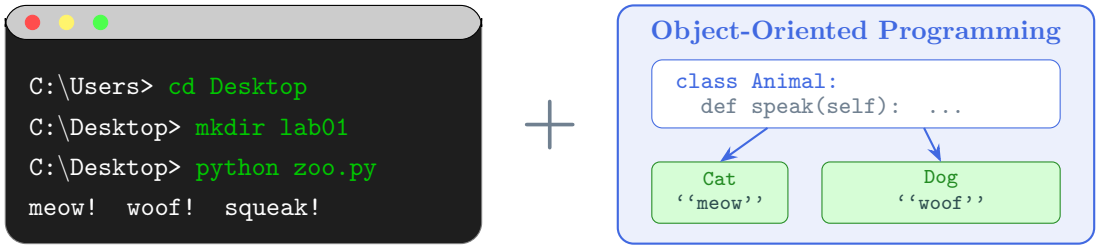
Programming 2: Lab 1

Terminal Commands & OOP Fundamentals

A Walkthrough Tutorial



Comp 111 — Programming 2

Forman Christian College

Instructor: P2 course staff

Spring 2026

Estimated time: 3 hours

How to Use This Lab

Welcome to your first lab! This is a **walkthrough tutorial** — meaning it is designed to **guide** you through learning, not to test you. Work through each section in order. Read the explanations, type every command and code example yourself (don't copy-paste!), and attempt every exercise.

Exercises are marked by difficulty:

★ **Easy** — You should be able to do these right away. They reinforce what you just read.

★★ **Medium** — These require a bit of thinking. Combine concepts you've learned.

★★★ **Hard** — Challenging! These push you further. Don't worry if you need help.

Before You Begin

Make sure you have:

- A Windows 10 or 11 computer
- Python installed (type `python --version` in your terminal to check)
- A willingness to make mistakes — that's how you learn!

1 Opening the Terminal

The **terminal** (also called the **command prompt** or **console**) is a text-based way to interact with your computer. Instead of clicking on folders and icons, you type commands. Every professional developer uses the terminal daily — and you will too, especially when we start using **Git** in upcoming lectures.

1.1 How to Open It

There are several ways to open the Command Prompt on Windows:

1. Press `Win + R`, type `cmd`, and press Enter.
2. Search for "Command Prompt" in the Start menu.
3. Right-click the Start button and select "Terminal" or "Command Prompt."

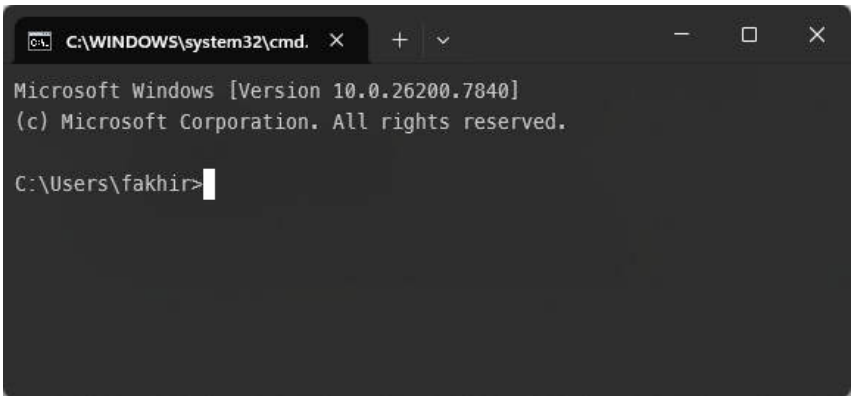


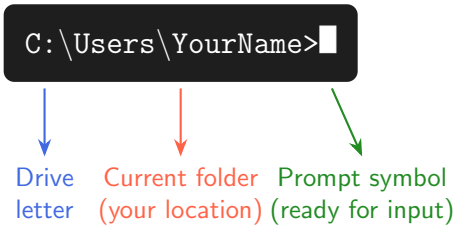
Figure 1: Opening the Command Prompt on Windows.

1.2 Anatomy of the Prompt

When the terminal opens, you'll see something like this:

```
C:\Users\YourName>
```

This is called the **prompt**. Let's break it down:

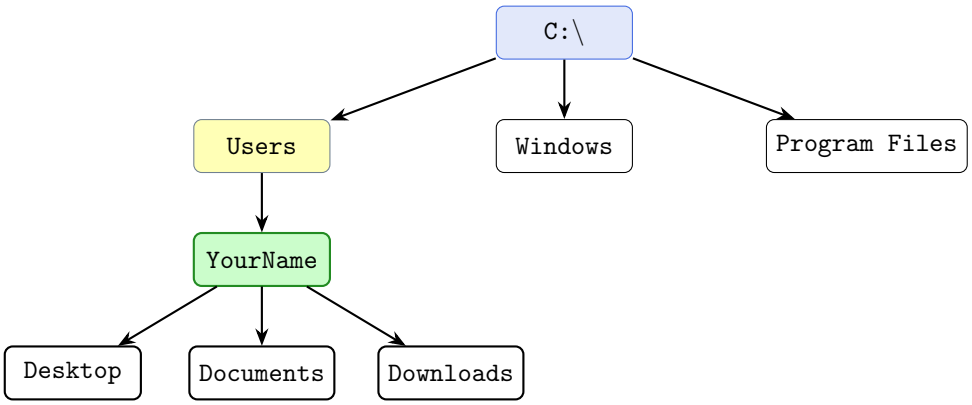


Big Idea

The prompt always shows you **where you are** in the file system. Think of it like a "You Are Here" marker on a map.

2 Navigating the File System

Your computer organizes files in **folders** (also called **directories**). These folders are nested inside each other, like boxes within boxes. The terminal lets you move between them.



2.1 Essential Navigation Commands

Command	Example	What it does
`cd`	`cd`	Shows your current directory (where you are)
`cd folder`	`cd Desktop`	Move into a folder
`cd ..`	`cd ..`	Go back one folder (to the parent)
`cd \`	`cd \`	Go to the root of the drive
`dir`	`dir`	List everything in the current folder
`cls`	`cls`	Clear the screen

Absolute vs. Relative Paths

- **Absolute path**: The full address from the root.
  Example: `C:\Users\YourName\Desktop`
- **Relative path**: The address from where you currently are.
  Example: If you're in `C:\Users\YourName`, just type `Desktop`.

Think of it like giving directions: "123 Main Street" (absolute) vs. "two doors down" (relative).

2.2 Exercises: Navigation

1. ★ Easy Type `cd` by itself and press Enter. What does it print? This is your **current directory**.
2. ★ Easy Type `dir` and press Enter. You should see a list of files and folders. How many folders do you see?
3. ★ Easy Type `cls` to clear the screen. The prompt is still there — the screen is just clean now.
4. ★★ Medium Navigate to your Desktop:

```
C:\Users\YourName> cd Desktop
C:\Users\YourName\Desktop>
```

Notice how the prompt changed to show your new location. Type `dir` to see what's on your Desktop.

5. ★★ Medium Go back to your user folder using `cd ..` (two dots means "go up one level"):

```
C:\Users\YourName\Desktop> cd ..
C:\Users\YourName>
```

6. ★★ Medium Navigate to Desktop using an **absolute path**. Type:

```
C:\Users\YourName> cd C:\Users\YourName\Desktop
```

This works no matter where you currently are.

7. ★★ Medium Navigate to `Documents`, then back to `Desktop`. Write down the two commands you used.

8. ★★★ Hard Starting from `C:\`, navigate to your Desktop using **only relative paths**, one step at a time. Write down every command. Then verify you're in the right place with `cd`.

3 Creating and Managing Folders

Now that you can navigate, let's learn to **create** and **organize** folders from the terminal.

3.1 Folder Commands

Command	Example	What it does
`mkdir name`	`mkdir projects`	Create a new folder
`rmdir name`	`rmdir projects`	Remove an *empty* folder
`mkdir a\b\c`	`mkdir a\b\c`	Create nested folders in one command

3.2 Exercises: Creating Folders

1. ★ Easy Navigate to your Desktop. Create a new folder:

```
C:\Users\YourName\Desktop> mkdir MyFirstFolder
```

Look at your Desktop — you should see the new folder appear!

2. ★ Easy Verify it exists by listing the contents: `dir`. Find `MyFirstFolder` in the list.

3. ★ Easy Enter the folder:

```
C:\Users\YourName\Desktop> cd MyFirstFolder
C:\Users\YourName\Desktop\MyFirstFolder>
```

4. ★★ Medium Inside `MyFirstFolder`, create a folder called `projects`. Then go into `projects` and create a folder called `comp111`.

5. ★★ Medium Now navigate from `comp111` all the way back to the Desktop in **one command**:

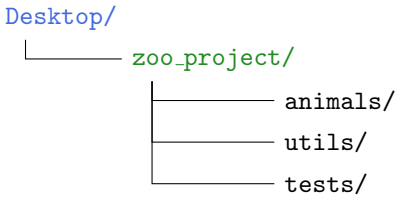
```
C:\...\comp111> cd ..\..\..
C:\Users\YourName\Desktop>
```

6. ★★ Medium Create nested folders in a single command:

```
C:\Users\YourName\Desktop> mkdir lab01\src\animals
```

Verify the structure was created by navigating into each folder.

7. ★★★ Hard Create the following **entire** folder structure from the terminal. You'll use this for the rest of the lab:



Hint: You can use `mkdir zoo_project\animals` *and similar commands, or create them one at a time.*

Tip

The folder structure you just created in Exercise 7 will be used throughout the rest of this lab. Keep your terminal open and stay on the Desktop!

4 Creating and Viewing Files

You can create files directly from the terminal — no text editor needed (though we'll use one for longer programs later).

4.1 File Commands

Command	What it does
`echo Hello > file.txt`	Create a file with the text "Hello" (overwrites if exists)
`echo Line2 >> file.txt`	Append text to an existing file
`type file.txt`	Display the contents of a file
`del file.txt`	Delete a file

Warning

Be careful with `>` vs `>>`: a single `>` **overwrites** the entire file! Use `>>` to **append** (add to the end).

4.2 Exercises: Creating Files

1. ★ Easy Navigate into the `zoo_project` folder. Create a text file:

```
C:\...\zoo_project> echo Zoo Management System > README.txt
```

View it: `type README.txt`

2. ★ Easy Append a second line:

```
C:\...\zoo_project> echo By: YourName >> README.txt
```

View it again with `type README.txt`. You should see both lines.

3. ★★ Medium Create a simple Python file:

```
C:\...\zoo_project> echo print("Welcome to the Zoo!") > main.py
```

View it with `type main.py` to make sure it looks correct.

4. ★★ Medium Navigate into the `animals` folder and create a file called `hello_animal.py` that prints `Hello, I am an animal!`. Then view it with `type`.

5. ★★★ Hard Without using any text editor, create a file called `info.txt` inside `zoo_project` that contains exactly three lines: your name, your roll number, and "Comp 111". Use `echo` commands. Verify with `type`.

5 Running Python from the Terminal

This is where terminal skills meet programming. You'll run every Python program in this lab from the terminal.

5.1 Two Ways to Use Python

Interactive Mode

Type `python` to start.
Type commands one at a time.
Great for quick experiments.
Type `exit()` to quit.

Script Mode

Write code in a `.py` file.
Run with `python filename.py`.
Great for real programs.
This is what you'll use most.

5.2 Exercises: Running Python

1. ★ Easy Try interactive mode. Type `python` to start, then:

```
C:\...\zoo_project> python
>>> print("Hello from Python!")
Hello from Python!
>>> 2 + 3
5
>>> exit()
```

2. ★ Easy Run the `main.py` file you created earlier:

```
C:\...\zoo_project> python main.py
Welcome to the Zoo!
```

3. ★★ Medium Navigate into `animals`. Create a file `adder.py` (using `echo`) that prints the sum of 10 and 25. Run it with `python adder.py`. Verify the output is 35.

4. ★★ Medium Using your preferred text editor, create a file `greet.py` inside `zoo_project` with:

```
name = input("What is your name? ")
print("Welcome to the Zoo, " + name + "!")
```

   Save it and run it from the terminal. The program should ask for your name and greet you.

5. ★★★ Hard Create a file `calculator.py` in `zoo_project\utils` that:

   - Asks the user for two numbers (use `input()` and `int()`)
   - Prints their sum, difference, and product

   Run it from the terminal and verify it works.

5.3 Command Reference

Here is a summary of every command you've learned. Keep this page bookmarked!

Terminal Command Reference Card

Navigation

`cd`	show current directory
`cd folder`	enter a folder
`cd ..`	go up one level
`cd \`	go to root
`dir`	list folder contents
`cls`	clear the screen

Folders

`mkdir name`	create folder
`rmdir name`	remove empty folder
`mkdir a\b`	create nested

Files

`echo text > f.txt`	create/overwrite
`echo text >> f.txt`	append
`type f.txt`	view file
`del f.txt`	delete file

Python

`python`	interactive mode
`python file.py`	run a script
`python --version`	check version
`exit()`	quit interactive

Paths

`.` current folder `..` parent folder `\` path separator

Absolute: `C:\Users\You\Desktop` Relative: `..\Documents`

6 Why Object-Oriented Programming?

So far, you may have written Python programs as a sequence of functions and statements. This is called **procedural programming**. It works fine for small scripts, but as programs grow, it becomes hard to manage. **Object-Oriented Programming (OOP)** gives us a better way to organize code.

Procedural

Data and functions are **separate**.

Functions operate on data passed to them.

Hard to manage as code grows.

```
name = "Simba"
age = 2
def speak(name):
    print(name + " meows")
```

Better!

Object-Oriented

Data and functions **bundled together** inside objects.

Objects manage their own data.

Scales well for large programs.

```
class Cat:
    def __init__(self, name, age):
        self.name = name
    def speak(self):
        print(self.name + " meows")
```

6.1 The Four Pillars of OOP

Encapsulation

Bundle data and methods together. Protect internal state.

Abstraction

Hide complex details. Show only what's needed.

Inheritance

Create new classes from existing ones. Reuse code.

Polymorphism

Same interface, different behaviors. Many forms.

Object-Oriented Programming

You have already seen all four of these in lectures. In this lab, you'll practice each one by writing real code.

7 Classes and Objects

A **class** is a blueprint. An **object** is a thing built from that blueprint. One class can create many objects, each with its own data.

Class: Student

```
name: ???
roll_no: ???
greet(): "Hi, I am ..."
```

(Blueprint / Template)

```
name:  "Zain"
roll_no:  101
```

```
name:  "Fatima"
roll_no:  102
```

```
name:  "Ahmed"
roll_no:  103
```

(Objects / Instances)

7.1 Walkthrough: Your First Class

Let's build a `Student` class step by step. Using your text editor, create a file called `student.py` inside the `zoo_project` folder.

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

    def greet(self):
        print("Hi, I am " + self.name + " (" + str(self.roll_no) + ")
            ")

# --- Create objects and test ---
s1 = Student("Zain", 101)
s2 = Student("Fatima", 102)

s1.greet()
s2.greet()
```

Run it from the terminal:

```
C:\...\zoo_project> python student.py
Hi, I am Zain (101)
Hi, I am Fatima (102)
```

Refer to the lecture slides for a detailed breakdown of this syntax.

Big Idea

`__init__()` is called **automatically** when you create an object.

`self` refers to the object that is calling the method.

7.2 Exercises: Classes and Objects

1. ★ Easy Create a file `point.py`. Define a class `Point` with `__init__(self, x, y)` that stores two coordinates. Create a point `p = Point(3, 4)` and print `p.x` and `p.y`. Run it from the terminal.

2. ★ Easy Add a method `display(self)` to `Point` that prints `(x, y)` using an f-string. Call it on your point.

3. ★ Easy Create **two** different Point objects and call `display()` on each. Verify they have different data.

4. ★★ Medium Add a `gpa` attribute to your `Student` class with a default value of `0.0`. Add methods `set_gpa(self, gpa)` and `get_gpa(self)`. Create a student, set their GPA to 3.7, and print it.

5. ★★ Medium Add a method `is_on_dean_list(self)` to `Student` that returns `True` if GPA is ≥ 3.5. Test it with two students: one with GPA 3.8 (should be True) and one with GPA 2.9 (should be False).

6. ★★ Medium Create a `Book` class in a new file `book.py` with attributes `title`, `author`, and `price`. Add a method `discounted_price(self, percent)` that returns the price after a discount. Test with a book priced at 500, with a 20% discount (expected: 400.0).

7. ★★ Medium Create a `Rectangle` class with `length` and `width`. Add methods `area()` and `perimeter()`. Create two rectangles (3×4 and 5×6) and print both their areas and perimeters. Run from terminal.

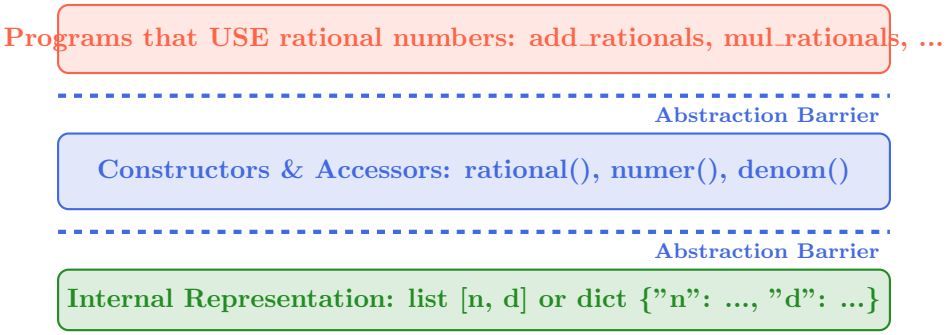
8. ★★★ **Hard** Create a `BankAccount` class in `bank.py` with:

- Attributes: `owner` (string) and `balance` (starts at 0)
- Method `deposit(amount)`: adds to balance
- Method `withdraw(amount)`: subtracts from balance, but **prints a warning and does nothing** if the withdrawal would make balance negative
- Method `__str__(self)`: returns `"Owner: X, Balance: Y"`

Test all methods, including trying to withdraw more than the balance.

8 Data Abstraction and Encapsulation

In the lectures, you learned that **abstraction** means hiding details and showing only what's necessary, and **encapsulation** means bundling data and methods together. The key idea is the **abstraction barrier**: users of a class should only interact with it through its methods, never by touching internal data directly.



Big Idea

An **Abstraction Barrier Violation** happens when a higher-level function bypasses the proper interface to directly access lower-level implementation details.

8.1 Exercises: Abstraction and Encapsulation

For the following exercises, use this `Point` class (save it as `point.py`):

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def get_x(self):
        return self.x

    def get_y(self):
        return self.y

    def __str__(self):
        return f"({self.x}, {self.y})"
```

1. ★ **Easy** Create two points `p1 = Point(1, 2)` and `p2 = Point(4, 6)`. Print their coordinates using the **getter methods** (`get_x()`, `get_y()`).

2. ★ **Easy** Write a function `distance(p1, p2)` that computes the distance between two points **using the getter methods** (not `p1.x` directly). Test it with `(0,0)` and `(3,4)` — the answer should be 5.0.

> **Tip**
>
> Distance formula:
> $d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$.
> In Python: `((x2-x1)**2 + (y2-y1)**2) ** 0.5`

3. ★★ **Medium** The following code has an **abstraction barrier violation**. Identify the violation and rewrite the code correctly:

```
p = Point(3, 4)
result = (p.x ** 2 + p.y ** 2) ** 0.5    # What's wrong here?
print("Distance from origin:", result)
```

4. ★★ **Medium** Create a `Rational` class in `rational.py`:

- `__init__(self, numer, denom)`: stores numerator and denominator in **simplest form** (use `math.gcd`)
- `get_numer(self)`, `get_denom(self)`: accessor methods
- `__str__(self)`: returns `"n/d"`

Test: `Rational(6, 9)` should print as `2/3`.

5. ★★ **Medium** Add a method `add(self, other)` to `Rational` that returns a **new** Rational representing the sum. Use only the accessor methods (don't access `other.numer` directly). Test: $\frac{1}{2} + \frac{1}{3} = \frac{5}{6}$.

6. ★★★ **Hard** Create a `Circle` class that uses **composition** — its center is a `Point` object:

```
class Circle:
    def __init__(self, center, radius):
        # center is a Point object
        ...
```

Add a method `contains_point(self, point)` that returns `True` if the given point is inside the circle. **Important:** use your `distance()` function — do **not** directly access `.x` or `.y` on the Point.

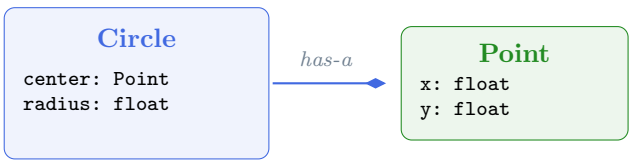
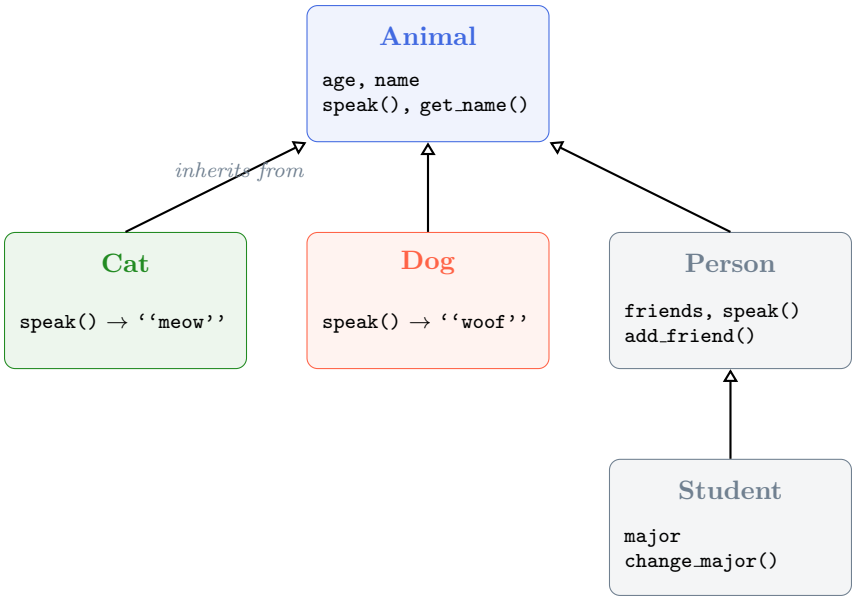


Figure 2: Composition: Circle *has-a* Point as its center.

9 Inheritance

Inheritance lets you create new classes based on existing ones. The new class (child/subclass) gets all the attributes and methods of the existing class (parent/superclass) and can add new ones or override existing ones.



9.1 Walkthrough: Building the Animal Hierarchy

Create a file called `animal.py` inside your `zoo_project\animals` folder:

```
class Animal:
    def __init__(self, age):
        self.age = age
        self.name = None

    def get_age(self):
        return self.age

    def get_name(self):
        return self.name

    def set_name(self, name):
        self.name = name

    def speak(self):
        print("...")

    def __str__(self):
        return f"animal:{self.name}-{self.age}"
```

Now let’s create a subclass. Add this below (in the same file or a new one):

```
class Cat(Animal):              # Cat inherits from Animal
    def speak(self):            # Override speak()
        print("meow")

    def __str__(self):
        return f"cat:{self.name}-{self.age}"

# Test
c = Cat(2)                      # Uses Animal's __init__
c.set_name("Simba")             # Uses Animal's set_name
c.speak()                       # Uses Cat's speak (overridden)
print(c)                        # Uses Cat's __str__ (overridden)
```

```
meow
cat:Simba-2
```

Big Idea

A subclass can **use** parent attributes, **override** parent methods, or define **new** attributes and methods.

9.2 Exercises: Inheritance

1. ★ Easy Using the `Animal` class above, create a `Dog` subclass that overrides `speak()` to print `"woof"`. Create a Dog, set its name to `"Buddy"`, and make it speak. Run from terminal.

2. ★ Easy Create a Dog and call `get_name()` and `get_age()` on it. These methods are inherited from `Animal` — you didn't have to write them again!

3. ★ Easy Try calling `speak()` on a plain `Animal` object (not a Cat or Dog). What does it print?

4. ★★ Medium Create a `Person` class that inherits from `Animal`. Override `__init__` to accept both `name` and `age`. Call `Animal.__init__(self, age)` inside it. Add a `friends` list and an `add_friend(self, friend_name)` method. Test by adding two friends and printing the list.

5. ★★ Medium Create a `Student` class that inherits from `Person`. Add a `major` attribute. Override `__str__` to include the major. Create a student, change their major, and print them.

6. ★★ Medium Create an `Employee` class with `name` and `salary`. Create a `Manager` subclass that adds a `bonus`. Override `calculate_pay(self)` so Manager returns salary + bonus. Test with an Employee (salary 50000) and a Manager (salary 60000, bonus 10000).

7. ★★ Medium Write code that demonstrates a parent object **cannot** call a child's method. Create an `Animal` and try to call a method that only exists in `Cat`. What error do you get? Write the error as a comment in your code.

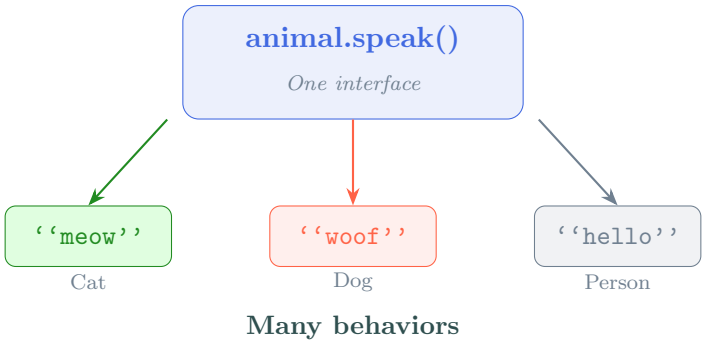
8. ★★★ Hard Create a `Vehicle` class with `make`, `model`, and `year`. Create `Car(Vehicle)` that adds `num_doors` and `Truck(Vehicle)` that adds `payload_capacity`. Both override `__str__`. Add a **class variable** `vehicle_count` that tracks how many vehicles have been created total (incremented in `Vehicle.__init__`). Create 3 cars and 2 trucks, then print `Vehicle.vehicle_count` (should be 5).

Class Variables vs Instance Variables

A **class variable** is shared by all instances. It's defined directly in the class body (not inside `__init__`). An **instance variable** is unique to each object and is defined with `self.something` in `__init__`.

10 Polymorphism

Polymorphism (from Greek: "many forms") means that the **same method name** can behave **differently** depending on the object that calls it. This is incredibly powerful: you can write one function that works with many different types.



10.1 Walkthrough: Polymorphism in Action

Without polymorphism, you'd have to check types manually (tedious and error-prone):

```
# BAD: Without polymorphism
for animal in animals:
    if isinstance(animal, Cat):
        animal.meow()
    elif isinstance(animal, Dog):
        animal.bark()
    # ... need a new elif for EVERY new type!
```

With polymorphism, it's simple:

```
# GOOD: With polymorphism
for animal in animals:
    animal.speak()  # Python calls the correct version automatically!
```

10.2 Exercises: Polymorphism

1. ★ **Easy** Create a list of animals and make them all speak:

```
animals = [Cat(2), Dog(3), Cat(1)]
for animal in animals:
    animal.speak()
```

Run it. You should see: `meow`, `woof`, `meow`.

2. ★ **Easy** Write a function `make_all_speak(animals)` that takes a list of animals and calls `speak()` on each one. Test it with a list containing cats, dogs, and persons.

3. ★★ **Medium** Create a `Shape` hierarchy:

- `Shape` base class with a method `area()` that returns 0
- `Rectangle(Shape)` with `width`, `height` — override `area()`
- `Circle(Shape)` with `radius` — override `area()` (use $\pi = 3.14159$)
- `Triangle(Shape)` with `base`, `height` — override `area()`

Write a function `total_area(shapes)` that returns the sum of areas of all shapes in a list. Test:

```
shapes = [Rectangle(4, 5), Circle(3), Triangle(6, 8)]
print(total_area(shapes))  # 20 + 28.27... + 24 = 72.27...
```

4. ★★ Medium Write a `print_payroll(employees)` function that works with `Employee`, `Manager`, and `Salesperson` objects (from the inheritance exercises). It should print each person's name and pay, then the total payroll. All through the same `calculate_pay()` interface.

5. ★★ Medium **(Duck Typing)** Create three **unrelated** classes — `Dog`, `Robot`, `Person` — none inheriting from each other, but each has a `speak()` method. Write a function `make_noise(thing)` that calls `thing.speak()`. Demonstrate it works with all three types.

Duck Typing

Python doesn't care about the *type* of an object — only whether it has the right *methods*. "If it walks like a duck and quacks like a duck, it must be a duck." This means objects don't even need to share a parent class to be used polymorphically!

6. ★★★ Hard Create an **Abstract Base Class** `Drawable` using Python's `abc` module:

```
from abc import ABC, abstractmethod

class Drawable(ABC):
    @abstractmethod
    def draw(self):
        pass
```

Create subclasses `Square`, `RightTriangle`, and `Line`, each implementing `draw()` to print an ASCII art representation. For example:

```
Square(3):          RightTriangle(3):     Line(5):
* * *                *                    * * * * *
* * *                * *
* * *                * * *
```

Write `draw_all(shapes)` that draws each shape. Also show that `Drawable()` by itself raises a `TypeError`.

11 Putting It All Together: Zoo Management System

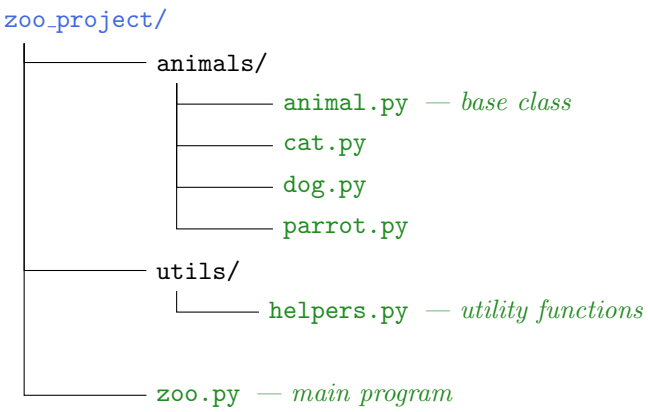
Time to combine everything! In this final exercise, you'll use your **terminal skills** to set up a project and your **OOP skills** to build a small system.

Challenge

Try to do as much as possible from the terminal: navigating folders, running Python files, checking output.

11.1 The Task

Build a Zoo Management System using the folder structure you created earlier:



11.2 Step-by-Step Instructions

Step 1. Set up the files. From the terminal, navigate to `zoo_project`. Create any missing files using your text editor.

Step 2. Create the base class (`animals/animal.py`):

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

    def speak(self):
        print("...")

    def info(self):
        return f"{self.name} (age {self.age})"

    def __str__(self):
        return self.info()
```

Step 3. Create at least 3 subclasses (e.g., `cat.py`, `dog.py`, `parrot.py`). Each should:

- Import `Animal`: `from animal import Animal`
- Override `speak()` with a unique sound
- Override `info()` to include the animal type

Step 4. Create a helper function (`utils/helpers.py`):

```python
def morning_routine(animals):
    """Make all animals speak and print their info."""
    print("=== Good Morning, Zoo! ===")
    for animal in animals:
        print(animal.info(), "says:", end=" ")
        animal.speak()
    print("============================")
```

Step 5. Create the main program (`zoo.py`):

- Import all your animal classes and the helper function
- Create a list of at least 5 animals (mix of types)
- Call `morning_routine(animals)`

Step 6. Run it! From the terminal:

```
C:\...\zoo_project> python zoo.py
=== Good Morning, Zoo! ===
Simba (age 3) says: meow
Buddy (age 5) says: woof
Polly (age 10) says: squawk!
Whiskers (age 1) says: meow
Rex (age 4) says: woof
==========================
```

Import Hint

Since your files are in subfolders, you may need to adjust imports. The simplest approach for now: copy all `.py` files into the same folder, or add `import sys; sys.path.append("animals")` at the top of `zoo.py`.

Congratulations!

You've completed Lab 01! Here's what you've accomplished:

Skills Unlocked

Terminal
- ✓ Navigate folders with `cd`
- ✓ Create folders with `mkdir`
- ✓ Create/view files
- ✓ Run Python scripts

OOP
- ✓ Classes and Objects
- ✓ Abstraction & Encapsulation
- ✓ Inheritance
- ✓ Polymorphism

Coming Up Next

Git version control — and you'll be ready for it because you already know how to use the terminal!