Chapter 11.3 – Importing Specific Names from a Module

Teacher Edition with Full Solutions

1 Learning Objectives

- Import only specific functions or variables from a module.
- Use aliases to simplify long module names.
- Understand the benefits and risks of using the wildcard * import.

2 Introduction

Sometimes you don't want to import an entire module—just one or two functions. Python allows you to import specific names directly using the syntax:

```
from module_name import function_name
```

You can also import multiple items:

```
from module_name import func1, func2
```

And for convenience:

```
from module_name import function_name as shortname
```

—

Example – Using math Functions

```
from math import sqrt, pi

print("Square root of 16:", sqrt(16))
print("Area of circle radius 2:", pi * (2 ** 2))
```

Output:

```
Square root of 16: 4.0
Area of circle radius 2: 12.566370614359172
```

Instructor Notes – Key Point

Encourage students to recognize readability trade-offs: Explicit imports make code more readable, but too many can clutter the namespace. Contrast this with importing the full module and using prefixes.

3 Using Aliases

You can rename imported items or modules for convenience:

```python
import math as m

print(m.sqrt(81))
```

or:

```python
from math import factorial as f
print(f(5))
```

Instructor Notes – Teaching Emphasis

Aliases improve readability when module names are long, like `numpy` → `np`. Have students reflect on readability vs clarity.

4 Wildcard Imports – Use With Caution

You can import everything with:

```python
from math import *
```

This loads all public functions and variables into your current namespace. However, it makes it hard to tell where names came from and can cause conflicts.

Activity – Namespace Chaos

Try running this experiment:

```python
from math import *
from random import *

print(sin(0))     # math.sin
print(random())   # random.random
```

Now define your own `sin()` function below and see what happens!

```python
def sin(x):
    return "This is not math.sin!"
print(sin(0))
```

Expected Output:

```
0.0
0.3748298023
```

```
This is not math.sin!
```

Instructor Notes – Discussion Prompt

Ask: "Why is this dangerous?" Students will see how one small override can break math functions silently. Encourage the use of explicit imports or aliases instead.

5 Mini Challenge – Custom Utility Module

Student Challenge

Step 1: Create a module named `converter.py`:

```
def c_to_f(c):
    return (c * 9/5) + 32

def f_to_c(f):
    return (f - 32) * 5/9
```

Step 2: In another file `main.py`, import specific functions:

```
from converter import c_to_f, f_to_c

print("0 C  =", c_to_f(0), " F ")
print("212 F  =", f_to_c(212), " C ")
```

Expected Output:

```
0 C  = 32.0  F
212 F  = 100.0  C
```

Instructor Notes – Instructor Solution Notes

Show students that importing specific functions keeps the code lightweight. Contrast this with importing the whole module and calling `converter.c_to_f(0)`.

6 Reflection – Why This Matters

Core Takeaways

- Specific imports keep code concise and readable.
- Wildcard imports can create confusion and bugs.
- Aliases simplify long names and improve code style.

Instructor Notes – Wrap-Up Discussion

Ask students to compare the following two lines:

```
from math import sqrt
import math
```

Which is clearer when debugging or sharing code? Let the class debate and justify their preferences.