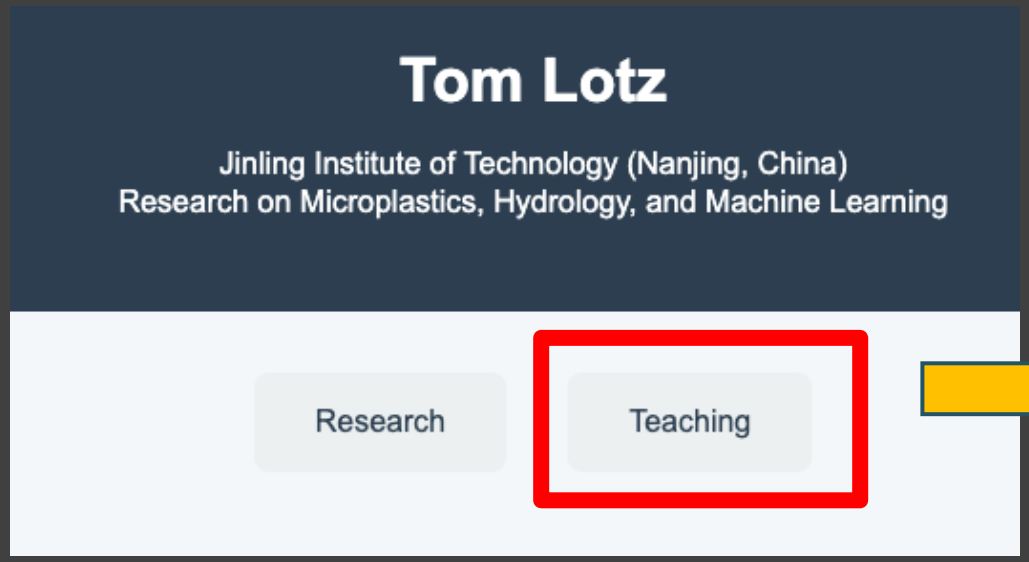
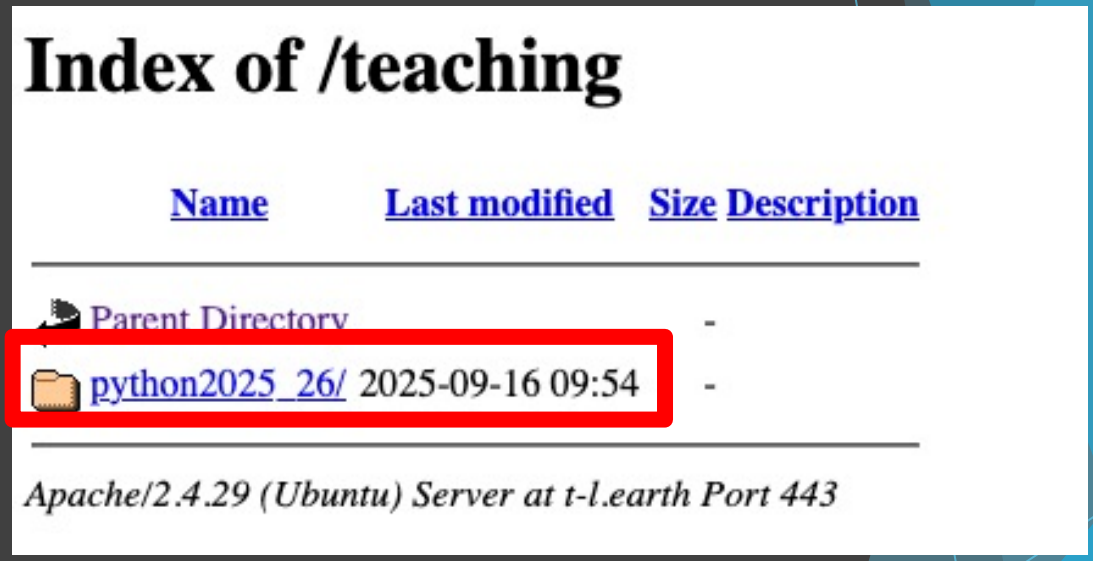
Download the cheat sheets and slides from here

t-l.earth





Python语言程序设计

Python Programming

2025/26

Session 08

Tom Lotz (tom.lotz@outlook.com)

Content

	Brain Activation + Review
01	Inheritance
02	Composition and Object Interaction
03	Exercises

Brain Activation

Basic Function

- Write a function that greets a user by name.

```
def greet(name):
    print(f"Hello, {name}!")


greet("Ali")
```

Working with Lists

- Create a list of three favorite fruits and print each one in a loop.

```
fruits = ["apple", "banana", "cherry"]
for fruit in fruits:
    print(fruit)
```

Using Dictionaries

- Create a small dictionary representing a student and print a message using its values.

```
student = {"name": "Lina", "major": "Software Engineering"}
print(f"{student['name']} studies {student['major']}")
```

Conditionals & Input

- Ask the user for a number and print whether it’s even or odd.

```
number = int(input("Enter a number: "))
if number % 2 == 0:
    print("Even number!")
else:
    print("Odd number!")
```

Simple Function Challenge

- Write a function that takes two numbers and returns the larger one.

Review

Python Indentation

Python Indentation

- Script flow logic in Python is determined by indentation
- Indent can be done by spaces or tabs (must always be the same number)

Python Indentation


```
print("Hello world!")

print("Hello world again!")
```

Python Indentation


```
print("Hello world!")

if 2 > 1:
    print("Hello world again!")
```

Python Indentation


```
if True:
    if 2 > 1:
        print("Hello world again!")
    print("Goodbye world!")
```

Python Indentation


```
class Student:
    def __init__(self,name,age):
        self.name=name
        self.age=age

    def print_info(self):
        print("Name:",self.name, "Age:",self.age)
```

Python Indentation


```
class Student:
    def __init__(self,name,age):
        self.name=name
        self.age=age

    def print_info(self):
        print("Name:",self.name, "Age:",self.age)


my_student = Student("John",20)
```

Python Indentation


```
class Student:
    def __init__(self,name,age):
        self.name=name
        self.age=age

def print_info(self):
    print("Name:",self.name, "Age:",self.age)


my_student = Student("John",20)
```

Python Indentation


```
class Student:
    def __init__(self,name,age):
        self.name=name
        self.age=age

    def print_info(self):
        print("Name:",self.name, "Age:",self.age)


    my_student = Student("John",20)
```

From Code to Models

- Programming is not just writing instructions; it's building models of reality. We use variables and lists for data, but when data and behavior belong together, we need something more structured.
- Example:
  - A student has a name, major, and GPA.
  - A bank account has an owner, balance, and actions (deposit, withdraw).
- This combination of data + actions is what a class represents.

The Core Idea of Object-Oriented Programming (OOP)

- In OOP, we describe things — not just actions. Each object in code mirrors an object in the real world.
- Concepts:
  - Class: the blueprint (defines what all objects of that kind know and do)
  - Object (instance): one specific example of that class
  - Attributes: data values inside each object
  - Methods: actions that object can perform

Thinking Like a Designer

- When designing a class:
  - Identify the entity (noun) you want to represent.
  - Decide which details are important (attributes).
  - Define the actions (methods) that belong to it.

- Example: Modeling a book. What do we need?

Defining a Class

- class keyword starts a class definition.
- __init__() initializes new objects.
- self refers to this particular instance.

```
class Dog:
    def __init__(self, name, age):
        self.name = name
        self.age = age
```

Creating an Object

- We can create instances of the Dog class by calling it.

```
my_dog = Dog('Willie', 6)
```

- Python calls __init__() automatically.
- my_dog is now an object with data inside.

Adding Behavior (Methods)

- We can add methods to a class.

```python
class Dog:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def sit(self):
        print(f"{self.name} is now sitting.")

    def roll_over(self):
        print(f"{self.name} rolled over!")
```

- Each method must include self.

Using Methods

- The methods of the instance are called in this way:

```
my_dog = Dog('Willie', 6)
my_dog.sit() # Willie is now sitting.
my_dog.roll_over() # Willie rolled over!
```

Multiple Instances

- Once a class has been defined, we can create as many instances as we want.

```
my_dog = Dog('Willie', 6)
your_dog = Dog('Lucy', 3)

my_dog.sit() # Willie is now sitting.
your_dog.roll_over() # Lucy rolled over!
```

Adding Default Attributes

- Every student starts with 0 credits

```
class Student:
    def __init__(self, name, major):
        self.name = name
        self.major = major
        self.credits = 0 # default value
```

Modifying Attributes Directly

- We can change values manually:

```
student1 = Student('James', 2)
student1.credits = 20
student1.show_info() # James studies 2 and has 20 credits.
```

- This works, but there is no control over the data added for the attribute (negative values, strings, ...).

Updating Attributes Safely

- Better to add a method that updates credits but protects against invalid data.

```
def update_credits(self, new_value):
    if new_value >= 0:
        self.credits = new_value
    else:
        print("Credits cannot be negative!")
```

```
student1.update_credits(-20)
```

Inheritance

Why Inheritance?

- Sometimes we have several classes that share the same structure or behavior. Instead of rewriting code, we can reuse it.
- Example:
  - Vehicle → common properties: make, model, year.
  - ElectricVehicle → adds battery or charging behavior.
- Inheritance saves time and avoids duplication.

The Parent Class

- Start with a general class.

```
class Vehicle:
    def __init__(self, make, model, year):
        self.make = make
        self.model = model
        self.year = year

    def describe(self):
        print(f"{self.year} {self.make} {self.model}")
```

The Child Class

- A subclass inherits from the parent.

```
# Child class
class ElectricVehicle(Vehicle):
    def __init__(self, make, model, year, battery_capacity):
        super().__init__(make, model, year)
        self.battery_capacity = battery_capacity
```

- super() calls the parent's __init__() to reuse setup code. Add only what's specific to this subclass.

The Child Class

- A subclass inherits from the parent.

```
# Child class
class ElectricVehicle(Vehicle):
    def __init__(self, make, model, year, battery_capacity):
        super().__init__(make, model, year)
        self.battery_capacity = battery_capacity


my_ev = ElectricVehicle("whatever", "Stellaria", "1999", 20000)
my_ev.describe()
```

Mini Task 1

- Extend Vehicle to create GasolineVehicle with one extra attribute tank_size (you can copy the Vehicle class from CheatSheet_08.py).
    - Use super() to initialize shared attributes.

Using Inherited Methods

- Child classes automatically get all methods from the parent.

```
# Child class
class ElectricVehicle(Vehicle):
    def __init__(self, make, model, year, battery_capacity):
        super().__init__(make, model, year)
        self.battery_capacity = battery_capacity


my_ev = ElectricVehicle("whatever", "Stellaria", "1999", 20000)
my_ev.describe()
```

Overriding Methods

- A child class can redefine a parent's method.

```
# Child class with method overwrite
class ElectricVehicle(Vehicle):
    def __init__(self, make, model, year, battery_capacity):
        super().__init__(make, model, year)
        self.battery_capacity = battery_capacity

    def describe(self):
        print(f"{self.year} {self.make} {self.model} with {self.battery_capacity} kWh battery")
```

Mini Task 2

- In GasolineVehicle, override describe() so it also shows tank size.
- Create one Vehicle, one ElectricVehicle (copy ElectricVehicle from the CheatSheet), and one GasolineVehicle.
- Call describe() on each.

Mini Task 3

- Add a method charge() to ElectricVehicle and refuel() to GasolineVehicle.
- Call both on respective objects.

Common Mistakes to Avoid

- Forgetting super().__init__() → attributes from parent won't exist.
- Misaligned indentation → methods not actually inside the class.
- Reusing parent names incorrectly → shadowing variables.
- Always check that your subclass runs the parent's setup first.

Wrap-up

- Inheritance connects related classes.
- super() lets you reuse parent initialization.
- Overriding allows customization.

Composition and Object Interaction

Why Composition?

- Not everything fits into inheritance. Sometimes one class should use another, not become another.
- Example: A Car has a Battery — not the same thing.
- Composition = building larger structures from smaller classes.

The Contained Class

- We define the contained class first.

```
class Battery:
    def __init__(self, capacity):
        self.capacity = capacity

    def describe_battery(self):
        print(f"Battery capacity: {self.capacity} kWh")
```

The Container Class

- Include a class as an attribute inside another.

```
class ElectricVehicle:
    def __init__(self, make, model, year, capacity):
        self.make = make
        self.model = model
        self.year = year
        self.battery = Battery(capacity)  #here is the contained class

    def describe(self):
        print(f"{self.year} {self.make} {self.model}")
        self.battery.describe_battery()
```

The Container Class

- Include a class as an attribute inside another.

```
class ElectricVehicle:
    def __init__(self, make, model, year, capacity):
        self.make = make
        self.model = model
        self.year = year
        self.battery = Battery(capacity)  #here is the contained class

    def describe(self):
        print(f"{self.year} {self.make} {self.model}")
        self.battery.describe_battery()
```

The Container Class

- Include a class as an attribute inside another.

```
class ElectricVehicle:
    def __init__(self, make, model, year, capacity):
        self.make = make
        self.model = model
        self.year = year
        self.battery = Battery(capacity)  #here is the contained class

    def describe(self):
        print(f"{self.year} {self.make} {self.model}")
        self.battery.describe_battery()
```

Mini Task 4

- Use your existing ElectricVehicle class and add a Battery class to it.
  - Each EV should automatically create a Battery when initialized.
  - Add a method show_range() inside Battery that prints the driving range.

Object Interaction

- Objects can communicate by calling each other's methods. Example:

```
ev = ElectricVehicle('Tesla', 'Model 3', 2024, 75)
ev.battery.describe_battery()
```

Extending Composition

- Composition allows complex relationships:
  - A Zoo has Animals.
  - A Library has Books.
  - A School has Students and Teachers. Each object keeps its own logic but can interact with others.

Mini Task 5

- Create a new class Fleet that stores multiple vehicles.
- Use show_all() of your Fleet object.
- Help:

```
class Fleet:
    def __init__(self):
        self.vehicles = []

    def add_vehicle(self, vehicle):
        self.vehicles.append(vehicle)

    def show_all(self):
        for v in self.vehicles:
            v.describe()
```

Common Mistakes

- Forgetting to initialize the contained class (e.g., self.battery = Battery(capacity)).
- Trying to access attributes that belong to another object directly.
- Forgetting to prefix with the correct object (self.battery.range, not just range).
- Always use dot notation to move through layers.

Wrap-up

- Composition = “has-a” relationship.
- Enables objects to collaborate.
- Builds modular, realistic programs.

Exercises

Zoo

- We are building a zoo that grows with each task.
- You can find the task list in the CheatSheet_08.py file on the server.
- Try to finish as many tasks as possible!