Lecture 7: Modules

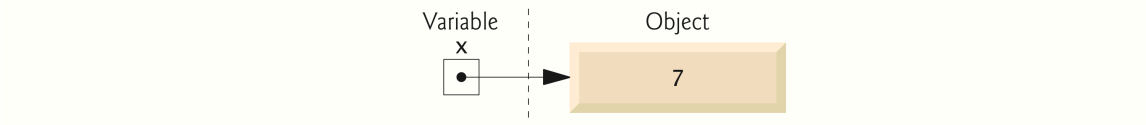
INF 605 · 2026-09-17

Open in Colab → **File → Save a copy in Drive**. You cannot edit the class original.

Start-of-class quiz

Type answers in the Canvas attendance quiz (empty boxes only). The questions stay here.

1. Last class was **Functions**. What keyword sends a value back from a function?
2. What happens if you call a function that has a default parameter and you omit that argument?



4.6 Python Standard Library

- You write Python programs by combining functions and classes (that is, custom types) that you create with preexisting functions and classes defined in modules, such as those in the Python Standard Library and other libraries.
- Avoid “reinventing the wheel.”
- A module is a file that groups related functions, data and classes.
- A **package** groups related modules.
- The Python Standard Library is provided with the core Python language.
- Its packages and modules contain capabilities for a wide variety of everyday programming tasks.

| Some popular Python Standard Library modules | ---------- | `collections` —Data structures beyond lists, tuples, dictionaries and sets. | Cryptography modules—Encrypting data for secure transmission. | `csv` —Processing comma-separated value files (like those in Excel). | `datetime` —Date and time manipulations. Also modules `time` and `calendar`. | `decimal` —Fixed-point and floating-point arithmetic, including monetary calculations. | `doctest` —Embed validation tests and expected results in docstrings for simple unit testing. | `gettext` and `locale` —Internationalization and localization modules. | `json` —JavaScript Object Notation (JSON) processing used with web services and NoSQL document databases. |

`math` —Common math constants and operations. | `os` —Interacting with the operating system. | `profile`, `pstats`, `timeit` —Performance analysis. | `random` —Pseudorandom numbers. | `re` —Regular expressions for pattern matching. | `sqlite3` —SQLite relational database access. | `statistics` —Mathematical statistics functions such as `mean`, `median`, `mode` and `variance`. | `string` —String processing. | `sys` —Command-line argument processing; standard input, standard output and standard error streams. | `tkinter` —Graphical user interfaces (GUIs) and canvas-based graphics. | `turtle` —Turtle graphics. | `webbrowser` —For conveniently displaying web pages in Python apps.

4.7 `math` Module Functions

- The **`math` module** defines functions for performing various common mathematical calculations.

```
In [1]: import math
```

The next cell runs `math.sqrt(900)`.

```
In [2]: math.sqrt(900)
```

```
Out[2]: 30.0
```

The next cell runs `math.fabs(-10)`.

```
In [3]: math.fabs(-10)
```

```
Out[3]: 10.0
```

The next cell runs `abs(-10)`.

```
In [4]: abs(-10)
```

```
Out[4]: 10
```

- Some `math` module functions are summarized below
- View the complete list

Function	Description	Example
`ceil( x )`	Rounds x to the smallest integer not less than x	`ceil(9.2) is 10.0`, `ceil(-9.8) is -9.0`
`floor( x )`	Rounds x to the largest integer not greater than x	`floor(9.2) is 9.0`, `floor(-9.8) is -10.0`

Function	Description	Example
`sin( x )`	Trigonometric sine of x (x in radians)	`sin(0.0) is 0.0`
`cos( x )`	Trigonometric cosine of x (x in radians)	`cos(0.0) is 1.0`
`tan( x )`	Trigonometric tangent of x (x in radians)	`tan(0.0) is 0.0`
`exp( x )`	Exponential function ex	`exp(1.0) is 2.718282`, `exp(2.0) is 7.389056`
`log( x )`	Natural logarithm of x (base e)	`log(2.718282) is 1.0`, `log(7.389056) is 2.0`
`log10( x )`	Logarithm of x (base 10)	`log10(10.0) is 1.0`, `log10(100.0) is 2.0`
`pow( x, y )`	x raised to power y (xy)	`pow(2.0, 7.0) is 128.0`, `pow(9.0, .5) is 3.0`
`sqrt( x )`	square root of x	`sqrt(900.0) is 30.0`, `sqrt(9.0) is 3.0`
`fabs( x )`	Absolute value of x—always returns a float. Python also has the built-in function `abs`, which returns an `int` or a `float`, based on its argument.	`fabs(5.1) is 5.1`, `fabs(-5.1) is 5.1`
`fmod( x, y )`	Remainder of x/y as a floating-point number	`fmod(9.8, 4.0) is 1.8`

Exercise (crate diagonal)

A shipping crate is `4` by `7` by `4`. Write `space_diagonal(w, h, d)` that uses `math.sqrt` and returns the space diagonal (the long line through the box). Print `space_diagonal(4, 7, 4)`.

In [ ]:
```
import math

def space_diagonal(w, h, d):
    # long line through the box
    return None

print(space_diagonal(4, 7, 4))
```

Solution (crate diagonal). `sqrt(w*w + h*h + d*d)`.

In [1]:
```
import math

def space_diagonal(w, h, d):
    return math.sqrt(w * w + h * h + d * d)
```

```
print(space_diagonal(4, 7, 4))
```

```
9.0
```

4.14 `import`: A Deeper Look

Importing Multiple Identifiers from a Module

- Use `from…import` to import a comma-separated list of identifiers from a module then use them without having to precede them with the module name and a dot (`.`):

In [5]:
```
import math
math.fabs(-10)
```

Out[5]:
```
10.0
```

The next cell runs `from math import fabs,ceil, floor`.

In [6]:
```
from math import fabs,ceil, floor
```

The next cell runs `fabs(-10)`.

In [7]:
```
fabs(-10)
```

Out[7]:
```
10.0
```

The next cell runs `ceil(10.3)`.

In [8]:
```
ceil(10.3)
```

Out[8]:
```
11
```

The next cell runs `floor(10.7)`.

In [9]:
```
floor(10.7)
```

Out[9]:
```
10
```

Caution: Avoid Wildcard Imports

- Import *all* identifiers defined in a module with a **wildcard** `import`.
- Makes all of the module's identifiers available.
- Can lead to subtle errors.
- Considered a dangerous practice.

```
In [10]: e = 'hello'
```

The next cell runs `from math import *`.

```
In [11]: from math import *
```

The next cell runs `e`.

```
In [12]: e
```

```
Out[12]: 2.718281828459045
```

- After executing the import, variable `e` is replaced, possibly by accident, with the `math` module's constant `e`.

Binding Names for Modules and Module Identifiers

- Sometimes it's helpful to import a module and use an abbreviation for it to simplify your code.
- The `import` statement's **`as`** clause allows you to specify the name used to reference the module's identifiers.

```
In [13]: import statistics as stats
```

The next cell runs `grades = [85, 93, 45, 87, 93]`.

```
In [14]: grades = [85, 93, 45, 87, 93]
```

The next cell runs `stats.mean(grades)`.

```
In [15]: stats.mean(grades)
```

```
Out[15]: 80.6
```

Exercise (from-import pizza)

Import `pi` with `from math import pi`. Write `slice_area(diameter, n)` that returns the area of one pizza slice when the pizza is cut into `n` equal slices. Print `slice_area(12, 8)`.

In [ ]:
```
from math import pi

def slice_area(diameter, n):
    # one slice of a pizza with this diameter
    return None

print(slice_area(12, 8))
```

Solution

In [1]:
```
from math import pi

def slice_area(diameter, n):
    radius = diameter / 2
    return pi * radius * radius / n

print(slice_area(12, 8))
```

```
14.137166941154069
```

4.4 Random-Number Generation

- Can introduce the element of chance via the Python Standard Library's `random` module.

Rolling a Six-Sided Die

- Product 10 random integers in the range 1–6 to simulate rolling a six-sided die:

In [17]:
```
import random
```

The next cell runs `for roll in range(10):`.

In [18]:
```
for roll in range(10):
    print(random.randrange(1, 7), end=' ')
```

```
4 1 5 3 5 3 2 1 5 3
```

- `randrange` function generates an integer from the first argument value up to, but not including, the second argument value.
- Different values are displayed if you re-execute the loop.

In [19]:
```
for roll in range(10):
    print(random.randrange(1, 7), end=' ')
```

```
3 1 1 6 4 4 1 4 6 2
```

- Can guarantee **reproducibility** of a random sequence with the `random` module's `seed` function.

Rolling a Six-Sided Die 6,000,000 Times

- If `randrange` truly produces integers at random, every number in its range has an equal probability (or chance or likelihood) of being returned each time we call it.
- Roll a die 6,000,000 times.
- Each die face should occur approximately 1,000,000 times.
- We used Python's underscore (_) digit separator to make the value 6000000 more readable.

```
### Rolling a Six-Sided Die 6,000,000 Times

# fig04_01.py
"""Roll a six-sided die 6,000,000 times."""
import random

# face frequency counters
frequency1 = 0
frequency2 = 0
frequency3 = 0
frequency4 = 0
frequency5 = 0
frequency6 = 0

# 6,000,000 die rolls
for roll in range(6_000_000):  # note underscore separators
    face = random.randrange(1, 7)

    # increment appropriate face counter
    if face == 1:
        frequency1 += 1
    elif face == 2:
        frequency2 += 1
    elif face == 3:
        frequency3 += 1
    elif face == 4:
        frequency4 += 1
    elif face == 5:
        frequency5 += 1
    elif face == 6:
        frequency6 += 1

print(f'Face{"Frequency":>13}')
print(f'{1:>4}{frequency1:>13}')
print(f'{2:>4}{frequency2:>13}')
print(f'{3:>4}{frequency3:>13}')
print(f'{4:>4}{frequency4:>13}')
```

```
print(f'{5:>4}{frequency5:>13}')
print(f'{6:>4}{frequency6:>13}')
```

In [20]:
```
run fig04_01.py
```

```
Face    Frequency
   1      1000704
   2      1000701
   3      1000067
   4       997833
   5      1000032
   6      1000663
```

Seeding the Random-Number Generator for Reproducibility

- Function `randrange` generates pseudorandom numbers.
- Numbers appear to be random, because each time you start a new interactive session or execute a script that uses the random module's functions, Python internally uses a different seed value.
- When you're debugging logic errors in programs that use randomly generated data, it can be helpful to use the same sequence of random numbers.
- To do this, use the random module's `seed` function to seed the random-number generator:

In [21]:
```
random.seed(37)
```

The next cell runs `for roll in range(10):`.

In [22]:
```
for roll in range(10):
    print(random.randrange(1, 7), end=' ')
```

```
6 5 1 5 6 6 6 5 1 6
```

The next cell runs `for roll in range(10):`.

In [23]:
```
for roll in range(10):
    print(random.randrange(1, 7), end=' ')
```

```
3 4 5 1 6 4 5 3 4 6
```

The next cell runs `random.seed(42)`.

In [24]:
```
random.seed(42)
```

The next cell runs `for roll in range(10):`.

In [25]:
```
for roll in range(10):
    print(random.randrange(1, 7), end=' ')
```

```
6 1 1 6 3 2 2 2 6 1
```

The 6,000,000-roll program from the book, inlined so Colab can run it:

In [26]:
```
# fig04_01.py
"""Roll a six-sided die 6,000,000 times."""
import random

# face frequency counters
frequency1 = 0
frequency2 = 0
frequency3 = 0
frequency4 = 0
frequency5 = 0
frequency6 = 0

# 6,000,000 die rolls
for roll in range(6_000_000):  # note underscore separators
    face = random.randrange(1, 7)

    # increment appropriate face counter
    if face == 1:
        frequency1 += 1
    elif face == 2:
        frequency2 += 1
    elif face == 3:
        frequency3 += 1
    elif face == 4:
        frequency4 += 1
    elif face == 5:
        frequency5 += 1
    elif face == 6:
        frequency6 += 1

print(f'Face{"Frequency":>13}')
print(f'{1:>4}{frequency1:>13}')
print(f'{2:>4}{frequency2:>13}')
print(f'{3:>4}{frequency3:>13}')
print(f'{4:>4}{frequency4:>13}')
print(f'{5:>4}{frequency5:>13}')
print(f'{6:>4}{frequency6:>13}')
```

```
Face    Frequency
   1      1001857
   2       998954
   3      1000758
   4       998959
   5       999839
   6       999633
```

Exercise (d6)

Write `rolls_until_six()` that calls `random.randrange(1, 7)` until the result is 6 and **returns how many rolls that took**, including the 6. Print one run.

In [ ]:
```
import random

def rolls_until_six():
```

```
    # keep rolling until 6; return the count
    return None

print(rolls_until_six())
```

Solution (d6). A counter and a loop; stop when the roll is 6.

In [1]:
```
import random

def rolls_until_six():
    n = 0
    while True:
        n += 1
        if random.randrange(1, 7) == 6:
            return n

print(rolls_until_six())
```

```
6
```

Comprehensive exercises

Exercise (comprehensive)

Two random grid points: write `random_distance()` that picks integers `x1, y1, x2, y2` each from 0 through 9 with `random.randrange`, then returns the Euclidean distance using `math.sqrt`. Print one distance.

In [ ]:
```
import math
import random

def random_distance():
    # four random coordinates, then distance
    return None

print(random_distance())
```

Solution

In [1]:
```
import math
import random

def random_distance():
    x1 = random.randrange(0, 10)
    y1 = random.randrange(0, 10)
    x2 = random.randrange(0, 10)
    y2 = random.randrange(0, 10)
    return math.sqrt((x2 - x1) ** 2 + (y2 - y1) ** 2)

print(random_distance())
```

```
5.0
```

Takeaways

- `import math` and `import random` load the standard library. Prefer `import math` over `from math import *`.
- `random.randrange(1, 7)` rolls a six-sided die. Seed the generator when you need the same rolls again.
- Next class: **More functions, tuples, and dictionaries**.