prefix. `from math import sqrt` copies name `sqrt` directly.

2. **What is the value of `__name__` inside a module when it is imported?**

A. `'__main__'` B. `''` C. The module's filename without `.py` D. `None`

Answer: C

Explanation: When imported, `__name__` is the module's name (usually the filename without `.py`). When run directly, it's `'__main__'`.

3. **What does `importlib.reload(module)` do?**

A. Deletes the module from memory B. Re-executes the module code and updates the module object C. Creates a copy of the module D. Converts the module into a package

Answer: B

Explanation: `reload` re-executes the module's code in its existing module object. Existing references to old objects may remain unchanged.

4. **Which of these is the correct way to get a random floating-point number in `[0.0, 1.0)`?**

A. `random.uniform(0,1)` B. `random.random()` C. `random.randint(0,1)` D. `random.choice([0,1])`

Answer: B

Explanation: `random.random()` returns a float in `[0.0, 1.0)`. `random.uniform(0,1)` also works but returns floats in [0,1] inclusive for endpoints depending on implementation.

5. **If module `a` imports module `b`, and then `a` is imported by `c`, which of these is true?**

A. `b` will not be imported unless `c` explicitly imports `b` B. `b` will be imported when `a` is imported (only once) C. `b` is copied into `c`'s namespace automatically D. `b` is garbage-collected immediately

Answer: B

Explanation: Importing `a` executes `a`'s import statements; `b` will be imported as part of `a` and cached in `sys.modules`.

6. **Which import avoids polluting the local namespace but allows shorter access names?**

A. `from module import *` B. `from module import func` C. `import module as mod` D. `import func as f`

Answer: C

Explanation: `import module as mod` keeps names under `mod.` but shortens the module name.

7. **What will `random.randint(1, 6)` return?**

A. A float between 1 and 6 B. An integer from 1 to 6 inclusive C. An integer from 1 to 5 inclusive D. Always 3

Answer: B

Explanation: `random.randint(a,b)` returns an integer N such that `a <= N <= b`.

8. **Which of the following is true about module-level variables?**

A. They are inaccessible outside the module. B. They can be changed by other modules that import the module object. C. They are always immutable. D. They are copied to every module that imports them and changes won't reflect.

Answer: B

Explanation: Module globals can be mutated; other modules that reference the module object see the change.

9. **What is the recommended way to expose only some names from a module when using `from module import *`?**

A. Use `__all__` list inside the module. B. Use `hidden_` prefix in names. C. Remove names at runtime. D. It's not possible.

Answer: A

Explanation: Defining `__all__ = ['name1','name2']` controls what `from module import *` imports.

10. **Why use `if __name__ == '__main__'` in a module?**

A. To make the module private B. To run demo or test code only when executed as a script C. To import the module faster D. To hide sensitive information

Answer: B

Explanation: It prevents demo code from running on import.

11. **Which of these statements about packages is correct?**

A. Packages are directories with a special `__init__.py` file (Python 3.3+ may treat directories without it as namespace packages). B. Packages must be single files only. C. Packages cannot contain subpackages. D. Packages are not supported in Python.

Answer: A

Explanation: Packages group modules in a directory; `__init__.py` makes a regular package; Python also supports namespace packages.

12. **Which function from `random` will randomly pick an item from a list?**

A. `random.sample()` B. `random.choice()` C. `random.pick()` D. `random.shuffle()`

Answer: B

Explanation: `random.choice(seq)` returns a single random element from `seq`.

13. **What does `from package.module import func` do?**

A. Imports the package, module, and binds `func` into your namespace. B. Only imports the function source code text. C. Deletes the package after import. D. Imports the module but not the package.

Answer: A

Explanation: Python imports the package and module as needed and binds the name `func` in your namespace.

14. **Which of these is a safe way to avoid circular imports?**

A. Restructure code so common utilities go to a third module. B. Import inside functions (local import) C. Merge modules carefully. D. All of the above.

Answer: D

Explanation: All strategies can help avoid circular import problems.

15. **Which `random` function returns k unique items from a population?**

A. `random.choices(population, k)` B. `random.sample(population, k)` C. `random.choice(population)` D. `random.shuffle(population)`

Answer: B

Explanation: `random.sample` returns unique items without replacement. `choices` allows replacement.

16. **What happens when you `import math` twice in the same interpreter session?**

A. Module is loaded twice in memory B. The second import is ignored because module is loaded and cached in `sys.modules` C. Raises an ImportError D. Deletes the module

Answer: B

Explanation: Imports are cached; subsequent imports reuse the same module object.

17. **Which is the correct way to access a module's docstring programmatically?**

A. `module.__doc__` B. `module.__doc` C. `module.doc` D. `help(module)` only

Answer: A

Explanation: `__doc__` is the module docstring attribute.

18. **Which of the following returns a list of names defined in a module?**

A. `dir(module)` B. `names(module)` C. `list(module)` D. `vars(module)`

Answer: A

Explanation: `dir(module)` lists names; `vars(module)` returns the `__dict__` mapping (also useful).

19. **Which `random` function will return integers with a specified step, e.g., 0, 5, 10?**

A. `random.randrange(0, 11, 5)` B. `random.randint(0,10)` C. `random.choice(range(0,11,5))` D. Both A and C

Answer: D

Explanation: `randrange(0,11,5)` generates values with step 5; `choice(range(...))` also works.

20. **When you do `from module import *`, which names are NOT imported?**

A. Names starting with `_` B. Names in `__all__` C. Functions only D. Variables only

Answer: A

Explanation: By default, names starting with underscore are omitted. `__all__` can override what gets imported.

In [ ]: