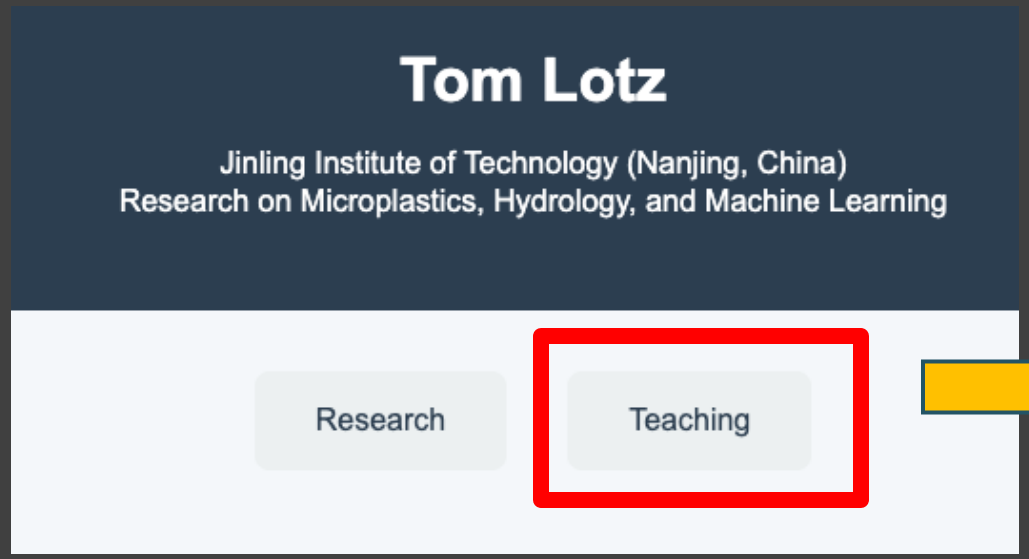
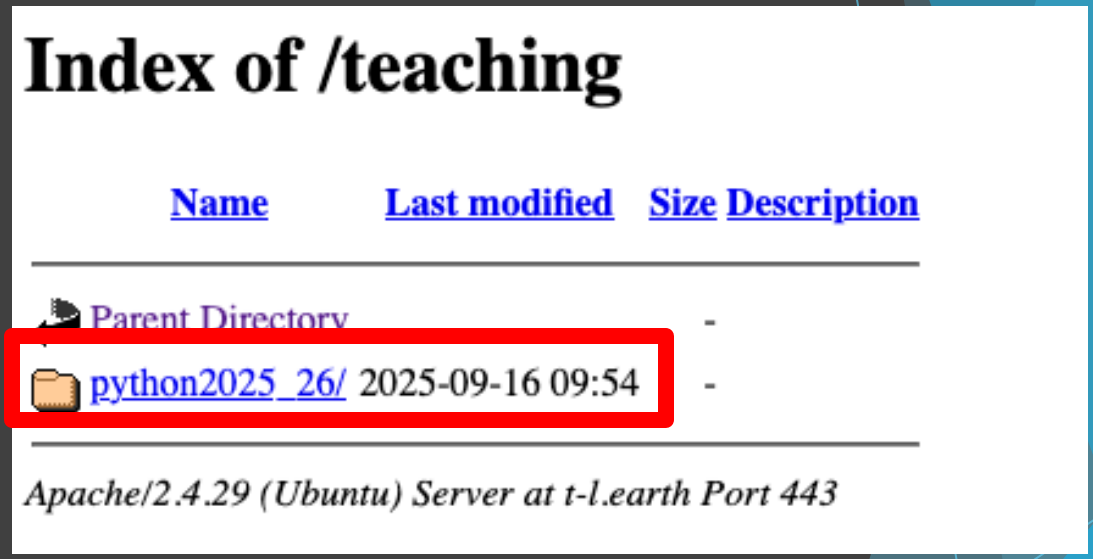
Download the cheat sheets and slides from here

t-l.earth





Python语言程序设计

Python Programming

2025/26

Session 09

Tom Lotz (tom.lotz@outlook.com)

Content

	Brain Activation + Review
01	Imports and Modules
02	Reading and Writing Files
03	Exceptions and Safe Programs
04	Exercises

Brain Activation

Function and Loop

```python
def square(n):
    return n ** 2

for number in range(1, 6):
    result = square(number)
    print(f"{number} squared is {result}")
```

Simple Dictionary

```
fruits = {
    "apple": "red",
    "banana": "yellow",
    "grape": "purple"
}

print(f"The color of a banana is {fruits['banana']}.")
```

Review

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

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.

```
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!
```

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.

```python
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
```

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

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")
```

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 insi

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

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


```python
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()
```

Imports and Modules

Why Modules?

- Large programs become messy if everything is in one file.
- Modules allow you to:
  - Separate logic cleanly
  - Reuse classes/functions across files
  - Avoid repetition

Why Modules?

- Example idea:
  - car.py → contains Car class
  - my_car.py → imports and uses Car
- Modules = clean, organized, reusable code.

A Module in Python

- A module is simply a .py file that Python can import.

```
# car.py
class Car:
    def __init__(self, make):
        self.make = make
```

Importing a Single Class

- A module is simply a .py file that Python can import.

```
from car import Car


my_car = Car("Audi")
print(my_car.make)
```

Importing a Single Class

- A module is simply a .py file that Python can import.

```python
from car import Car

my_car = Car("Audi")
print(my_car.make)
```

car.py

```python
class Car:
    def __init__(self, make):
        self.make = make
```

Mini Task 1

- Create two files:
  - tools.py
  - main.py
- In tools.py: write a function hello() that prints a greeting. In main.py: import and call hello().

Importing Multiple Classes

- A single module may store several related classes.

```
class Car:
    def __init__(self, make):
        self.make = make

class ElectricCar:
    def __init__(self, make, battery):
        self.make = make
        self.battery = battery
```

Importing Multiple Classes

- A single module may store several related classes.

```python
class Car:
    def __init__(self, make):
        self.make = make

class ElectricCar:
    def __init__(self, make, battery):
        self.make = make
        self.battery = battery
```

```python
from car import Car, ElectricCar

my_car = Car("Audi")
my_electric_car = ElectricCar("BYD", 10000)
```

Importing the Entire Module

- Use this when you want clarity or to avoid name conflicts.

```
import car


my_car = car.Car("Ford")
```

Importing the Entire Module

- Use this when you want clarity or to avoid name conflicts.

```
import car

my_car = car.Car("Ford")
```

Mini Task 2

- Create a real module and use it:
  - Make a file math_tools.py with two functions: add(a, b) and multiply(a, b).
  - In main.py, import the module.
- Let the user enter two numbers and print the sum and product.

Modules Importing Modules

- Modules can import each other (inheritance in this example).

```
# electric_car.py
from car import Car


class ElectricCar(Car):
    def __init__(self, make, battery):
        super().__init__(make)
        self.battery = battery
```

Using Aliases

- Shorten long names:

```
from electric_car import ElectricCar as EC
leaf = EC("Nissan")
```

- Or alias entire modules:

```
import electric_car as ec
car = ec.ElectricCar("Tesla")
```

Wrap-up

- Modules structure programs into clean files.
- Different import styles offer flexibility.
- Aliases improve readability.
- Modules can import each other.

Reading and Writing Files

Why Work With Files?

- Programs become more useful when they can:
  - Load data from disk
  - Save user progress or results
  - Process text, logs, CSV files
- Examples: reading weather data, saving notes, storing game scores.

Reading a File (Basic)

- Use Path.read_text() to load a file.

```
from pathlib import Path

path = Path('pi_digits.txt')
contents = path.read_text()
print(contents)
```

- Reads the entire file as one string.

Stripping Whitespace, splitting lines

- Use Path.read_text() to load a file, then remove whitespaces and split into lines.

```
path = Path('pi_digits.txt')
contents = path.read_text()
contents = contents.rstrip()
contents = contents.splitlines()
print(contents)
```

Mini Task 3

- Download the file notes.txt with three lines of text. Write a script to:
  - Read the file
  - Strip trailing whitespace
  - Print each line separately after using .splitlines() (hint: you need a loop)

Writing to a File

- Use write_text() to create or overwrite a file.

```
from pathlib import Path

path = Path('output.txt')
path.write_text("Hello from Python!\n")
```

- This creates the file if it doesn’t exist.

Appending Multiple Lines

- To write multiple lines, build a string with newlines:

```
contents = "Line 1\n"
contents += "Line 2\n"

path.write_text(contents)
```

Mini Task 4

- Write a script that:
  - Asks the user for three sentences (hint: input())
  - Saves all three into sentences.txt, each on its own line

What Is JSON?

- JSON = JavaScript Object Notation
  - Text format for structured data
  - Works well with Python dictionaries and lists
  - Easy to save to and read from files
- Useful for:
  - App settings
  - User preferences
- Small data files exchanged between programs

Storing a Simple Settings Dict

- Example: language + theme settings.

```
import json
from pathlib import Path

settings = {
"language": "en",
"theme": "dark"
}

path = Path("settings.json")
json_text = json.dumps(settings)
path.write_text(json_text)
```

Loading Settings Back

- Read the file, then convert JSON → Python Dictionary.

```
import json
from pathlib import Path



path = Path("settings.json")
json_text = path.read_text()
settings = json.loads(json_text)

print(settings["language"])
print(settings["theme"])
```

Mini Task 5

- Create a small user profile dictionary:
    - name
    - age
    - country
- Steps:
    - Store it into user_profile.json using json.dumps() and write_text().
- In a new script, read it back and print a nice sentence:
- Example: Tom from Germany is 90 years old.

Wrap-up

- Path.read_text() loads file content.
- Path.write_text() writes or replaces content.
- .rstrip() removes unneeded characters.
- .splitlines() helps process files line by line.
- JSON is a text format for structured data (dicts, lists, etc.).
- json.dumps() Python object → JSON string.
- json.loads() JSON string → Python object.

Exceptions and Safe Programs

What Is an Exception?

- An exception is an error that stops normal program execution.
- Examples:
  - Dividing by zero
  - Converting text to a number
  - Opening a missing file
- Without handling, the program crashes and prints a traceback.

Simple Exception Example

```
print(5 / 0)
```

```
ZeroDivisionError: division by zero
```

Basic try/except Structure

- Use try for risky code. Use except to catch the error.

```
try:
    print(5 / 0)
except ZeroDivisionError:
    print("You can't divide by zero!")
```

- The program continues running without crashing.

Mini Task 6

- Write a script that:
  - Asks the user for two numbers
  - Tries to divide them
  - Catches ZeroDivisionError and prints a friendly message

The else Block

- Use else when the try block succeeds.

```
try:
    result = 10 / 2
except ZeroDivisionError:
    print("Error!")
else:
    print(result)
```

- Keeps the success logic separate and clean.

Handling Bad User Input

- Converting text to a number may fail.

```
try:
    n = int(input("Enter a number: "))
except ValueError:
    print("That's not a number.")
```

- Prevents crashes from invalid input.

Mini Task 7

- Create a simple calculator that:
  - Prompts for two numbers
  - Uses try/except to catch ValueError
  - Prints the sum if inputs are valid

File Errors: Missing Files

- Trying to open a missing file raises FileNotFoundError.

```
from pathlib import Path

path = Path("unknown.txt")

try:
    text = path.read_text()
except FileNotFoundError:
    print("File not found.")
```

Using else with File Reads

- Separate successful read logic.

```
try:
    text = path.read_text()
except FileNotFoundError:
    print("File missing.")
else:
    print(text)
```

Mini Task 8

- Write a script that:
  - Asks for a filename
  - Tries to open it
  - If missing → prints a friendly error
  - If found → prints the content

Chaining Multiple Exceptions

- You can catch different errors separately.

```
try:
    number = int(input("Enter number: "))
    result = 10 / number
except ValueError:
    print("Please enter a valid integer.")
except ZeroDivisionError:
    print("Number cannot be zero.")
```

Wrap-up

- Exceptions prevent crashes and improve user experience.
- try/except handles errors.
- else keeps clean success logic.
- FileNotFoundError and ValueError are common in real programs.
- Use multiple except blocks for different problems.

Exercises

Exercise

- We are building a project that grows with each task.
- You can find the task list in the CheatSheet_09.py file on the server.
- Try to finish as many tasks as possible!