Object-Oriented Programming

Fundamentals for Python Development

Python Class Methods

Understanding Instance, Class and Static Methods

Instance Methods

Work with specific object data using 'self'

Class Methods

Operate on class-level data using 'cls'

Static Methods

Utility functions within class namespace

What are Methods? Methods are functions defined inside a class that represent the behaviors and actions that objects can perform. Python offers three distinct types of methods, each serving different purposes.

Instance Methods

The Standard Method - Operation on a Specific Object

Core Concept: Instance methods are bound to specific instances (objects) of the class. They can access and modify the object's unique data through the 'self' parameter.

🔑 The Magic of 'self'

- First argument of every instance method is always `self`
- Provides reference to the specific object calling the method
- Allows access to instance attributes (e.g., self.name, self.age)
- Use when method needs to know about object's state

Class Methods

Operation on a Class Itself

> **Core Concept:** Class methods are bound to the class, not instances. They work with class-level data and are commonly used for factory methods that create instances in alternative ways.

🎯 Key Features

- Decorated with `@classmethod`
- First argument is `cls` (reference to the class)
- Cannot access instance-specific data
- Perfect for alternative constructors

```python
class Person:
    # Class attribute
    species = "Homo sapiens"

    def __init__(self, name, age):
        self.name = name
        self.age = age

    # Factory class method
    @classmethod
    def from_birth_year(cls, name, birth_year):
        current_year = 2025
        age = current_year - birth_year
        return cls(name, age)  # Same as Person(name, age)

# Usage
person1 = Person("Riya", 22)                     # Regular way
person2 = Person.from_birth_year("Karan", 2001)  # Factory method
```

Static Methods

Utility Functions in Class Namespaces

Core Concept: Static methods are not bound to instances or classes. They're regular functions grouped with a class for organizational purposes and logical association.

⚡ Characteristics

- Decorated with `@staticmethod`
- No self or `cls` parameter
- Cannot access instance or class data
- Used for utility/helper functions

```
class MathUtils: # Static method for mathematical operations @staticmethod def add(x, y): return x + y @staticmethod def
is_positive(num): return num > 0 # Call directly on the class result = MathUtils.add(5, 10) # Output: 15 print(f"Result is: {result}")
print(f"Is 10 positive? {MathUtils.is_positive(10)}")
```

Methods Types Comparison

Understanding the Key Differences

Feature	Instance Method	Class Method	Static Method
Decorator	None	@classmethod	@staticmethod
First Argument	self (object instance)	cls (class itself)	None
Access Level	Instance + Class state	Class state only	No state access
Main Purpose	Operate on specific objects	Factory methods, class operations	Utility functions
When to Use	Need object's data	Alternative constructors	Helper functions

Quick Decision Guide:

✓ **Instance method:** When you need access to object's specific data

✓ **Class method:** When creating objects or working with class-level data

✓ **Static method:** When function is related but independent of object/class state

Hands-on Exercise

The Ultimate Car class Challenge

🚗 Your Mission

Create a **Car class** for a dealership's inventory management system.

📋 Requirements

Class Attribute:

- `total_cars` counter starting at 0
- Increment on each new car creation

Constructor:

- Initialize make, model, mileage
- Increment total_cars counter

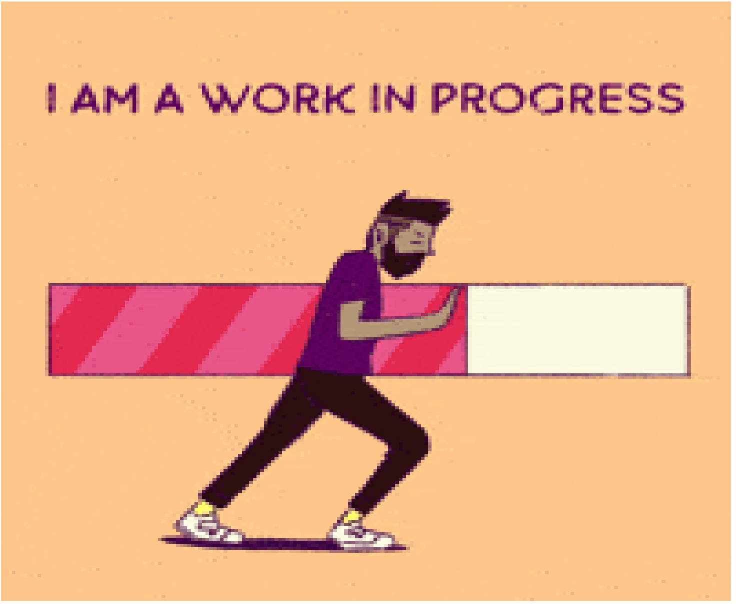
Methods:

- `display_details()` - Instance method
- `from_dict()` - Class method factory
- `convert_km_to_miles()` - Static utility

Conversion: 1 km = 0.621371 miles

Solutions

Complete Car Class Implementation



Solution is in **typeofmethods.py**

Key Takeaways

Essential Points to Remember

Instance Methods

Most common type. Use when you need to access or modify specific object data. Always include 'self' parameter.

Class Methods

Perfect for factory patterns and alternative constructors. Use @classmethod decorator and 'cls' parameter.

Static Methods

Independent utility functions. Use @staticmethod decorator. No access to instance or class state.

Quick Reference Rules:

- ✓ **Need object data?** → Use Instance Method
- ✓ **Need to create objects differently?** → Use Class Method
- ✓ **Independent utility function?** → Use Static Method
- ✓ **When in doubt,** start with Instance Method

🎉 Congratulations!

You've mastered Python Class Methods. Ready to build amazing object-oriented applications!

Object-Oriented Programming

Fundamentals for Python Development

FIL Fresher Training Program

> *"Think of OOP as creating 'ingredients' (objects) with their own properties and behaviors, making your code more organized and reusable—just like in the real world!"*

Classes & Objects

Class: A blueprint for creating objects

Object: An instance created from that blueprint

```
# Blueprint (Class) class Dog: def bark(self):
print("Woof!") # Objects (Instances) dog1 = Dog()
dog2 = Dog()
```

Constructor & Self

`__init__` runs automatically when creating objects

`self` refers to the current instance

```
class Dog: def __init__(self, name, breed):
self.name = name self.breed = breed def bark(self):
print(f"{self.name} says: Woof!")
```

Core OOP Pillars

💊 Encapsulation

Bundling data and methods into a single unit (class) while protecting internal data from external interference

👪 Inheritance

Creating new classes that inherit attributes and methods from existing classes, promoting code reuse and hierarchy

🔄 Inheritance in Action

Dog
• name, breed
• bark()

→

GuideDog
• Inherits all from Dog
• + guide() method

```
class GuideDog(Dog): # Inherits from Dog def guide(self): print(f"{self.name} is guiding the way.") sunny =
GuideDog("Sunny", "Labrador") sunny.bark() # From parent Dog class sunny.guide() # From GuideDog class
```

🏆 Hands-on Exercise: Create a Car Class

1. Create a `Car` class with `__init__` constructor
2. Add three attributes: `make` , `model` , `year`
3. Create `display_info()` method to print car details
4. Create two different Car objects and test the method

```python
# 1. Create the class
class Car:
    # 2. Create the constructor with three attributes
    def __init__(self, make, model, year):
        self.make = make
        self.model = model
        self.year = year

    # 3. Create the display_info method
    def display_info(self):
        print(f"This is a {self.year} {self.make} {self.model}.")

# 4. Create two Car objects and call their methods
car1 = Car("Tata", "Nexon", 2023)
car2 = Car("Hyundai", "Creta", 2024)

car1.display_info()
car2.display_info()
```

FIL - Fresher Training Program

A Deeper Dive into Encapsulation 💊

Object-Oriented Programming Fundamentals

Encapsulation is the practice of bundling an object's data (attributes) and the methods that operate on that data into a single unit (the class). Think of it as creating a protective barrier around your data.

🏦 Real-World Analogy: Bank Account

Imagine a bank account. You can't just walk into the bank's database and change your balance to a million dollars. Instead, you have to use approved methods like an ATM `deposit()` or `withdraw()` function. These methods have built-in rules (like not letting you withdraw more than you have). That's encapsulation in the real world!

The Problem vs The Solution

❌ **The Wrong Way - Unprotected**

```python
class BankAccount:
    def __init__(self, owner, balance=0):
        self.owner = owner
        # Public attribute - anyone can change it!
        self.balance = balance

    def display_balance(self):
        print(f"Owner: {self.owner}")
        print(f"Balance: ${self.balance:,.2f}")

# Create account
account = BankAccount("Ravi Kumar", 1000)

# The problem: Direct access
account.balance = -500  # This shouldn't be possible!
account.display_balance()  # Balance: $-500.00
```

✅ **The Right Way - Encapsulated**

```python
class BankAccount:
    def __init__(self, owner, balance=0):
        self.owner = owner
        # Private attribute (underscore convention)
        self._balance = max(0, balance)

    def get_balance(self):
        return self._balance

    def deposit(self, amount):
        if amount > 0:
            self._balance += amount
            print(f"Deposited ${amount:,.2f}")
        else:
            print("Deposit must be positive")

    def withdraw(self, amount):
        if 0 < amount <= self._balance:
            self._balance -= amount
            print(f"Withdrew ${amount:,.2f}")
        else:
            print("Invalid withdrawal")
```

✍️ Hands-on Exercise: Student Class

Create a Student class with proper encapsulation:

✓ Constructor takes `name` and `marks` parameters

✓ Make `marks` private using underscore (`_marks`)

✓ Create `get_marks()` getter method

✓ Create `set_marks(new_marks)` setter method

✓ Validate marks are between 0 and 100

✓ Test with both valid and invalid values

🎯 Key Takeaways for Encapsulation

🎯 Key Takeaways for Encapsulation

Bundling & Hiding

Bundle data and methods together while hiding internal complexity from the outside world.

Control & Safety

Control how object data is modified. Use setter methods with validation to prevent bugs and maintain consistent state.

Flexibility

Makes code easier to change. Update logic inside class methods without affecting external code.

What is Inheritance?

Building on What Already Exists

Core Concept

Inheritance allows a new class (the **Child Class** or **Subclass**) to inherit attributes and methods from an existing class (the **Parent Class** or **Superclass**).

Real-World Analogy: *Think of it like a family tree. A child inherits traits from their parent, who inherited traits from their grandparent.*

Why use inheritance?

- For code reusability and organization
- Follows the DRY principle: **Don't Repeat Yourself**
- Creates logical hierarchies in your code structure
- Promotes maintainable and scalable software design

Method Overriding

Doing Things Your Own Way

Method Overriding Concept

A child class can provide its own specific implementation of a method that it inherited from its parent class. This is **method overriding**.

Example: *Both a Car and a Bicycle are Vehicles and can move(). But how they move is different. The Car class would override the move() method to use an engine, while Bicycle would override it to use pedals.*

```
class Vehicle: def move(self): print("Vehicle is moving.") class Car(Vehicle): # This overrides the parent's move() method def move(self): print("Car is driving on the road.")
```

The super() Function

How to Access the Parent's Logic

The Problem & Solution

Problem: What if you override a method, but you still need to run the original code from the parent's method?

Solution: The **super()** function! It gives you a way to call methods from the parent class.

Why is this amazing?

• It lets you extend the parent's functionality, not just replace it

• You can run the parent's logic and then add your own specific code

```
class Parent: def __init__(self): print("Parent constructor called.") class Child(Parent): def
__init__(self): print("Child constructor starting...") super().__init__() # This calls the Parent's
__init__ method print("Child constructor finished.")
```

```
Output:
Child constructor starting...
Parent constructor called.
Child constructor finished.
```

Multi-Level Inheritance

Grandparents, Parents, and Children

Multi-Level Inheritance Concept

A class can inherit from a child class, creating a "grandchild" relationship. This is called **multi-level inheritance**.

Structure: A → B → C (C inherits from B, and B inherits from A)

The Confusing Part: When you use super() in class C, which method does it call?

The Rule

super() doesn't just call the "parent." It calls the next method in the **Method Resolution Order (MRO)**. In a simple chain like this, it effectively calls the method from the immediate parent (B).

The Chain Reaction

If C calls super().method(), it runs B's method. If B's method also has super().method(), it will then run A's method. This creates a **chain of execution** from child up to the highest ancestor.

Hands-on Exercise

The Evolving Product Challenge

Scenario

You are modeling different types of digital products for an online store. There is a base product, a more specific software product, and a very specific subscription-based software. Each one builds on the last.

Your Task

Create three classes with a multi-level inheritance structure:

DigitalProduct → Software → SubscriptionSoftware

Learning Goal: *This exercise will help you understand how super() creates a chain of method calls and how each class level can extend functionality while reusing parent class logic.*

Exercise Instructions

Step-by-Step Implementation Guide

Class 1: DigitalProduct (Grandparent)

1. __init__ constructor should accept **name** and **base_price**
2. Create a method **display_info()** that prints the product's name and price in a formatted way

Class 2: Software (Parent)

1. Must inherit from DigitalProduct
2. __init__ should accept name, base_price, and **license_key**
3. Use **super()** to call the parent's constructor
4. Override display_info(): call super().display_info() first, then print license_key

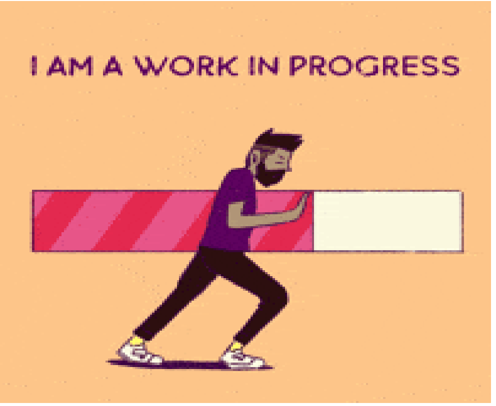
Exercise Instructions

Step-by-Step Implementation Guide

Class 3: SubscriptionSoftware (Child)

1. Must inherit from Software
2. __init__ should accept name, base_price, license_key, and **subscription_period**
3. Use super() to call parent's constructor
4. Override display_info(): call super().display_info() first, then print subscription_period

```
Expected Output Pattern:
Product Name: Pro Image Editor
Price: $99.99
License Key: ABCD-1234-EFGH-5678
Subscription: yearly
```



🎭 Polymorphism in Python

Object-Oriented Programming Concepts

One Name, Many Forms

🎯 *Training Objective*

Master the concept of Polymorphism to write flexible, maintainable Python code that adapts to different object types seamlessly.

What you'll learn today:

- ✨ Method Overloading
- 🔄 Method Overriding
- ➕ Operator Overloading
- 🦆 Duck Typing

🎭 Definition

Polymorphism comes from Greek: **"poly" (many)** + **"morph" (form)**

In programming: **A single interface can work with objects of different types, each responding in their own specific way.**

🎵 ***Real-World Analogy: The Verb "PLAY"***

- 🎸 *A **musician** can play an instrument*
- 🎭 *An **actor** can play a role*
- ⚽ *A **child** can play a game*
- 🎬 *A **media player** can play a video*

Same action word, different behaviors based on context!

Why is this powerful in programming?

🎯 Benefits of Polymorphism

- ✓ **Code Reusability:** Write once, use with multiple types
- ✓ **Flexibility:** Easy to extend with new types
- ✓ **Maintainability:** Changes in one place affect the whole system
- ✓ **Abstraction:** Focus on what objects do, not how they're implemented

Python's Approach to Method Overloading

Traditional Definition: Having multiple methods with the same name but different parameters.

Python's Reality: Python doesn't support traditional method overloading. The last defined method wins!

🐍 The Pythonic Solution

We achieve similar functionality using **default arguments** and ***args** :

🐍 Python Method Overloading Example

```
class Calculator: # This single method can act like add(a, b) or add(a, b, c) def add(self, a, b, c=0): return a + b + c # Usage demonstration calc = Calculator() # Calling it in two "forms" print(f"Two args: {calc.add(5, 10)}") # Output: Two args: 15 print(f"Three args: {calc.add(5, 10, 20)}") # Output: Three args: 35
```

🎯 Key Takeaway

- ✓ One method definition handles multiple use cases
- ✓ More flexible than traditional overloading
- ✓ Cleaner, more maintainable code

Method Overriding in Action

Concept: A child class provides its own specific implementation of a method already defined in the parent class.

Shape Hierarchy Example

```
class Shape: def area(self): print("I am a generic shape. I don't have an area.") class Square(Shape): def __init__(self, side): self.side = side # Overriding the parent's area method def area(self): print(f"The area of the square is {self.side * self.side}.") class Circle(Shape): def __init__(self, radius): self.radius = radius # Overriding the parent's area method def area(self): print(f"The area of the circle is {3.14 * self.radius ** 2}.") # Polymorphism in action! shapes = [Square(5), Circle(10)] for shape in shapes: shape.area() # Same method call, different behavior!
```

🌟 The Magic of Method Overriding

- ✓ Same interface (method name) across different classes
- ✓ Each class implements behavior specific to its needs
- ✓ Enables writing generic code that works with multiple types

Custom Behaviour for Built-in Operators

You can define how standard operators like **+, -, *, <, >** work with your custom objects using **"dunder" methods** (double underscore).

🔑 Important Dunder Methods

__add__(self, other) : Defines behavior for the + operator

__str__(self) : Defines what print() shows (most useful!)

Point Class with Operator Overloading

```
class Point: def __init__(self, x, y): self.x = x self.y = y # Defines the '+' operator for Point objects def __add__(self, other): return Point(self.x + other.x, self.y + other.y) # Defines how the object should be printed def __str__(self): return f"Point({self.x}, {self.y})" # Usage p1 = Point(1, 2) p2 = Point(3, 4) p3 = p1 + p2 # Behind the scenes: p1.__add__(p2) print(p1) # Calls p1.__str__() → Point(1, 2) print(p2) # Point(3, 4) print(p3) # Point(4, 6)
```

✨ Benefits of Operator Overloading

- ✓ Makes your objects behave like built-in types
- ✓ Code becomes more intuitive and readable
- ✓ Natural syntax for complex operations

Python's Core Philosophy

🦆 ***The Duck Test***

"If it walks like a duck and it quacks like a duck, then it must be a duck."

In Programming: Python doesn't care about an object's type—it cares about its **behavior** (what methods it has).

Duck Typing in Action

```
class Dog: def speak(self): return "Woof!" class Cat: def speak(self): return "Meow!" class Duck: def speak(self): return "Quack!" # This function works with ANY object that has a .speak() method def make_it_speak(animal): print(animal.speak()) # Create different, unrelated objects dog = Dog() cat = Cat() duck = Duck() # Pass them all to the same function make_it_speak(dog) # Output: Woof! make_it_speak(cat) # Output: Meow! make_it_speak(duck) # Output: Quack!
```

🎯 Duck Typing Power

- ✓ Functions work with any object that "speaks the language"
- ✓ No inheritance required between classes
- ✓ Maximum flexibility and code reuse
- ✓ Focus on interface, not implementation

Python's Core Philosophy

Ask the output of the code in **python-core-philospy.py**

✍️ Hands-on Exercise

Build a Media Player System

🎯 Your Challenge: Create a Media Player System

Scenario: Build a system that handles different media files and playlists, demonstrating all polymorphism concepts.

Requirements:

1. **MediaFile Base Class:** title, duration, play(), __str__()
2. **AudioFile & VideoFile:** Override play() method
3. **Playlist Class:** Operator overloading for +, Duck typing for play_all()
4. **Test all features**

🔍 Show Solution

✍️ Hands-on Exercise

Build a Media Player System

🎓 Key Takeaways

- ✓ **Polymorphism = Flexibility:** Write code that works with multiple types
- ✓ **Overriding:** Same interface, different implementations
- ✓ **Operator Overloading:** Make objects work naturally with Python operators
- ✓ **Duck Typing:** Focus on behavior, not type hierarchy

🌟 What We've Learned

🔧 Method Overloading

- ✓ Use default parameters
- ✓ Leverage *args and **kwargs
- ✓ One method, multiple behaviors

🔄 Method Overriding

- ✓ Child classes customize parent behavior
- ✓ Maintain consistent interfaces
- ✓ Enable polymorphic collections

➕ Operator Overloading

- ✓ Use dunder methods (__add__, __str__)
- ✓ Make objects behave naturally
- ✓ Improve code readability

🦆 Duck Typing

- ✓ Focus on object capabilities
- ✓ No inheritance required
- ✓ Maximum flexibility

What is Abstraction? 👻

Hiding Complexity, Showing Only What's Necessary

Abstraction means hiding the complex implementation details of an object and only exposing the essential features the user needs to interact with.

The Perfect Analogy: A Car Dashboard 🚗

To drive a car, you use a simple interface: a steering wheel, accelerator, and brake pedal.

You don't need to know the complex details of:

- The engine's combustion cycle
- The transmission's gear ratios
- The braking system's hydraulics

Abstraction hides this complexity, making the car easy to use.

The Need for Abstraction in Design

Creating a Contract for Your Code

In large software projects, you want to ensure that different parts of the system work together consistently. Abstraction lets you define a **"contract"** or a **"blueprint"**.

This contract specifies *what* a set of related classes must be able to do, without dictating *how* they must do it.

Why is this useful?

⚖️

Enforces Consistency

Guarantees that every class of a certain type will have the same essential methods.

🎯

Reduces Complexity

Programmers can use objects without needing to know their complex inner workings.

🔧

Improves Maintainability

You can change the inner workings of one class without breaking the code that uses it.

Abstract Base Classes (ABC)

The Blueprint That Can't Be Built

Python provides the **abc** module to implement abstraction. ABC stands for **Abstract Base Class**.

> An Abstract Base Class is a special type of class that is meant to be a **blueprint** for other classes.
>
> **Key Rule:** You cannot create an object (an instance) directly from an abstract class. Trying to do so will result in an error.

You create an ABC by inheriting from `ABC`.

Python

```python
from abc import ABC, abstractmethod # 'Payable' is an abstract class. It defines a contract. class Payable(ABC): # This class cannot be instantiated directly. # For example: p = Payable() would raise a TypeError. pass
```

Interfaces using @abstractmethod 📜

Forcing Child Classes to Follow the Rules

An **abstract method** is a method that is declared in an abstract class but has no implementation. It's just a name and a set of parameters.

We use the `@abstractmethod` decorator to define one.

> **The Golden Rule:** Any regular (concrete) class that inherits from an ABC must provide an implementation for all of its parent's abstract methods.
>
> If a child class fails to implement even one abstract method, it also becomes an abstract class, and you cannot create objects from it either.

Interfaces using @abstractmethod 📜

Forcing Child Classes to Follow the Rules

```python
from abc import ABC, abstractmethod


class Payable(ABC):
    """
    An abstract base class that defines the contract for any class
    that can process a payment.
    """
    @abstractmethod
    def process_payment(self, amount):
        """
        This is the contract. Any class that inherits from Payable
        MUST implement this method.
        """
        pass


class CreditCardPayment(Payable):
    """A concrete class that processes payments via credit card."""
    # This class MUST implement the abstract method from the parent.
    def process_payment(self, amount):
        print(f"Charging ${amount} to the credit card.")


class UpiPayment(Payable):
    """A concrete class that processes payments via UPI."""
    # This class also MUST implement the abstract method.
    def process_payment(self, amount):
        # Assuming a conversion rate for demonstration
        inr_amount = amount * 83
        print(f"Processing UPI payment of ₹{inr_amount:,.2f}.")
```

✍️ Hands-on Exercise 12

The Notification System

Scenario: You're designing a system that can send notifications through various channels (Email, SMS, etc.). You want to ensure that any new notification channel you add in the future will work seamlessly with the rest of your system.

Your Task:

Create an abstract base class `NotificationSender` and three concrete classes that implement its contract.

Instructions:

1. **Create the Abstract Base Class NotificationSender:**
   - Must inherit from ABC
   - Should have one abstract method called `send(message)`
2. **Create Concrete EmailSender Class:**
   - Must inherit from NotificationSender
   - Must implement `send(message)` method
3. **Create Concrete SMSSender Class:**
   - Must inherit from NotificationSender
   - Must implement `send(message)` method
4. **Create Concrete PushNotificationSender Class:**
   - Must inherit from NotificationSender
   - Must implement `send(message)` method

Complete Solution

The Notification System Implementation

Follow the solution in **abstraction-example.py**

Graceful Error Management: An Introduction to Exception Handling

Mastering Python Exception Handling

When Things Don't Go as Planned

What is an Exception?

An exception is an event that occurs during the execution of a program that disrupts its normal flow.

🤖 Analogy: The Robot Chef

Imagine you're a robot following a recipe. The recipe says, "Take one egg from the carton." But when you open the carton, it's empty!

You can't proceed. Your normal flow is disrupted. You have encountered an "exception."

In Python, when an error occurs at runtime, it creates an exception object. If not handled, the program crashes and prints a "traceback."

```
# This code will crash if the user enters text instead of a number
age_str = input("Enter your age: ")
age_int = int(age_str)
# 💥 What if age_str is "twenty"? This will raise an exception! print(f"Next
year you will be {age_int + 1}.")
```

Broken Rules vs. Unexpected Events

It's crucial to understand the difference between a **Syntax Error** and an **Exception**.

Syntax Errors	Exceptions (Runtime Errors)
• These are parsing errors • Code violates Python's grammatical rules • Program won't even start running • Python spots these before execution • **Example:** `print("Hello"` (missing closing parenthesis)	• Code's syntax is perfectly valid • Error occurs while program is running • Happens when operation is impossible to perform • Can be handled gracefully • **Example:** `10 / 0` (mathematically impossible)

Remember: Python catches *Errors* before the show starts. *Exceptions* are unexpected drama that happens live on stage.

Meet the Usual Suspects

You'll encounter these frequently. Knowing their names helps you debug faster!

❌ ValueError

Raised when a function receives an argument of the correct type but an inappropriate value.

int("hello") # Can't convert "hello" to an integer.

🔧 TypeError

Raised when an operation is applied to an object of the wrong type.

"2" + 2 # Can't add a string and an integer.

📍 IndexError

Raised when you try to access an index from a sequence (like a list) that is out of range.

my_list = [1, 2, 3]
my_list[3] # The last index is 2.

🔑 KeyError

The dictionary version of IndexError. Raised when you try to access a key that doesn't exist.

```
my_dict = {"name": "Ria"}
my_dict["age"] # The "age" key does not exist.
```

➗ ZeroDivisionError

Raised when the second argument of a division or modulo operation is zero.

```
result = 100 / 0
```

The Safety Net

Instead of letting our program crash, we can handle exceptions gracefully.

•**The try block:** You put your "risky" code here—the code that might raise an exception.

•**The except block:** This code only runs if an exception occurs in the try block. It's your Plan B.

Syntax:

```
try:
    # Code that might raise an exception
    numerator = int(input("Enter a numerator: "))
    denominator = int(input("Enter a denominator: "))
    result = numerator / denominator
    print(f"The result is {result}")

except ValueError:
    # This block runs ONLY if the user enters a non-integer
    print("Invalid input! Please enter numbers only.")

except ZeroDivisionError:
    # This block runs ONLY if the user enters 0 for the denominator
    print("Error: You cannot divide by zero!")
```

Covering All the Bases

You can make your handling more robust with two more optional blocks.

✅ else block:

- This code runs only if the try block completes successfully (i.e., NO exceptions were raised)
- It's perfect for code that should only run if the risky part succeeded

🔄 finally block:

- This code runs no matter what. It will run whether an exception occurred or not
- It's essential for cleanup actions, like closing a file or a network connection

Covering All the Bases

```python
try:
    num = int(input("Enter a number: "))
except ValueError:
    print("That's not a valid number!")
else:
    # This only runs if the int() conversion was successful
    print(f"You entered the number {num}.")
finally:
    # This always runs, at the very end
    print("Execution complete.")
```

✍️ Hands-on Exercise & Solution

Exercise 11: The "Safe" Grade Calculator

Scenario: Create a function that calculates a student's score percentage. The function takes a list of student data and an index as input. The data can be messy, and the input might be invalid.

Task: Write a function `calculate_score(data, index)` that is "crash-proof."

Handle these exceptions:

- **IndexError:** Index out of bounds
- **KeyError:** Missing keys in record
- **ZeroDivisionError:** Total marks is 0
- **TypeError:** Non-numeric marks

✍️ Solution

Output

```
Test 1: Student Arun scored 87.00%.
Test 2: Error: Total marks for Bina is zero.
Cannot calculate percentage.
Test 3: Error: Invalid data type for marks for
Chloe. Please use numbers.
Test 4: Error: Missing key in student record:
'total_marks’.
Test 5: Error: Invalid index 4. No student
record found.
```

🎯 Key Takeaways

- **Prevents Crashes:** Exception handling builds robust applications that don't crash
- **Separates Logic:** Clean separation between normal logic and error handling
- **Be Specific:** Catch specific exceptions rather than generic ones
- **finally Guarantees Cleanup:** Critical cleanup code always runs