Programming 2: OOP Revision (Lecture 2)
Inheritance & Polymorphism

Comp 111

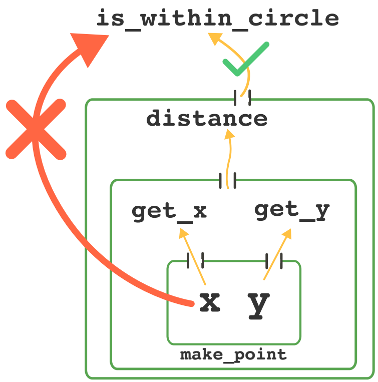
Forman Christian University

Today's Agenda

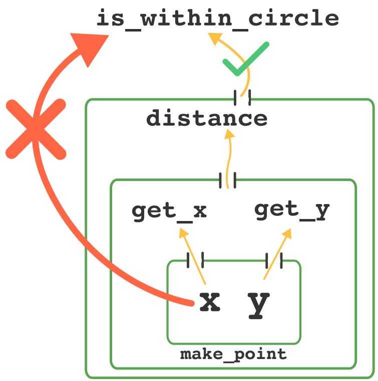
1. Quick Recap: Classes & Objects
2. Inheritance
3. Polymorphism
4. Abstract Base Classes
5. Summary: OOP Advantages

Quick Recap

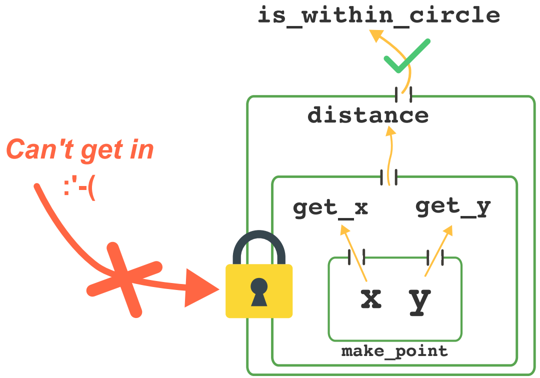
Abstraction



Abstraction

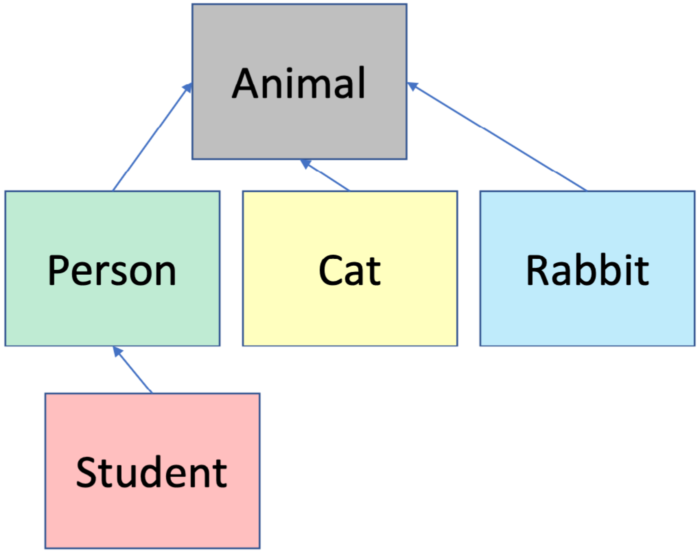


Encapsulation



Inheritance

- **Parent class** (superclass)
- **Child class** (subclass)
  - **Inherits** all data and behaviors of parent class
  - **Add** more **info**
  - **Add** more **behavior**
  - **Override** behavior



Inheritance: Parent Class

```python
class Animal(object):
    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_age(self, newage):
        self.age = newage
    def set_name(self, newname=""):
        self.name = newname
    def __str__(self):
        return f'animal:{self.name}-{self.age}'
```

Inheritance: Parent Class

```
class Animal(object):
    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_age(self, newage):
        self.age = newage
    def set_name(self, newname=""):
        self.name = newname
    def __str__(self):
        return f'animal:{self.name}-{self.age}'
```

object is the parent class of all classes in Python

Subclass Cat

```python
class Cat(Animal):
    def speak(self):
        print("meow")
    def __str__(self):
        return f'cat:{self.name}-{self.age}'

c = Cat(2)
c.set_name('simba')
print(c)
```

Subclass Cat

```python
class Cat(Animal):
    def speak(self):
        print("meow")
    def __str__(self):
        return f'cat:{self.name}-{self.age}'

c = Cat(2)
c.set_name('simba')
print(c)
```

Inherits all attributes and methods from the Animal class

Subclass Cat

```
class Cat(Animal):
    def speak(self):
        print("meow")
    def __str__(self):
        return f'cat:{self.name}-{self.age}'

c = Cat(2)
c.set_name('simba')
print(c)
```

Add new functionality. Not present in the parent class

Subclass Cat

```
class Cat(Animal):
    def speak(self):
        print("meow")
    def __str__(self):
        return f'cat:{self.name}-{self.age}'

c = Cat(2)
c.set_name('simba')
print(c)
```

"**Override**" `__str__`, replacing parent's method

Subclass Cat

```python
class Cat(Animal):
    def speak(self):
        print("meow")
    def __str__(self):
        return f'cat:{self.name}-{self.age}'

c = Cat(2)
c.set_name('simba')
print(c)
```

`__init__` is not missing, uses the `Animal` version

Big Idea

Override a method: Create a **new method** in the child class but with **same name** as in the parent class.

Can't use child class Methods

```
a = Animal(1)
c = Cat(2)    # Child CAN use parent's methods:
              #   (__init__)
c.speak()     # meow
a.speak()     # ERROR: parent can NOT use
              #   child's methods or attributes
```

Which Method to Use?

- Subclass can have **methods with same name** as superclass *(method* **override***)*

Which Method to Use?

- Subclass can have **methods with same name** as superclass *(method* **override***)*
- For an instance of a class, look for a method name in **current class definition**
- If not found, look for method name **up the hierarchy** *(in parent, then grandparent, and so on)*

Which Method to Use?

- Subclass can have **methods with same name** as superclass *(method* **override***)*
- For an instance of a class, look for a method name in **current class definition**
- If not found, look for method name **up the hierarchy** *(in parent, then grandparent, and so on)*
- Use first method up the hierarchy that you found with that method name

Subclass Person

```python
class Person(Animal):
    def __init__(self, name, age):
        Animal.__init__(self, age)
        self.set_name(name)
        self.friends = []
    def get_friends(self):
        return self.friends.copy()
    def add_friend(self, fname):
        if fname not in self.friends:
            self.friends.append(fname)
    def speak(self):
        print("hello")
    def age_diff(self, other):
        diff = self.age - other.age
        print(abs(diff), "year difference")
    def __str__(self):
        return f'person:{self.name}-{self.age}'
```

Subclass Person

```python
class Person(Animal):
    def __init__(self, name, age):
        Animal.__init__(self, age)
        self.set_name(name)
        self.friends = []
    def get_friends(self):
        return self.friends.copy()
    def add_friend(self, fname):
        if fname not in self.friends:
            self.friends.append(fname)
    def speak(self):
        print("hello")
    def age_diff(self, other):
        diff = self.age - other.age
        print(abs(diff), "year difference")
    def __str__(self):
        return f'person:{self.name}-{self.age}'
```

Parent is the Animal class

Subclass Person

```python
class Person(Animal):
    def __init__(self, name, age):
        Animal.__init__(self, age)
        self.set_name(name)
        self.friends = []
    def get_friends(self):
        return self.friends.copy()
    def add_friend(self, fname):
        if fname not in self.friends:
            self.friends.append(fname)
    def speak(self):
        print("hello")
    def age_diff(self, other):
        diff = self.age - other.age
        print(abs(diff), "year difference")
    def __str__(self):
        return f'person:{self.name}-{self.age}'
```

Note: Person class **overrides** `__init__` method

Subclass Person

```
class Person(Animal):
    def __init__(self, name, age):
        Animal.__init__(self, age)
        self.set_name(name)
        self.friends = []
    def get_friends(self):
        return self.friends.copy()
    def add_friend(self, fname):
        if fname not in self.friends:
            self.friends.append(fname)
    def speak(self):
        print("hello")
    def age_diff(self, other):
        diff = self.age - other.age
        print(abs(diff), "year difference")
    def __str__(self):
        return f'person:{self.name}-{self.age}'
```

Due to overriding, Animal's `__init__` method will **not** be called automatically

Subclass Person

```python
class Person(Animal):
    def __init__(self, name, age):
        Animal.__init__(self, age)
        self.set_name(name)
        self.friends = []
    def get_friends(self):
        return self.friends.copy()
    def add_friend(self, fname):
        if fname not in self.friends:
            self.friends.append(fname)
    def speak(self):
        print("hello")
    def age_diff(self, other):
        diff = self.age - other.age
        print(abs(diff), "year difference")
    def __str__(self):
        return f'person:{self.name}-{self.age}'
```

Person class has **additional** attributes

Subclass Person

```
class Person(Animal):
    def __init__(self, name, age):
        Animal.__init__(self, age)
        self.set_name(name)
        self.friends = []
    def get_friends(self):
        return self.friends.copy()
    def add_friend(self, fname):
        if fname not in self.friends:
            self.friends.append(fname)
    def speak(self):
        print("hello")
    def age_diff(self, other):
        diff = self.age - other.age
        print(abs(diff), "year difference")
    def __str__(self):
        return f'person:{self.name}-{self.age}'
```

Person class has **additional** methods

Subclass Person

```python
class Person(Animal):
    def __init__(self, name, age):
        Animal.__init__(self, age)
        self.set_name(name)
        self.friends = []
    def get_friends(self):
        return self.friends.copy()
    def add_friend(self, fname):
        if fname not in self.friends:
            self.friends.append(fname)
    def speak(self):
        print("hello")
    def age_diff(self, other):
        diff = self.age - other.age
        print(abs(diff), "year difference")
    def __str__(self):
        return f'person:{self.name}-{self.age}'
```

Person class **overrides** __str__ method

You Try!

Write a function according to the following specification:

```python
def make_pets(d):
    ''' Input: d is a dict mapping a Person obj to a Cat obj
    Prints: on each line, the name of a person, a colon, and the
    name of that person's cat
    Output: None '''
    pass

p1 = Person("zaid", 54)
p2 = Person("ahmed", 38)
c1 = Cat(1)
c1.set_name("simba")
c2 = Cat(1)
c2.set_name("tom")
d = {p1:c1, p2:c2}
make_pets(d) # prints zaid:simba
             #        ahmed:tom
```

Big Idea

A subclass can **use** a parent's attributes, **override** a parent's attributes, or **define new** attributes.

Attributes are either data or methods.

```python
import random
class Student(Person):
    def __init__(self, name, age, major=None):
        Person.__init__(self, name, age)
        self.major = major
    def change_major(self, major):
        self.major = major
    def speak(self):
        r = random.random()
        if r < 0.25:
            print("i have homework")
        elif 0.25 <= r < 0.5:
            print("i need sleep")
        elif 0.5 <= r < 0.75:
            print("i should eat")
        else:
            print("i'm still zooming")
    def __str__(self):
        return f'person:{self.name}-{self.age}-{self.major}'
```

```python
import random
class Student(Person):
    def __init__(self, name, age, major=None):
        Person.__init__(self, name, age)
        self.major = major
    def change_major(self, major):
        self.major = major
    def speak(self):
        r = random.random()
        if r < 0.25:
            print("i have homework")
        elif 0.25 <= r < 0.5:
            print("i need sleep")
        elif 0.5 <= r < 0.75:
            print("i should eat")
        else:
            print("i’m still zooming")
    def __str__(self):
        return f’person:{self.name}-{self.age}-{self.major}’
```

Student inherits both **Person** and **Animal** attributes

```python
import random
class Student(Person):
    def __init__(self, name, age, major=None):
        Person.__init__(self, name, age)
        self.major = major
    def change_major(self, major):
        self.major = major
    def speak(self):
        r = random.random()
        if r < 0.25:
            print("i have homework")
        elif 0.25 <= r < 0.5:
            print("i need sleep")
        elif 0.5 <= r < 0.75:
            print("i should eat")
        else:
            print("i'm still zooming")
    def __str__(self):
        return f'person:{self.name}-{self.age}-{self.major}'
```

Person's `__init__` creates it's own attributes as well as **Animal's** attributes

```python
import random
class Student(Person):
    def __init__(self, name, age, major=None):
        Person.__init__(self, name, age)
        self.major = major
    def change_major(self, major):
        self.major = major
    def speak(self):
        r = random.random()
        if r < 0.25:
            print("i have homework")
        elif 0.25 <= r < 0.5:
            print("i need sleep")
        elif 0.5 <= r < 0.75:
            print("i should eat")
        else:
            print("i'm still zooming")
    def __str__(self):
        return f'person:{self.name}-{self.age}-{self.major}'
```

Student class creates **additional** attributes

```python
import random
class Student(Person):
    def __init__(self, name, age, major=None):
        Person.__init__(self, name, age)
        self.major = major
    def change_major(self, major):
        self.major = major
    def speak(self):
        r = random.random()
        if r < 0.25:
            print("i have homework")
        elif 0.25 <= r < 0.5:
            print("i need sleep")
        elif 0.5 <= r < 0.75:
            print("i should eat")
        else:
            print("i’m still zooming")
    def __str__(self):
        return f'person:{self.name}-{self.age}-{self.major}'
```

Student speaks differently than **Person** *(behavior override)*

Class Variables and the Rabbit Subclass

- **Class variables** and their values are shared between all instances of a class

```python
class Rabbit(Animal):
    tag = 1
    def __init__(self,age,parent1=None,parent2=None):
        Animal.__init__(self, age)
        self.parent1 = parent1
        self.parent2 = parent2
        self.rid = Rabbit.tag
        Rabbit.tag += 1
```

Class Variables and the Rabbit Subclass

- **Class variables** and their values are shared between all instances of a class

parent class

```python
class Rabbit(Animal):
    tag = 1
    def __init__(self,age,parent1=None,parent2=None):
        Animal.__init__(self, age)
        self.parent1 = parent1
        self.parent2 = parent2
        self.rid = Rabbit.tag
        Rabbit.tag += 1
```

Class Variables and the Rabbit Subclass

- **Class variables** and their values are shared between all instances of a class

shared **class variable**

```python
class Rabbit(Animal):
    tag = 1
    def __init__(self,age,parent1=None,parent2=None):
        Animal.__init__(self, age)
        self.parent1 = parent1
        self.parent2 = parent2
        self.rid = Rabbit.tag
        Rabbit.tag += 1
```

Class Variables and the Rabbit Subclass

- Class variables and their values are shared between all instances of a class

```python
class Rabbit(Animal):
    tag = 1
    def __init__(self,age,parent1=None,parent2=None):
        Animal.__init__(self, age)
        self.parent1 = parent1
        self.parent2 = parent2
        self.rid = Rabbit.tag
        Rabbit.tag += 1
```

instance variable

Class Variables and the Rabbit Subclass

- **Class variables** and their values are shared between all instances of a class

```python
class Rabbit(Animal):
    tag = 1
    def __init__(self,age,parent1=None,parent2=None):
        Animal.__init__(self, age)
        self.parent1 = parent1
        self.parent2 = parent2
        self.rid = Rabbit.tag
        Rabbit.tag += 1
```

instance variable

read shared **class variable**

Class Variables and the Rabbit Subclass

- **Class variables** and their values are shared between all instances of a class

```python
class Rabbit(Animal):
    tag = 1
    def __init__(self,age,parent1=None,parent2=None):
        Animal.__init__(self, age)
        self.parent1 = parent1
        self.parent2 = parent2
        self.rid = Rabbit.tag
        Rabbit.tag += 1
```

Modifying class variable changes it for **all** instances that may reference it

Class Variables and the Rabbit Subclass

- **Class variables** and their values are shared between all instances of a class

```python
class Rabbit(Animal):
    tag = 1
    def __init__(self,age,parent1=None,parent2=None):
        Animal.__init__(self, age)
        self.parent1 = parent1
        self.parent2 = parent2
        self.rid = Rabbit.tag
        Rabbit.tag += 1
```

- `tag` used to give **unique id** to each new rabbit instance

```
def __init__(self,age,parent1=None,
                    parent2=None):
    Animal.__init__(self, age)
    self.parent1 = parent1
    self.parent2 = parent2
    self.rid = Rabbit.tag
    Rabbit.tag += 1
```

Rabbit.tag $\boxed{1}$

```python
def __init__(self,age,parent1=None,
                       parent2=None):
    Animal.__init__(self, age)
    self.parent1 = parent1
    self.parent2 = parent2
    self.rid = Rabbit.tag
    Rabbit.tag += 1
```

Rabbit.tag $\boxed{2}$

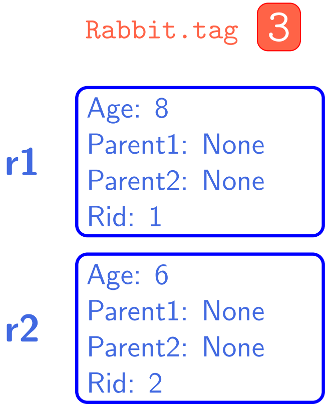
r1

Age: 8
Parent1: None
Parent2: None
Rid: 1

```python
r1 = Rabbit(8)
```

```
def __init__(self,age,parent1=None,
                        parent2=None):
    Animal.__init__(self, age)
    self.parent1 = parent1
    self.parent2 = parent2
    self.rid = Rabbit.tag
    Rabbit.tag += 1
```

```
r1 = Rabbit(8)
r2 = Rabbit(6)
```



Rabbit.tag 3

r1
Age: 8
Parent1: None
Parent2: None
Rid: 1

r2
Age: 6
Parent1: None
Parent2: None
Rid: 2

```
def  __init__(self,age,parent1=None,
                    parent2=None):
    Animal.__init__(self, age)
    self.parent1 = parent1
    self.parent2 = parent2
    self.rid = Rabbit.tag
    Rabbit.tag += 1
```

Rabbit.tag 4

```
r1 = Rabbit(8)
r2 = Rabbit(6)
r3 = Rabbit(10)
```

r1

Age: 8
Parent1: None
Parent2: None
Rid: 1

r2

Age: 6
Parent1: None
Parent2: None
Rid: 2

r3

Age: 10
Parent1: None
Parent2: None
Rid: 3

Rabbit Getter Methods

```python
class Rabbit(Animal):
    tag = 1
    def __init__(self, age, parent1=None, parent2=None):
        Animal.__init__(self, age)
        self.parent1 = parent1
        self.parent2 = parent2
        self.rid = Rabbit.tag
        Rabbit.tag += 1
    #-----------------------------------#
    def get_rid(self):                  #
        return str(self.rid).zfill(5)   #  Getter Methods
    def get_parent1(self):              #    specific to the
        return self.parent1             #    Rabbit class
    def get_parent2(self):              #
        return self.parent2             #
    #-----------------------------------#
```

Working with Your Own Types

```
def __add__(self, other):
    # returning object of same type as this class
    return Rabbit(0, self, other)
```

- Define **+ operator** between two Rabbit instances
  - For example:

    `r4 = r1 + r2`

    `r1` and `r2` are `Rabbit` instances, combine to create `r4`

Working with Your Own Types

```python
def __add__(self, other):
    # returning object of same type as this class
    return Rabbit(0, self, other)
```

- Define **+ operator** between two Rabbit instances
  - For example:

    `r4 = r1 + r2`

    `r1` and `r2` are `Rabbit` instances, combine to create `r4`
- `r4` is a new `Rabbit` instance with age 0

Working with Your Own Types

```
def __add__(self, other):
    # returning object of same type as this class
    return Rabbit(0, self, other)
```

- Define **+ operator** between two Rabbit instances
  - For example:

    `r4 = r1 + r2`

    `r1` and `r2` are Rabbit instances, combine to create `r4`
- `r4` is a new Rabbit instance with age 0
- `r4` has `self` as one parent and `other` as the other parent

Working with Your Own Types

```
def __add__(self, other):
    # returning object of same type as this class
    return Rabbit(0, self, other)
```

- Define + **operator** between two Rabbit instances
  - For example:

    `r4 = r1 + r2`

    `r1` and `r2` are `Rabbit` instances, combine to create `r4`
- `r4` is a new `Rabbit` instance with age `0`
- `r4` has `self` as one parent and `other` as the other parent
- In `__init__`, `parent1` and `parent2` are of type `Rabbit`

Special Method to Compare Two Rabbits

- Decide that two rabbits are equal if they have the **same two parents**

```
def __eq__(self, other):
    parents_same = (self.p1.rid == other.p1.rid and
                    self.p2.rid == other.p2.rid)
    parents_opp  = (self.p2.rid == other.p1.rid and
                   self.p1.rid == other.p2.rid)
    return parents_same or parents_opp
```

Special Method to Compare Two Rabbits

- Decide that two rabbits are equal if they have the **same two parents**

```
def __eq__(self, other):
    parents_same = (self.p1.rid == other.p1.rid and
                    self.p2.rid == other.p2.rid)
    parents_opp = (self.p2.rid == other.p1.rid and
                   self.p1.rid == other.p2.rid)
    return parents_same or parents_opp
```

Booleans checking
r1 + r2 or
r2 + r1

Special Method to Compare Two Rabbits

- Decide that two rabbits are equal if they have the **same two parents**

```python
def __eq__(self, other):
    parents_same = (self.p1.rid == other.p1.rid and
                    self.p2.rid == other.p2.rid)
    parents_opp  = (self.p2.rid == other.p1.rid and
                   self.p1.rid == other.p2.rid)
    return parents_same or parents_opp
```

- Compare ids of parents since **ids are unique** *(due to class var)*

Special Method to Compare Two Rabbits

- Decide that two rabbits are equal if they have the **same two parents**

```
def __eq__(self, other):
    parents_same = (self.p1.rid == other.p1.rid and
                    self.p2.rid == other.p2.rid)
    parents_opp  = (self.p2.rid == other.p1.rid and
                   self.p1.rid == other.p2.rid)
    return parents_same or parents_opp
```

- Compare ids of parents since **ids are unique** *(due to class var)*
- **Note:** you **CAN'T** compare objects directly *(recursive if __eq__)* Also, can't call on `None` *(`AttributeError` when `None.parent1`)*

Big Idea

Class Variables are **shared** between all instances

If one instance changes it, it's changed for every instance.

Polymorphism

Quick Recap: Class Hierarchy

```python
class Animal(object):
    def __init__(self, age):
        self.age = age
    def speak(self):
        print("some sound")

class Cat(Animal):
    def meo(self):
        print("meow")

class Dog(Animal):
    def bark(self):
        print("woof")
```

A Common Programming Problem

Suppose we want to make all our animals speak:

```
animals = [Cat(2), Dog(3), Rabbit(1), Cat(5)]

# Without polymorphism, we'd need:
for animal in animals:
    if isinstance(animal, Cat):
        animal.meo()
    elif isinstance(animal, Dog):
        animal.bark()
    elif isinstance(animal, Rabbit):
        animal.squeak()
# Need to add elif for EVERY new animal type!
```

A Common Programming Problem

Suppose we want to make all our animals speak:

```
animals = [Cat(2), Dog(3), Rabbit(1), Cat(5)]

# Without polymorphism, we'd need:
for animal in animals:
    if isinstance(animal, Cat):
        animal.meo()
    elif isinstance(animal, Dog):
        animal.bark()
    elif isinstance(animal, Rabbit):
        animal.squeak()
# Need to add elif for EVERY new animal type!
```

Must check type of each animal manually - tedious!

A Common Programming Problem

Suppose we want to make all our animals speak:

```python
animals = [Cat(2), Dog(3), Rabbit(1), Cat(5)]

# Without polymorphism, we'd need:
for animal in animals:
    if isinstance(animal, Cat):
        animal.meo()
    elif isinstance(animal, Dog):
        animal.bark()
    elif isinstance(animal, Rabbit):
        animal.squeak()
# Need to add elif for EVERY new animal type!
```

Must check type of each animal manually - tedious!

This is tedious and doesn't scale!

The Solution

Polymorphism

"Many forms"

The ability to use objects of different types through a uniform interface

Polymorphism

- Greek: **"poly"** = many, **"morph"** = form

Polymorphism

- Greek: **"poly"** = many, **"morph"** = form
- The ability to **treat objects of different types** in a **similar way**

Polymorphism

- Greek: **"poly"** = many, **"morph"** = form
- The ability to **treat objects of different types** in a **similar way**
- Same **method name**, different **implementations**

Polymorphism

- Greek: **"poly"** = many, **"morph"** = form
- The ability to **treat objects of different types** in a **similar way**
- Same **method name**, different **implementations**
- Python automatically calls the **correct version** of the method based on the object's type

Polymorphism in Action

```python
class Animal:
    def speak(self):
        print("some sound")

class Cat(Animal):
    def speak(self):
        print("meow")

class Dog(Animal):
    def speak(self):
        print("woof")

animals = [Cat(2), Dog(3), Cat(1)]
for animal in animals:
    # Each calls their own version!
    animal.speak()
```

Polymorphism in Action

```python
class Animal:
    def speak(self):
        print("some sound")

class Cat(Animal):
    def speak(self):
        print("meow")

class Dog(Animal):
    def speak(self):
        print("woof")

animals = [Cat(2), Dog(3), Cat(1)]
for animal in animals:
    # Each calls their own version!
    animal.speak()
```

Same method name in all subclasses

Polymorphism in Action

```python
class Animal:
    def speak(self):
        print("some sound")

class Cat(Animal):
    def speak(self):
        print("meow")

class Dog(Animal):
    def speak(self):
        print("woof")

animals = [Cat(2), Dog(3), Cat(1)]
for animal in animals:
    # Each calls their own version!
    animal.speak()
```

Python automatically calls the correct `speak()` method!

Output

```
meow
woof
meow
```

Output

```
meow
woof
meow
```

- Python **automatically** determines which `speak()` to call
- Based on the **actual type** of the object
- We don't need to check types manually!

Benefits of Polymorphism

1. **Flexibility**
   - Write code that works with parent class but accepts any subclass

Benefits of Polymorphism

1. **Flexibility**
   - Write code that works with parent class but accepts any subclass
2. **Extensibility**
   - Add new subclasses without changing existing code

Benefits of Polymorphism

1. **Flexibility**
    - Write code that works with parent class but accepts any subclass
2. **Extensibility**
    - Add new subclasses without changing existing code
3. **Code Reusability**
    - One function works with many types

Benefits of Polymorphism

1. **Flexibility**
   - Write code that works with parent class but accepts any subclass
2. **Extensibility**
   - Add new subclasses without changing existing code
3. **Code Reusability**
   - One function works with many types
4. **Cleaner Code**
   - No need for long if-elif chains

Example 1: Animal Shelter

```python
def make_sound(animal):
    """Works with ANY Animal subclass"""
    animal.speak()

cat = Cat(2)
dog = Dog(3)
rabbit = Rabbit(1)

make_sound(cat)      # meow
make_sound(dog)      # woof
make_sound(rabbit)   # squeak
```

Example 1: Animal Shelter

```python
def make_sound(animal):
    """Works with ANY Animal subclass"""
    animal.speak()

cat = Cat(2)
dog = Dog(3)
rabbit = Rabbit(1)

make_sound(cat)      # meow
make_sound(dog)      # woof
make_sound(rabbit)   # squeak
```

Function accepts **Animal** but works with **any subclass**

Example 1: Animal Shelter

```python
def make_sound(animal):
    """Works with ANY Animal subclass"""
    animal.speak()

cat = Cat(2)
dog = Dog(3)
rabbit = Rabbit(1)

make_sound(cat)      # meow
make_sound(dog)      # woof
make_sound(rabbit)   # squeak
```

Same function, different behaviors!

Example 2: Processing Collections

```python
def morning_routine(animals):
    """Make all animals speak in the morning"""
    for animal in animals:
        animal.speak()

# Mix of different animal types
zoo = [Cat(2), Dog(3), Rabbit(1),
       Cat(1), Dog(5)]

morning_routine(zoo)
# Output: meow, woof, squeak, meow, woof
```

Example 3: More Complex Behavior

```python
class Animal:
    def __init__(self, age, name):
        self.age = age
        self.name = name
    def introduce(self):
        print(f"I'm {self.name}, I'm {self.age} years old")
        self.speak()

class Cat(Animal):
    def speak(self):
        print("meow")

class Dog(Animal):
    def speak(self):
        print("woof")

c = Cat(2, "Fluffy")
c.introduce()  # I'm Fluffy, I'm 2 years old
               # meow
```

Example 3: More Complex Behavior

```python
class Animal:
    def __init__(self, age, name):
        self.age = age
        self.name = name
    def introduce(self):
        print(f"I'm {self.name}, I'm {self.age} years old")
        self.speak()

class Cat(Animal):
    def speak(self):
        print("meow")

class Dog(Animal):
    def speak(self):
        print("woof")

c = Cat(2, "Fluffy")
c.introduce()  # I'm Fluffy, I'm 2 years old
               # meow
```

Parent method calls polymorphic method

You Try! Exercise 1

Create a Shape hierarchy with polymorphic area() method:

```python
class Shape:
    def area(self):
        pass  # To be overridden

class Rectangle(Shape):
    def __init__(self, width, height):
        # Your code here
    def area(self):
        # Your code here

class Circle(Shape):
    def __init__(self, radius):
        # Your code here
    def area(self):
        # Your code here (use 3.14 for pi)
```

You Try! Exercise 1 (continued)

Write a function that uses polymorphism:

```python
def total_area(shapes):
    """
    Input: shapes is a list of Shape objects
    Returns: total area of all shapes
    """
    # Your code here
    pass

# Test your code:
shapes = [Rectangle(4, 5),
          Circle(3),
          Rectangle(2, 3)]
print(total_area(shapes))  # Should print: 54.26
```

You Try! Exercise 2

Create an `Employee` hierarchy:

```
class Employee:
    def __init__(self, name, base_salary):
        self.name = name
        self.base_salary = base_salary
    def calculate_pay(self):
        return self.base_salary

class Manager(Employee):
    def __init__(self, name, base_salary, bonus):
        # Your code: call parent __init__ and store bonus
    def calculate_pay(self):
        # Your code: return base_salary + bonus

class Salesperson(Employee):
    def __init__(self, name, base_salary, commission):
        # Your code: call parent __init__ and store commission
    def calculate_pay(self):
        # Your code: return base_salary + commission
```

You Try! Exercise 2 (continued)

```python
def print_payroll(employees):
    """
    Input: employees is a list of Employee objects
    Prints: name and pay for each employee
    Returns: total payroll
    """
    # Your code here
    pass

# Test:
employees = [ Employee("Alice", 50000),  Manager("Bob", 60000, 10000),
    Salesperson("Charlie", 40000, 15000) ]

total = print_payroll(employees)
# Should print:
# Alice: \$50000
# Bob: \$70000
# Charlie: \$55000
# Total: \$175000
```

Abstract Base Classes

The Problem with Our Animal Class

- What if someone creates an `Animal` directly?

The Problem with Our Animal Class

- What if someone creates an `Animal` directly?
- `a = Animal(5, "Generic")`

The Problem with Our Animal Class

- What if someone creates an `Animal` directly?
- `a = Animal(5, "Generic")`
- What sound does a generic "animal" make?

The Problem with Our Animal Class

- What if someone creates an `Animal` directly?
- `a = Animal(5, "Generic")`
- What sound does a generic ”animal” make?
- We want `Animal` to be a **template** only

The Problem with Our Animal Class

- What if someone creates an `Animal` directly?
- `a = Animal(5, "Generic")`
- What sound does a generic "animal" make?
- We want `Animal` to be a **template** only
- Force subclasses to **implement** `speak()`

Solution

Abstract Base Class (ABC)

A class that:

- Cannot be instantiated directly
- Forces subclasses to implement certain methods
- Defines a **contract** for subclasses

Creating an Abstract Base Class

```python
from abc import ABC, abstractmethod

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

    @abstractmethod
    def speak(self):
        pass

    def introduce(self):
        print(f"I'm {self.name}, I'm {self.age} years old")
        self.speak()
```

Creating an Abstract Base Class

```python
from abc import ABC, abstractmethod

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

    @abstractmethod
    def speak(self):
        pass

    def introduce(self):
        print(f"I'm {self.name}, I'm {self.age} years old")
        self.speak()
```

Import ABC tools from Python

Creating an Abstract Base Class

```python
from abc import ABC, abstractmethod

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

    @abstractmethod
    def speak(self):
        pass

    def introduce(self):
        print(f"I'm {self.name}, I'm {self.age} years old")
        self.speak()
```

Inherit from ABC

Creating an Abstract Base Class

```python
from abc import ABC, abstractmethod

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

    @abstractmethod
    def speak(self):
        pass

    def introduce(self):
        print(f"I'm {self.name}, I'm {self.age} years old")
        self.speak()
```

Mark method as abstract - subclasses **must** implement it

Cannot Instantiate Abstract Classes

```python
from abc import ABC, abstractmethod

class Animal(ABC):
    @abstractmethod
    def speak(self):
        pass

# This will cause an ERROR:
a = Animal(5, "Generic")

# TypeError: Can't instantiate abstract
# class Animal with abstract method speak
```

Cannot Instantiate Abstract Classes

```python
from abc import ABC, abstractmethod

class Animal(ABC):
    @abstractmethod
    def speak(self):
        pass

# This will cause an ERROR:
a = Animal(5, "Generic")

# TypeError: Can't instantiate abstract
# class Animal with abstract method speak
```

Python prevents creating Animal objects!

Subclasses Must Implement Abstract Methods

```python
class Cat(Animal):
    def speak(self):
        print("meow")

class Dog(Animal):
    def speak(self):
        print("woof")

# Now these work fine:
c = Cat(2, "Fluffy")
d = Dog(3, "Buddy")
c.speak()  # meow
d.speak()  # woof
```

Subclasses Must Implement Abstract Methods

```python
class Cat(Animal):
    def speak(self):
        print("meow")

class Dog(Animal):
    def speak(self):
        print("woof")

# Now these work fine:
c = Cat(2, "Fluffy")
d = Dog(3, "Buddy")
c.speak()  # meow
d.speak()  # woof
```

Subclass implements the abstract method

Forgetting to Implement Causes Error

```python
class Rabbit(Animal):
    def hop(self):
        print("hopping")
    # Forgot to implement speak()!

# This will cause an ERROR:
r = Rabbit(1, "Fluffy")

# TypeError: Can't instantiate abstract
# class Rabbit with abstract method speak
```

Forgetting to Implement Causes Error

```python
class Rabbit(Animal):
    def hop(self):
        print("hopping")
    # Forgot to implement speak()!

# This will cause an ERROR:
r = Rabbit(1, "Fluffy")

# TypeError: Can't instantiate abstract
# class Rabbit with abstract method speak
```

Missing required speak() method

Forgetting to Implement Causes Error

```python
class Rabbit(Animal):
    def hop(self):
        print("hopping")
    # Forgot to implement speak()!

# This will cause an ERROR:
r = Rabbit(1, "Fluffy")

# TypeError: Can't instantiate abstract
# class Rabbit with abstract method speak
```

Python catches the error immediately!

Why Use Abstract Classes?

1. Enforce consistency - all subclasses have required methods
2. Catch errors early - at instantiation, not when method is called
3. Document intent - clearly shows which methods subclasses need
4. Prevent misuse - can't create incomplete objects

Common Pitfalls

Pitfall 1: Forgetting to Override

```python
class Animal:
    def speak(self):
        print("some sound")

class Cat(Animal):
    def meow(self):  # Wrong method name!
        print("meow")

c = Cat(2)
c.speak()  # Prints "some sound" (not "meow")
```

Pitfall 1: Forgetting to Override

```python
class Animal:
    def speak(self):
        print("some sound")

class Cat(Animal):
    def meow(self):  # Wrong method name!
        print("meow")

c = Cat(2)
c.speak()  # Prints "some sound" (not "meow")
```

Must use **same name** as parent method!

Pitfall 2: Wrong Method Signature

```python
class Animal:
    def speak(self):
        print("some sound")

class Cat(Animal):
    def speak(self, volume):  # Extra parameter!
        print(f"meow at volume {volume}")

def make_sound(animal):
    animal.speak()  # Error! Missing argument

c = Cat(2)
make_sound(c)  # TypeError!
```

Pitfall 2: Wrong Method Signature

```
class Animal:
    def speak(self):
        print("some sound")

class Cat(Animal):
    def speak(self, volume):  # Extra parameter!
        print(f"meow at volume {volume}")

def make_sound(animal):
    animal.speak()  # Error! Missing argument

c = Cat(2)
make_sound(c)  # TypeError!
```

Signatures must match!

Key Principles

1. Use the **same method name** in parent and child
2. Keep the **same parameters** (method signature)
3. Write functions that accept **parent type** but work with **any subclass**
4. Python handles the rest **automatically**!

Duck Typing in Python

Duck Typing

- Python's approach to polymorphism is **"duck typing"**

Duck Typing

- Python's approach to polymorphism is **"duck typing"**
- *"If it walks like a duck and quacks like a duck, then it must be a duck"*

Duck Typing

- Python's approach to polymorphism is **"duck typing"**
- *"If it walks like a duck and quacks like a duck, then it must be a duck"*
- Python doesn't care about the **type** of an object

Duck Typing

- Python's approach to polymorphism is **"duck typing"**
- *"If it walks like a duck and quacks like a duck, then it must be a duck"*
- Python doesn't care about the **type** of an object
- Python only cares if the object has the **right methods**

Duck Typing

- Python's approach to polymorphism is **"duck typing"**
- *"If it walks like a duck and quacks like a duck, then it must be a duck"*
- Python doesn't care about the **type** of an object
- Python only cares if the object has the **right methods**
- Objects don't even need to inherit from the same parent!

Duck Typing Example

```python
class Dog:
    def speak(self):
        print("woof")

class Robot:  # Not related to Animal!
    def speak(self):
        print("beep boop")

class Person:
    def speak(self):
        print("hello")

def make_speak(thing):
    thing.speak()  # Works with anything that has speak()

make_speak(Dog())      # woof
make_speak(Robot())    # beep boop
make_speak(Person())   # hello
```

Duck Typing Example

```python
class Dog:
    def speak(self):
        print("woof")

class Robot:  # Not related to Animal!
    def speak(self):
        print("beep boop")

class Person:
    def speak(self):
        print("hello")

def make_speak(thing):
    thing.speak()  # Works with anything that has speak()

make_speak(Dog())      # woof
make_speak(Robot())    # beep boop
make_speak(Person())   # hello
```

No inheritance relationship needed!

Duck Typing Example

```python
class Dog:
    def speak(self):
        print("woof")

class Robot:  # Not related to Animal!
    def speak(self):
        print("beep boop")

class Person:
    def speak(self):
        print("hello")

def make_speak(thing):
    thing.speak()  # Works with anything that has speak()

make_speak(Dog())      # woof
make_speak(Robot())    # beep boop
make_speak(Person())   # hello
```

Works with **any** object with a `speak()` method

Summary

Summary: Key Takeaways

1. **Polymorphism** = "many forms"
2. Write code that works with **parent class**, automatically works with **all subclasses**
3. Same **method name**, different **implementations**
4. Python uses **duck typing** - only cares about methods, not types
5. Makes code more **flexible**, **extensible**, and **reusable**

Remember

Polymorphism allows you to write functions that work with **many different types** of objects through a **common interface**

This is one of the most powerful features of OOP!

Why OOP over Procedural?

- **Modularity** – Code organized into self-contained objects
- **Reusability** – Inheritance allows code reuse
- **Maintainability** – Changes isolated to specific classes
- **Flexibility** – Polymorphism enables extensible design

Questions?