Class

Python Programming

Byeongjoon Noh

powernoh@sch.ac.kr

SOON CHUN HYANG UNIVERSITY

Most of the slides are available on Senseable AI Lab homepage: https://sailab.space/courses/

Contents

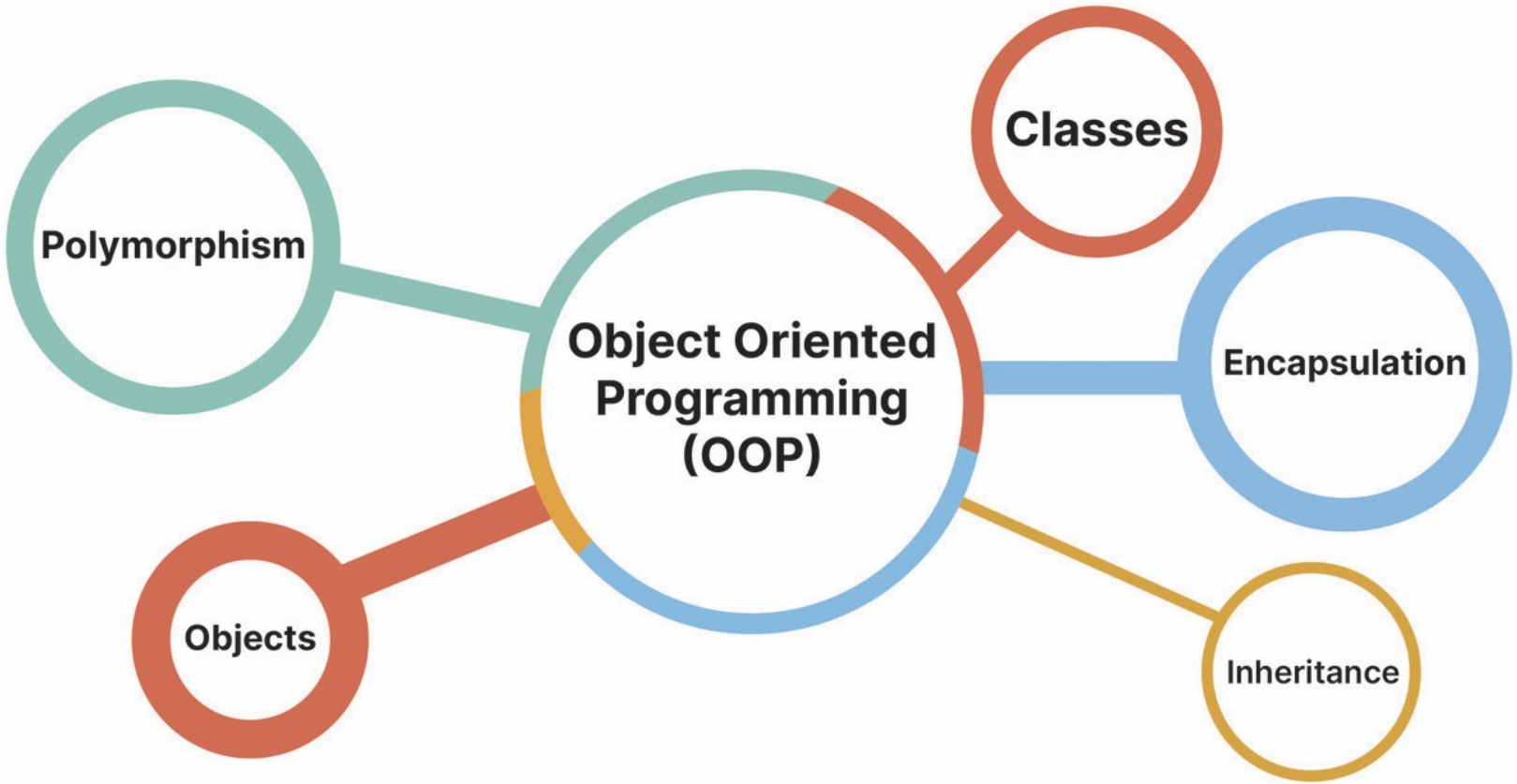
1. Introduction on object orientation
2. Python class
3. Class inheritance

Textbook: Chapter 17, Chapter 18, Chapter 19, Chapter 20, Chapter 21, Chapter 22, Chapter 23, Chapter26, Chapter 27

1. Introduction on object orientation

Object-oriented programming (OOP)

- A computer programming model that organizes software design around data, or objects, rather than functions and logic



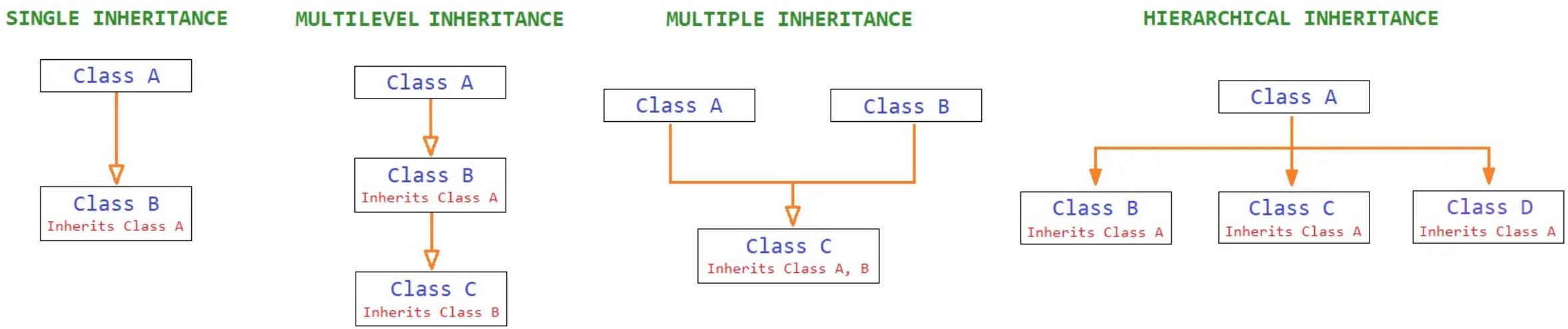
Key concepts

- Class
  - a blueprint for creating objects
  - defines a datatypes by bundling **data** and **methods** that operate on that data

- Object
  - an instance of a class
  - each object can have unique data (attributes) and share the structure and behavior defined by its class

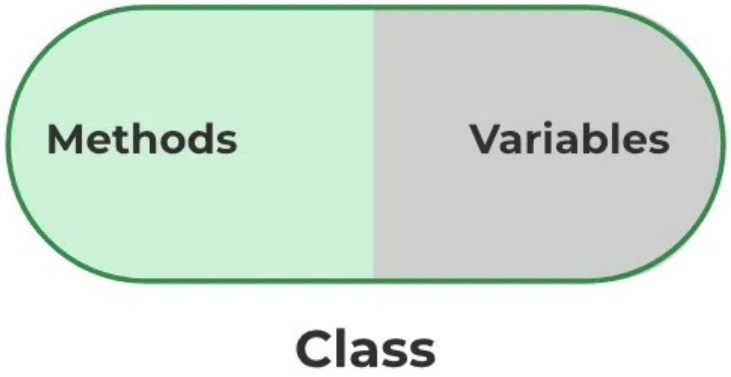
Key concepts

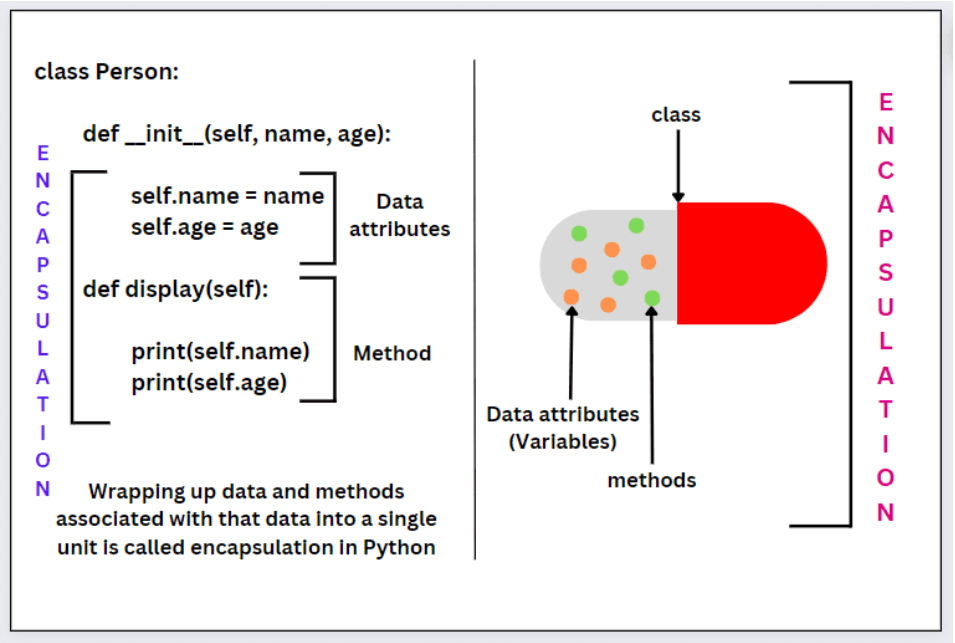
- Inheritance
  - a mechanism for a new class to inherit properties and behaviors from an existing class
  - allowing for code reuse and the creation of hierarchical relationships among classes



Key concepts

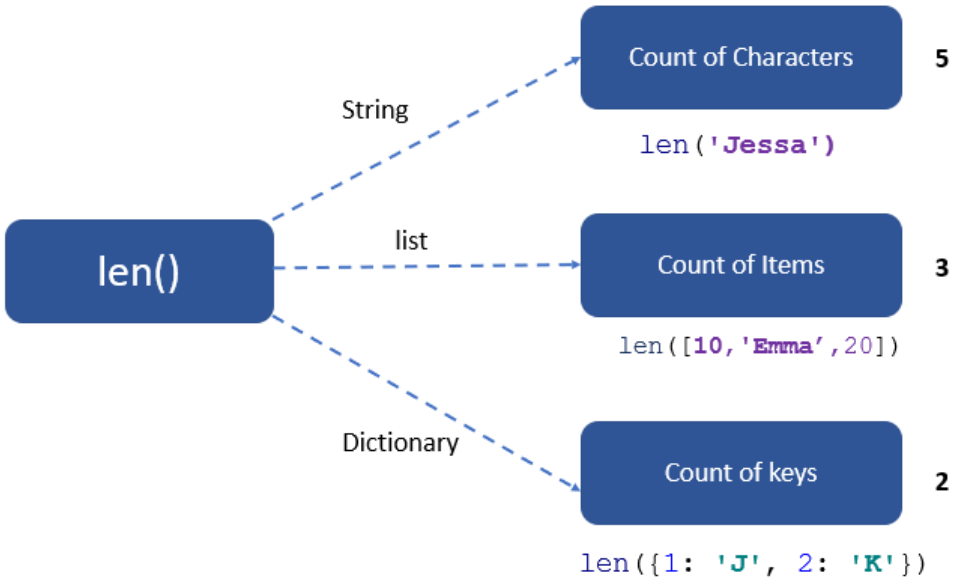
- Encapsulation
  - the bundling of data with the methods that operate on that data
  - restricting direct access to some of an objects' components, preventing accidental interference and misuse of the methods and data





Key concepts

- Polymorphism
  - the ability to present the same interface for differing underlying data types



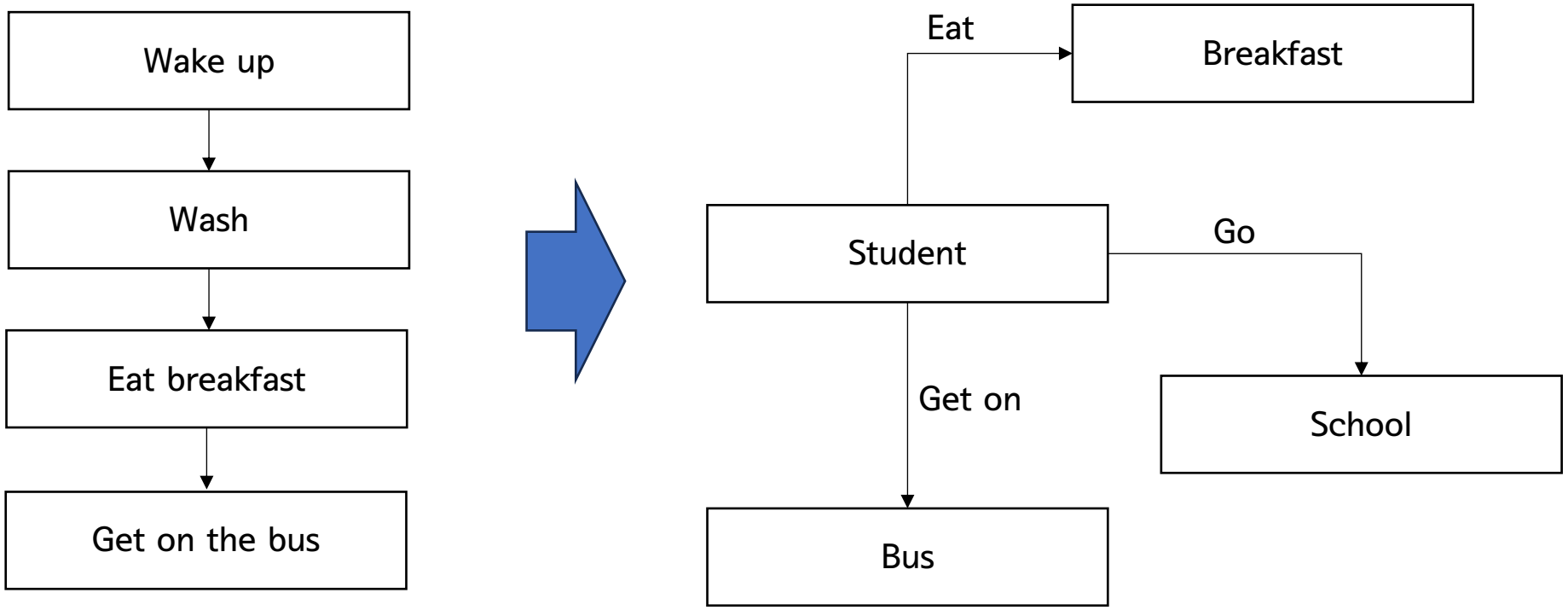
Polymorphic len() function

Advantages

- Modularity: the source code for a class can be written and maintained independently of the source code for other classes
- Reusability: classes can be reused in different programs
- Pluggable and debugging ease: objects are typically self-contained, and it is easy to swap out objects, as well as identify and fix issue
- Productivity
- Data redundancy
- Code flexibility
- Security
- …

OOP vs. Procedural programming

- Diagrams for OOP and procedural programming



2. Python class

Objects in Python

- Everything is “object” in Python
  - a type or class of thins; int, float, str, chr, dictionary, list, generator, etc.

```
print(type(4))
print(type(5.6))
print(type(True))
print(type('Ewan'))
print(type([1, 2, 3, 4]))
```

```
<class 'int'>
<class 'float'>
<class 'bool'>
<class 'str'>
<class 'list'>
```

Class definition

- Use `class` keyword to define a new class
  - General syntax

```
class nameOfClass(SuperClass):
    __init__
    attributes
    methods
```

  - three component in class
    - constructor (initializer) `__init__()`
      - naming convention: double underbars (`__`) as prefix and postfix in method name
    - attributes
      - member variables within an instance of the class
    - methods
      - the name given to behavior that is linked directly to the class; not free-standing function

Class definition

- `Person` class definition

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

  - `self` as special variable
    - indicates the values with `self` stored within an instance of class
    - convention in Python for the first parameter of a method in a class to be `self`
      - refers to the object itself

Class definition

- Roles of `self` in `Person` class

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

  - 1) defining instance methods
    - `self` is the first parameter in `__init__()` method
    - refers to the instance of `Person` that is being created
  - 2) accessing and setting attributes
    - `self.name = name` set the `name` attributes of the `self` to the value passed in the `name` parameter
    - `self.age = age` similarly set the `age` attributes to the value passed in the `age` parameter

Class definition

- Roles of `self` in `Person` class

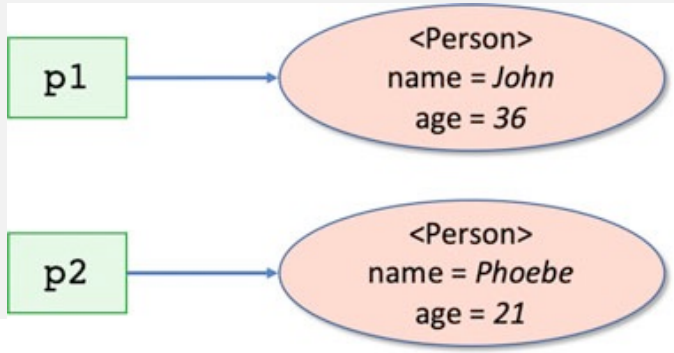
```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age
```

  - 3) creating an instance
    - when a new `Person` object is created with `person = Person("Allice", 30)`, a new instance of `Person` is created
    - Python automatically passes this new instance as the first argument to `__init__()` method
  - 4) accessing attributes
    - when you call `person.name` or `person.age`, accessing the `name` and `age` attributes of the `person` instance

Creating instances

- Instance creation of `Person` class

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age
'''
'''
p1 = Person("John", 36)
p2 = Person("Phoebe", 21)
```



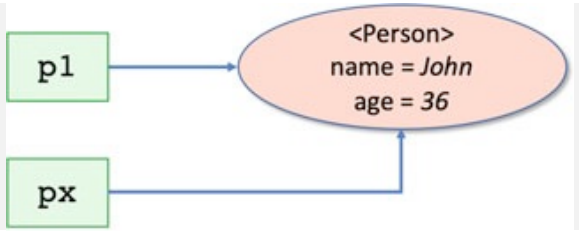
  - `p1` holds a reference to the instance or object of the class `Person` whose attributes hold the value 'John' (for name attribute) and 36 (for age attribute)
  - `p2` also holds 'Phoebe' and 21
  - `p1` and `p2` are "instance (or object)"; its own unique identifier

```
print(id(p1)) # Output: 1631806549904
print(id(p2)) # Output: 1631806550224
```

Note: Instance assignment

- Assignment the instance to another variable

```
p1 = Person("John", 36)
px = p1
print(id(p1)) # Output: 1631806549904
print(id(px)) # Output: 1631806549904
```



  - it holds the address of the object

Accessing object attributes

- Accessing the attributes using dot (`.`) notation
  - reading the attributes

```
print(p1.name, "is", p1.age)
print(p2.name, "is", p2.age)
```

```
John is 36
Phoebe is 21
```

  - updating the attribute of an object directly

```
p1.name = "Bob"
p1.age = 54
print(p1.name, "is", p1.age)
```

```
Bob is 54
```

Accessing object attributes

- Private variables
  - variable names prefixed with double underscores within a class definition

```
class Person:
    def __init__(self, name, age):
        self.__name = name
        self.__age = age
'''
'''
p1 = Person("John", 36)
print(p1.__name, "is", p1.__age)
```

```
Traceback (most recent call last):
  File "C:\Users\#2 파이썬프로그래밍\src\note.py", line 11, in <module>
    print(p1.__name, 'is', p1.__age)
          ^^^^^^^^^
AttributeError: 'Person' object has no attribute '__name'
```

- Protected variables (naming convention)
  - single underscores; `_name`

Default string representation

- Default string representation for a class
  - `__str__()`
    - used to create a readable string representation of an object
    - provide a friendly, readable representation suitable for display to end-users
  - `__repr()__`
    - provides an unambiguous representation of the object
    - Should return a string that when fed back to `eval()`, should ideally recreate the object or give a detailed description of the object
    - what is a `eval()`?

Default string representation

- Default string representation for a class

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

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

    def __repr__(self):
        return f"Person(name='{self.name}', age={self.age})"

person = Person("Alice", 30)
print(person)
print(repr(person))
```

```
Alice is 30
Person(name='Alice', age=30)
```

Instance methods

- Defining a method in `Person` class

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

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

    def __repr__(self):
        return f"Person(name='{self.name}', age={self.age})"

    def brithday(self):
        print("Happy birthday you were", self.age)
        self.age += 1
        print("You are now", self.age)
```

Instance methods

- Defining a method in `Person` class

```
p3 = Person("Adam", 19)
print(p3)
p3.brithday()
print(p3)
```

```
Adam is 19
Happy birthday you were 19
You are now 20
Adam is 20
```

Instance methods

- Defining a method in `Person` class

```
class Person:
    # ...
    def calculate_pay(self, hours_worked):
        rate_of_pay = 7.5
        if self.age >= 21:
            rate_of_pay += 2.50
        return hours_worked * rate_of_pay
'''
'''
pay = p2.calculate_pay(40)
print('Pay’, p2.name, pay)
pay = p3.calculate_pay(40)
print('Pay’, p3.name, pay)
```

```
Pay Phoebe 400.0
Pay Adam 300.0
```

Instance methods

- Static method
  - defined within a class but are not tied to either the class nor any instance of class
  - do not receive the special first parameter representing `self`
  - the same as free-standing functions, but defined without a class for convenience or to provide a way to group such function together

```
class Person:

    @staticmethod
    def static_function():
        print('Static method')

'''
'''
Person.static_function()
```

- Python does not provide method overloading

Removing instance

- Delete objects which allows the memory they are using to be reclaimed and used by other parts
  - Use `del` keyword

```
p1 = Person('John', 36)
print(p1)
del p1
```

Intrinsic attributes

- Every class has a set of intrinsic attributes set up
- Classes have the following intrinsic attributes:
  - `__name__` : the name of the class
  - `__module__` : the module (or library) from which it was loaded
  - `__bases__` : a collection of its base classes (see inheritance later in this book)
  - `__dict__` : a dictionary (a set of key-value pairs) containing all the attributes (including methods)
  - `__doc__` : the documentation string
- For objects:
  - `__class__` : the name of the class of the object
  - `__dict__` : a dictionary containing all the object's attributes.

Intrinsic attributes

```
p1 = Person("John", 36)
p2 = Person("Phoebe", 21)

print('Class attributes')
print(Person.__name__)

print('Object attributes')
print(p1.__class__)
print(p1.__dict__)
print(p2.__class__)
print(p2.__dict__)
```

```
Class attributes
Person
Object attributes
<class '__main__.Person'>
{'name': 'John', 'age': 36}
<class '__main__.Person'>
{'name': 'Phoebe', 'age': 21}
```

In class practice

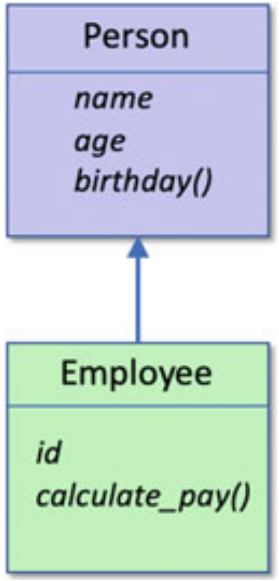
- P06-01 Write `Rectangle` class
  - attributes
    - width and height: width and height of rectangle
  - methods
    - __init__() and __str__()
    - area(): returns the size of rectangle
  - requirement: use dictionary as parameter as __init__() method

```
class Rectangle:
    ''' CODE HERE'''
'''
'''
rect = Rectangle(dict({'width': 10, 'height': 15}))
rect.area()
print(rect) # Output: width = 10 and height = 15
print(f"Size of rectangle = {rect.area()}")
```

3. Class inheritance

What is inheritance?

- Class inheritance in Python
  - a fundamental concept in OOP
  - allows a class (a.k.a subclass or child class) to inherit attributes and methods from another class (a.k.a. superclass or parent class)
  - promote code reusability and establishes a hierarchical relationship between classes



Descriptions

- Inherits features
  - a subclass inherits attributes and methods from the superclass, allowing it to reuse code
- Extensibility
  - a subclass can extends or modify the functionalities of the superclass
  - can add new attributes and methods or override existing ones (polymorphism)
- Hierarchical relationship
  - inheritance creates a tree-like hierarchy of classes, simplifies code organization and relationships between different entities

Syntax

- Subclass takes the name of superclass as parameter in definition line

```
class BaseClass:
    # Base class code

class DerivedClass(BaseClass):
    # Derived class code
```

Usage of class inheritance

- Subclass takes the name of superclass as parameter in definition line

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

    # ...

class Employee(Person):
    def __init__(self, name, age, employee_id, department):
        super().__init__(name, age)  # Call the initializer of the Person class
        self.employee_id = employee_id
        self.department = department

    # ...
```

  - using super() to call the super class

Usage of class inheritance

- Subclass takes the name of superclass as parameter in definition line

```
class Animal: # Superclass
    def __init__(self, species):
        self.species = species

    def make_sound(self):
        print("Some generic sound")

class Dog(Animal): # Subclass
    def __init__(self, species, name):
        super().__init__(species)
        self.name = name

    def make_sound(self):
        print("Woof!")

my_dog = Dog("Canine", "Buddy")
print(my_dog.species)  # Output: Canine
my_dog.make_sound()    # Output: Woof!
```

Method overriding

- A feature in OOP where a subclass provides a specific implementation of a method that is already defined in its parent class
  - allowing the subclass to customize or extend the behavior of that method

```
class Animal:
    def speak(self):
        return "This animal makes a generic sound"

class Dog(Animal):
    def speak(self):
        return "Woof! Woof!"

generic_animal = Animal()
print(generic_animal.speak())  # Output: This animal makes a generic sound

my_dog = Dog()
print(my_dog.speak())  # Output: Woof! Woof!
```

Quiz

- What is the result?

```
class A:
    def greet(self):
        return "Hello"

class B(A):
    pass

class C(B):
    def greet(self):
        return super().greet() + ", World!"

c = C()
print(c.greet())
```

- a) Hello
- b) Hello, World!
- c) World!
- d) Error

Quiz

- What is the result?

```
class Parent:
    def __init__(self):
        self.message = "Hello"

class Child(Parent):
    def __init__(self):
        super().__init__()
        self.message = self.message + "World"

child = Child()
print(child.message)
```

- a) Hello
- b) HelloWorld
- c) World
- d) Error

In class practice

- P06-02 Write `Circle` class and `Triangle` class that inherit from `Shape`
  - two methods (area() and perimeter()) override in `Circle` and `Triangle` classes

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

    def area(self):
        raise NotImplementedError("This method should be overridden by subclasses")

    def perimeter(self):
        raise NotImplementedError("This method should be overridden by subclasses")

class Circle(Shape):
    ''' CODE HERE '''

class Triangle(Shape):
    ''' CODE HERE '''
```

In class practice

- P06-02 Write `Circle` class and `Triangle` class that inherit from `Shape`
  - two methods (area() and perimeter()) override in `Circle` and `Triangle` classes
  - expected result

```
circle = Circle(5)
print(f"Area of {circle.name} is {circle.area():.2f} and perimeter is {circle.perimeter():.2f}")

triangle = Triangle(4)
print(f"Area of {triangle.name} is {triangle.area():.2f} and perimeter is
{triangle.perimeter():.2f}")
```

```
The area of the Circle is 78.54 and the perimeter is 31.42
The area of the Triangle is 6.93 and the perimeter is 12.00
```

End of slide