Fundamentals of Object-Oriented Programming

INTERMEDIATE OBJECT-ORIENTED PROGRAMMING IN PYTHON

Jake Roach
Data Engineer

datacamp

Defining a class

```
class Person:
  def __init__(self, name, age):
    self.name = name
    self.age = age
    self.height = 0

# This invokes a call to __init__
john = Person("John Casey", 38)
```

- Define a class with the `class` keyword
- `__init__` is known as a constructor, and is called when a new class object is created
- `self` refers to the current instance of a class

Instance attributes

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

```
# Create an instance of Person
sarah = Person("Sarah Walker", 31)
sarah.age  # Retrieve the age instance attribute
```

- Associated with an object of a class
- Can be set and retrieved with the syntax `self.<attribute-name>`
- Accessed via `<object-name>.<attribute-name>`

Class attributes

- Tied to the class itself
- Can be retrieved **without** an object of the class
- Store data that should be the same for all class objects

```
class Person:
  residence = "Planet Earth"
  ...
```

```
# Accessed without an instance
print(Person.residence)
```

```
Planet Earth
```

Instance methods

```
class Person:
  ...
  def introduce(self):
      print(f"Hello, my name is {self.name}")
```

```
chuck = Person("Chuck", 32)
chuck.introduce()  # Called on a Person object
```

- Require an object of the class to exist to be called
- Take the `self` keyword as the first parameter

Class methods

```
class Person:
  @classmethod
  def wake_up():
      print("Time to start your day!")
```

```
# Calling a class method
Person.wake_up()
```

- Decorated with `@classmethod`
- Do not require a class object to be called

Inheritance

Inheritance allows for code to be reused between classes

- The child class (`Employee`) inherits all the functionality of the parent class (`Person`)
- "is-a" relationship
- Can implement additional functionality
  - Attributes
  - Methods

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

  def introduce(self):
      print(f"Hello, my name is {self.name}")
```

```
class Employee(Person):
  def __init__(self, name, age, title):
    Person.__init__(self, name, age)
    self.title = title

  def change_position(self, new_title):
      self.title = new_title
```

Inheritance

```
lester = Employee("Lester", 26, "Technician")
lester.introduce()  # Inherited from Person
print(lester.title)
```

```
Hello, my name is Lester
Technician
```

```
lester.change_position("Cashier")
print(lester.title)
```

```
Cashier
```

super()

```
class Employee(Person):
  def __init__(self, name, age, title):
    # Uses name of the parent class
    Person.__init__(self, name, age)

    self.title = title
  ...
```

- Class name, `__init__()`

Using `super()`

```
class Employee(Person):
  def __init__(self, name, age, title):
    # super(), instead of class name
    super().__init__(name, age)

    self.title = title
  ...
```

- `super()`, `__init__()`
- `self` does not need to be passed

Overriding

Child implements a method that was inherited from parent **in a new way**

```
class Employee(Person):
  ...
  def introduce(self):
      print(f"""My name is {self.name},
        I am a {self.title}""")
```

```
lester = Employee("Lester", 26, "Technician")
lester.introduce()
```

```
My name is Lester, I am a Technician
```

Overloading

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

  def __eq__(self, other):
      return self.name == other.name
```

- Customize the behavior of Python operators for a class
- `__eq__()` is used to overload `==`

```
# Compare two Person objects
chuck = Person("Charles Carmichael")
charles = Person("Charles Carmichael")
print(chuck == charles)
```

```
True
```

Let's practice!

INTERMEDIATE OBJECT-ORIENTED PROGRAMMING IN PYTHON

Overloading Python Operators

INTERMEDIATE OBJECT-ORIENTED PROGRAMMING IN PYTHON

Jake Roach
Data Engineer

datacamp

Overloading comparison operators

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

  def __eq__(self, other):
      return self.name == other.name
```

Customize the functionality of `==`

- Use the `__eq__()` magic method
- Takes `self` and `other`
- Returns a boolean value

```
bryce = Person("Bryce")
orion = Person("Orion")
print(bryce == orion)
```

```
False
```

```
chuck = Person("Charles Carmichael")
charles = Person("Charles Carmichael")
print(chuck == charles)
```

```
True
```

Adding two objects

```
class Team:
  def __init__(self, team_members):
      self.team_members = team_members

# Create two Team objects, attempt to add them
rookies = Team(["Casey", "Emmitt"])
veterans = Team(["Mike", "Chuck"])
dream_team = rookies + veterans
```

```
Traceback (most recent call last):
  File "<stdin>", line 8, in <module>
TypeError: unsupported operand type(s) for +: 'Team' and 'Team'
```

Overloading the + operator

```
class Team:
  def __init__(self, team_members):
    # team_members is a list of names
    self.team_members = team_members

  def __add__(self, other):
    # Adding Team objects creates a larger Team
    return Team(self.team_members + other.team_members)
```

- Use the `__add__()` magic method to overload `+`
- `self` and `other` are passed to `__add__()`
- Create a new `Team` object using the `team_members` from objects being "added"

Adding two objects

```
# Create two Team objects
rookies = Team(["Casey", "Emmitt"])
veterans = Team(["Mike", "Chuck"])

# Attempt to add these two Teams together
dream_team = rookies + veterans
print(type(dream_team))
print(dream_team.team_members)
```

```
Team
["Casey", "Emmitt", "Mike", "Chuck"]
```

Using + to create a new type of object

```
class Team:
  def __init__(self, team_members):
      self.team_members = team_members

class Employee:
  def __init__(self, name, title):
    self.name = name
    self.title = title

  def __add__(self, other):
    # Use the + operator to create a
    # Team with the name of each Employee
    return Team([self.name, other.name])
```

What if we want to create a `Team` by combining `Employee` objects?

- `__add__` is implemented in the `Employee` class
- Creates a new object of `Team` by creating a list of `Employee` names
- The result of adding two `Employee` objects is a single `Team`

Add two objects to create a new object

```
# Create two Employee objects
anna = Employee("Anna", "Technical Specialist")
jeff = Employee("Jeffrey", "Musician")

# Now, attempt to add these together to create a team
audio_team = anna + jeff
print(type(audio_team))
print(audio_team.team_members)
```

```
Team
["Anna", "Jeffrey"]
```

Overloading other operators

Operator	Magic Method	Operator Type
-	`__sub__`	Arithmetic
!=	`__ne__`	Comparison
<	`__lt__`	Comparison
>	`__gt__`	Comparison
+=	`__iadd__`	Assignment
and	`__and__`	Logical
in	`__contains__`	Membership
is	`__is__`	Identity

... And tons more!

Let's practice!

INTERMEDIATE OBJECT-ORIENTED PROGRAMMING IN PYTHON

Multiple Inheritance

INTERMEDIATE OBJECT-ORIENTED PROGRAMMING IN PYTHON

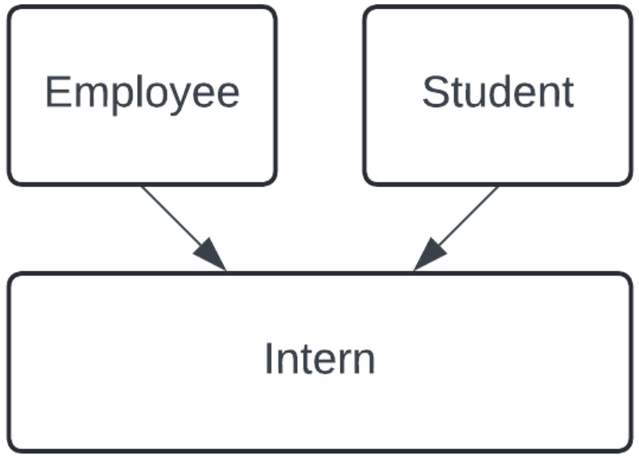
Jake Roach
Data Engineer

datacamp

Multiple inheritance

Allows for a class to inherit the functionality of more than a single class

- `Intern` can inherit from both `Employee` and `Student`
- Maintains an "is-a" relationship




```
class Employee:
  def __init__(self, department):
    self.department = department

  def begin_job(self):
    print(f"Welcome to {self.department}!")
```

```
class Student:
  def __init__(self, school):
    self.school = school
    self.courses = []

  def add_course(self, course_name):
    self.courses.append(course_name)
```

Multiple inheritance

```
class Intern(Employee, Student):
    def __init__(self, department, school, duration):
        # Make a call to BOTH constructors
        Employee.__init__(self, department)
        Student.__init__(self, school)
        self.duration = duration

    def onboard(self, mentor):
        # Implementation of a new method
        ...
```

Creating an Intern object

```
stephen = Intern("Software Development", "Echo University", 10)
stephen.begin_job()  # Method from Employee
```

```
Welcome to Software Development!
```

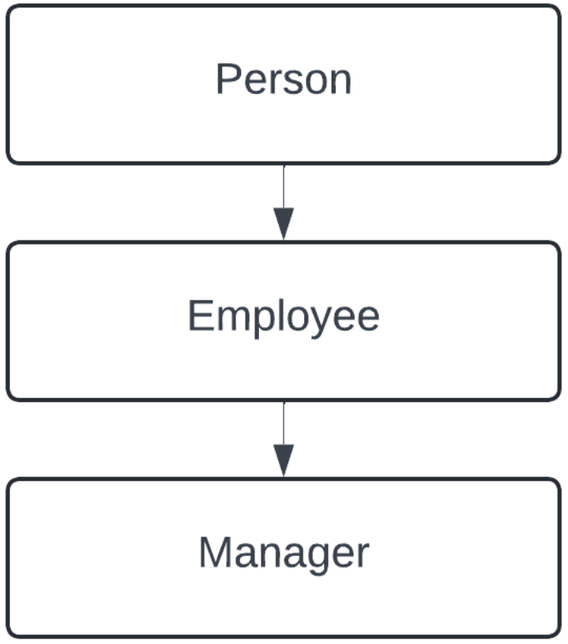
```
stephen.add_course("Intermediate OOP in Python")  # Method from Intern
print(stephen.courses)
```

```
["Intermediate OOP in Python"]
```

Multilevel inheritance

Inherit a class which inherits from another class, becoming a grandchild

- Maintain an "is-a" relationship




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

  def introduce(self):
      print(f"Hello, my name is {self.name}")
```

```
class Employee(Person):
  def __init__(self, name, title):
    Person.__init__(self, name)
    self.title = title

  def change_position(self, new_title):
    print(f"Starting new role as {new_title}")
    self.title = new_title
```

Multilevel inheritance

```
class Manager(Employee):
  def __init__(self, name, title, number_reports):
    Employee.__init__(self, name, title)
    self.number_reports = number_reports

mike = Manager("Mike", "Engineering Manager", 14)
mike.introduce()
mike.change_position("Director of Engineering")
print(mike.number_reports)
```

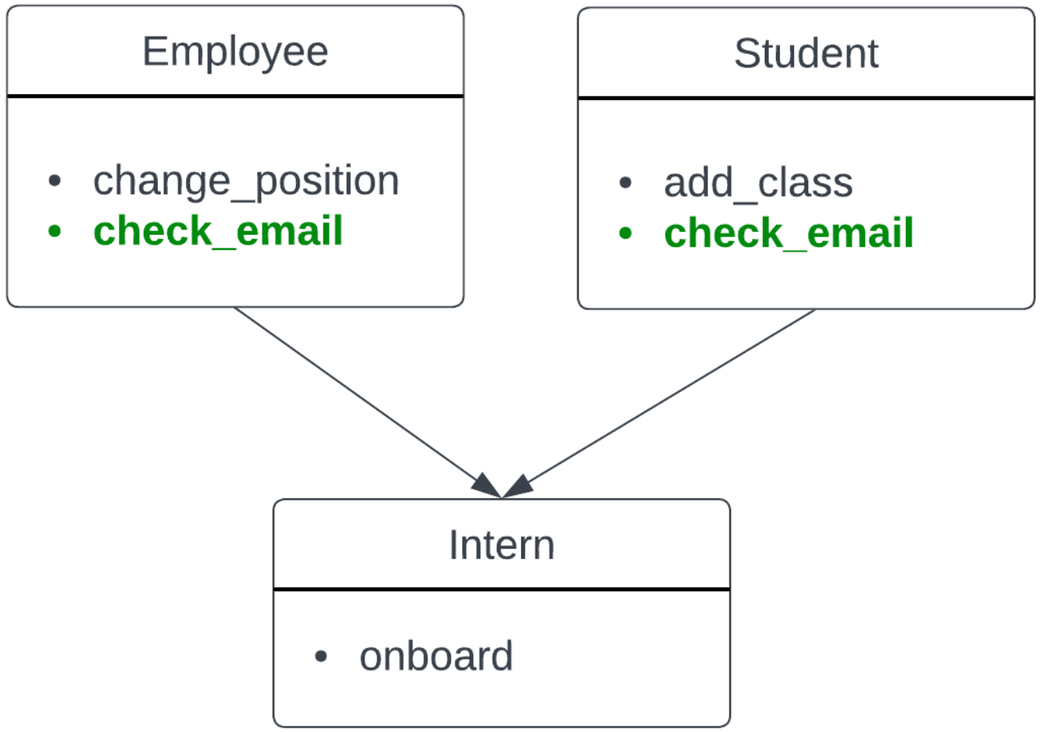
```
Hello, my name is Mike
Starting new role as Director of Engineering
14
```

Method resolution order (MRO)

The order in which Python determines which method is used when parent and children implement a method with the same name

Follow these rules to determine MRO:

- Children will be searched first
- Parent classes will be searched left-to-right as they were defined in class statement



`.mro()` method and `__mro__`

[1] Beyond the Basic Stuff with Python, Sweigart (pg. 311)

Method resolution order (MRO)

```
class Intern(Employee, Student):  # Intern inherits the Employee and the Student classes
  ...

# Find the MRO for Intern
print(Intern.mro())
```

```
[<class '__main__.Intern'>, <class '__main__.Employee'>, <class '__main__.Student'>, <class 'object'>]
```

```
# Find the MRO using __mro__
print(Intern.__mro__)
```

```
(<class '__main__.Intern'>, <class '__main__.Employee'>, <class '__main__.Student'>, <class 'object'>)
```

Let's practice!

INTERMEDIATE OBJECT-ORIENTED PROGRAMMING IN PYTHON