Python语言程序设计

Python Programming

Session 09



Tom Lotz (tom.lotz@outlook.com)

Content

	Review
01	Classes and objects
02	Attributes and __init__
03	Methods and self
04	Independent practice

Review

The Café Menu: One Dictionary



A dictionary stores key–value pairs.
Here the key is an item name and the value is its price.

Python code

```
prices = {"tea": 6,
          "fruit cup": 8,
          "sandwich": 15}

print(prices["fruit cup"])
```

Expected output

```
8
```

Dictionary Syntax



Part	Example	What it does
Braces	{ ... }	Hold the dictionary entries
Colon	"tea": 6	Separates a key from its value
Comma	"tea": 6, "cake": 9	Separates two entries
Square brackets	prices["tea"]	Look up a value by key

Adding an Entry



Assign a value to a new key to add an entry.

Python code

```
prices = {"tea": 6, "cake": 9}
prices["juice"] = 10
print(prices["juice"])
```

Expected output

```
10
```



Updating an Entry

Assigning to an existing key replaces its old value.
Each key has one value at a time.

Python code

```
prices = {"tea": 6, "cake": 9}
prices["tea"] = 7
print(prices["tea"])
print(prices["cake"])
```

Expected output

```
7
9
```

Missing Keys



"juice" is not a key, so Python raises a KeyError.
The program stops unless the exception is handled.
Next: check whether the key exists before looking it up.

Python code

```
prices = {"tea": 6, "cake": 9}
print(prices["juice"])
```

Result: KeyError exception

```
KeyError: 'juice'
```

Checking a Key with in



Use in to check the keys. Put the lookup inside the if branch. Try tea, then try juice.

Python code

```
prices = {"tea": 6, "cake": 9}
item = input("Item: ")
if item in prices:
    print(prices[item])
else:
    print("Item not found")
```

Displaying Keys and Values



Use the loop variable twice: once as the item name and once as the key for the price lookup.

Python code

```
prices = {"tea": 6,
          "fruit cup": 8,
          "sandwich": 15}

for item in prices:
    print(item, prices[item], "yuan")
```

One Dictionary for One Product



A product can have several fields. Here, name is a string, and price and stock are integers.

Python code

```
product = {"name": "tea",
           "price": 6, "stock": 12}
print(product["name"])
print(product["stock"])
```

Expected output

```
tea
12
```

A List of Product Dictionaries



Use an index to choose the product.
Then use a key to choose a field from that product.

Python code

```
products = [
    {"name": "tea", "price": 6},
    {"name": "cake", "price": 9}
]

print(products[0]["name"])
print(products[1]["price"])
```

Expected output

```
tea
9
```

Updating a Dictionary Inside a List



products[0] selects the first dictionary. ["stock"] selects its stock. The assignment changes 12 to 3. Other values stay the same. Expected output: 3.

Python code

```
products = [
    {"name": "tea", "price": 6, "stock": 12},
    {"name": "cake", "price": 9, "stock": 4}
]

products[0]["stock"] = 3
print(products[0]["stock"])
```

A Dictionary Can Hold Lists



Here the arrangement is reversed: a dictionary holds lists. Choose a key first, then use an index in that list.

Python code

```
groups = {"red": ["Mina", "Kai"],
          "blue": ["Noor", "Leo"]}

print(groups["red"][0])
```

Expected output

```
Mina
```

Classes and objects

A Product Dictionary

Last session, a dictionary held the name and stock.
The string "stock" is a key.

Python code

```
tea = {"name": "tea", "stock": 12}
print(tea["stock"])
```

Expected output

```
12
```

A Product Object

Product is a class. Product("tea", 12) creates an object.
Run this example first. We will unpack __init__ next.

Python code

```
class Product:
    def __init__(self, name, stock):
        self.name = name
        self.stock = stock

tea = Product("tea", 12)
print(tea.stock)
```

Expected output

```
12
```

One Class, Two Objects

Each call to Product(...) creates a new object.
Use the variable name to choose which object to read.

Python code

```
class Product:
    def __init__(self, name, stock):
        self.name = name
        self.stock = stock

tea = Product("tea", 12)
cake = Product("cake", 4)
print(tea.stock)
print(cake.stock)
```

Expected output

```
12
4
```

Keys and Attributes

Dictionary lookup: tea["stock"]

Object attribute: tea.stock

Both expressions retrieve the stock for tea.
An attribute is a named value on an object.

Practice: Reading Two Objects

CORE EXERCISE

Use the Product class from the example.

Create rice with stock 9 and milk with stock 2.

Print each name and stock using dot notation.

Expected: rice 9, then milk 2.

OPTIONAL CHALLENGES

Finish the core first.
Choose any challenge.

A. Add a third product and print its name and stock.

B. Predict the effect of changing only the first call’s stock argument.

Reading Two Objects: Solution

Keep the Product class above these lines.
Each line reads attributes from one object.

Python code

```
rice = Product("rice", 9)
milk = Product("milk", 2)

print(rice.name, rice.stock)
print(milk.name, milk.stock)
```

Expected output

```
rice 9
milk 2
```

Attributes and __init__

How __init__ Sets Starting Values

Python calls __init__ when it creates a Product.
The arguments "tea" and 12 become name and stock.

Python code

```
class Product:
    def __init__(self, name, stock):
        self.name = name
        self.stock = stock

tea = Product("tea", 12)
print(tea.name, tea.stock)
```

Expected output

```
tea 12
```

Parameter and Attribute

In self.stock = stock:

The right-hand stock is the parameter. It receives 12.

The left-hand self.stock stores 12 on the new object.

After __init__ finishes, tea.stock can still read that value.

What self Refers To

self refers to the object being initialized or used.

When you create tea, self refers to that new object.
When you create cake, self refers to the other new object.

You write self in the definition. Python supplies the object when you call Product(...) or call a method through a dot.

Practice: A Book Class

CORE EXERCISE

Write a Book class with __init__. Store title and pages as attributes.

Create a book named Maps with 80 pages and one named Rivers with 120 pages.

Print each title and page count.

Expected: Maps 80, Rivers 120.

OPTIONAL CHALLENGES

Finish the core first.
Choose any challenge.

A. Create a third book with the same title but a different page count.

B. Add an author attribute and update all the creation calls.

Book Class: Solution

The two calls use the same definition with different arguments.

Python code

```
class Book:
    def __init__(self, title, pages):
        self.title = title
        self.pages = pages

first = Book("Maps", 80)
second = Book("Rivers", 120)
print(first.title, first.pages)
print(second.title, second.pages)
```

Expected output

```
Maps 80
Rivers 120
```

Debugging: A Missing Attribute

CORE EXERCISE

Why does tea.stock fail?
Fix the class and run it in IDLE.

Python code

```
class Product:
    def __init__(self, n):
        stock = n

tea = Product(12)
print(tea.stock)
```

OPTIONAL CHALLENGES

Finish the core first.
Choose any challenge.

Add a second object with stock 4 and verify that both objects can report their own stock.

Missing Attribute: Solution

The original code assigns a local variable.
Use self.stock to store the value on the object.

Python code

```
class Product:
    def __init__(self, n):
        self.stock = n

tea = Product(12)
print(tea.stock)
```

Expected output

```
12
```

Methods and self

A Method Changes an Object

A method is a function defined in a class.
Keep restock indented at the same level as __init__.

Python code

```
class Product:
    def __init__(self, name, stock):
        self.name = name
        self.stock = stock

    def restock(self, amount):
        self.stock = self.stock + amount

tea = Product("tea", 12)
tea.restock(3)
print(tea.stock)
```

A Method Call and Its Arguments

Call: tea.restock(3)

Python supplies tea as self. The argument 3 goes to amount.

self.stock = self.stock + amount reads 12, adds 3,
and stores 15 on tea.

The method updates stock. The print call displays the result.

Practice: Predict the Stock Counts

CORE EXERCISE

Use Product with restock.
Predict both lines before running.

Python code

```
tea = Product("tea", 12)
cake = Product("cake", 4)
tea.restock(3)
print(tea.stock)
print(cake.stock)
```

OPTIONAL CHALLENGES

Finish the core first.
Choose any challenge.

Call cake.restock(2) before the print calls. Predict the output again. Explain which object self refers to in each call.

Stock Counts: Solution

The first call changes tea to 15. Cake still has 4.
With the extra call, cake would have 6.

Python code

```
tea = Product("tea", 12)
cake = Product("cake", 4)
tea.restock(3)
print(tea.stock)
print(cake.stock)
```

Expected output

```
15
4
```

Practice: Restock One Product

CORE EXERCISE

Use Product with restock.

Create juice with stock 5 and bread with stock 8.

Add 4 to bread. Print each name and stock to verify that only bread changed.

Expected: juice 5, bread 12.

OPTIONAL CHALLENGES

Finish the core first.
Choose any challenge.

A. Restock bread again and test whether the two changes add up.

B. Make restock ignore zero or negative amounts. Test -2.

Restock One Product: Solution

Keep the Product class with restock above this code.
Call the method on bread, then check both objects.

Python code

```
juice = Product("juice", 5)
bread = Product("bread", 8)
bread.restock(4)
print(juice.name, juice.stock)
print(bread.name, bread.stock)
```

Expected output

```
juice 5
bread 12
```

Independent practice

Independent Practice: Two Counters

CORE EXERCISE

Write a Counter class.
Use __init__ to store count.
Write add(amount) to increase it.

Create two counters: 0 and 5.
Add 3 to the first, then 2 to the second. Print both counts after each call.

After call 1: 3 and 5.
After call 2: 3 and 7.

OPTIONAL CHALLENGES

Finish the core first.
Choose any challenge.

A. Add a reset() method. Reset only the second counter.

B. Write get_count() to return the count. Print the returned value outside the class.

Counter Solution: The Class

Part 1 of 2. The class stores a count and defines how to change it.

Python code

```
class Counter:
    def __init__(self, count):
        self.count = count

    def add(self, amount):
        self.count = self.count + amount
```

Counter Solution: Testing Two Objects

Part 2 of 2. Put these lines below the class.
Check both counts after each method call.

Python code

```
morning = Counter(0)
afternoon = Counter(5)
morning.add(3)
print(morning.count, afternoon.count)
afternoon.add(2)
print(morning.count, afternoon.count)
```

Expected output

```
3 5
3 7
```

Independent Check

1. Which is the class in tea = Product("tea", 12)?
2. In self.stock = stock, which name is the parameter?
3. In cake.restock(2), which object is self?
4. If tea has 12 and cake has 4, what are both stocks after calling cake.restock(2)?

Recap

A class defines what its objects store and can do.

__init__ sets the starting attributes of a new object.

self refers to the object being used.

A method can change that object's attributes.

Test two objects to check that each keeps its own values.

Homework: Player Scores

CORE EXERCISE

Write a Player class with name and score attributes.

Write add_points(amount) to increase one player's score.

Create Mina with 5 and Kai with 2. Add 4 to Kai. Print both names and scores. Submit your .py file.

Expected: Mina 5, Kai 6.

OPTIONAL CHALLENGES

Finish the core first.
Choose any challenge.

A. Add reset_score() and test it on only one player.

B. Add a method that returns whether the player has at least 10 points.