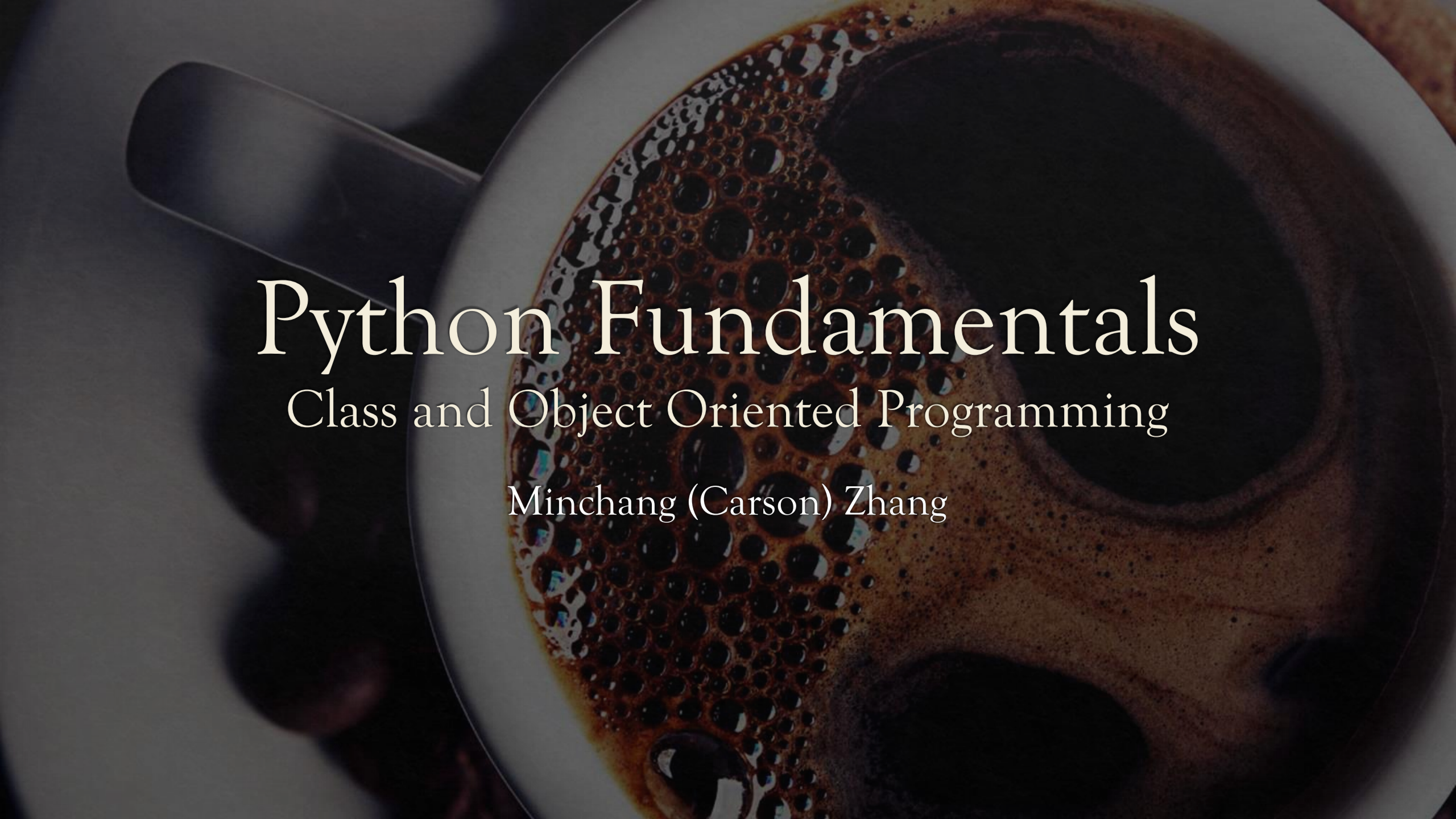

Python Fundamentals

Class and Object Oriented Programming

Minchang (Carson) Zhang

Python Fundamentals

01

CLASS

- DEFINITION
- ATTRIBUTES
- DECORATORS
- FUNCTIONS AND DUNDER METHODS

02

OOP

- BENEFITS
- TERMINOLOGIES

03

INHERITANCE

- SUPER()
- CUSTOM EXCEPTION

04

MULTI-INHERITANCE

- DIAMOND PROBLEM
- MIXINS
- ABSTRACT CLASS
- POLYMORPHISM

Class - Definition

- In Python 2 we have to inherit from object explicitly (i.e. "*old-style classes*"), which is not recommended. (i.e. you should write it as class Person~~(object)~~:)
- __init__: initialize the class instance.
  - When we call the class object, a new instance of the class is created, and the __init__ method on this new object is immediately executed with all the parameters that are passed to the class object.
  - __new__ is used for class construction
- self: the first parameter for class method definition
  - represents the instance of the class
  - allows access to the attributes and method of class

```
class Person:

    def __init__(self, name, surname, birthdate, address, telephone, email):
        self.name = name
        self.surname = surname
        self.birthdate = birthdate

        self.address = address
        self.telephone = telephone
        self.email = email

    def age(self):
        today = datetime.date.today()
        age = today.year - self.birthdate.year

        if today < datetime.date(today.year, self.birthdate.month, self.birthdate.day):
            age -= 1

        return age

person = Person(
    "Jane",
    "Doe",
    datetime.date(1992, 3, 12), # year, month, day
    "No. 12 Short Street, Greenville",
    "555 456 0987",
    "jane.doe@example.com"
)
```

Class - Attributes

- Instance attribute:
  - **Convention**: leading underscore (_) is used to indicate the attribute or method is internal property and should not be accessed directly
  - **The best practice** is to create **ALL** attributes inside the __init__ method, otherwise we run the risk of trying to use it before its been initialized

```
class Person:

    TITLES = ('Dr', 'Mr', 'Mrs', 'Ms')

    def __init__(self, title, name, surname):
        if title not in self.TITLES:
            raise ValueError("%s is not a valid title." % title)

        self.title = title
        self.name = name
        self.surname = surname
```

Class attribute

Instance attribute

- Class attribute: attributes which are set on the *class*
  - These attributes will be shared by all instances of that class
  - Class attributes are often used to define constants which are closely associated with a particular class
  - We should use *immutable* type

Class – Decorators

@classmethod	@staticmethod	@property
• First parameter is cls instead of self • We can use it without needing to create any class instance	• Doesn't have the calling object passed into it as first parameter (i.e. no cls or self) • Doesn't have access to class	• Makes a method behaves like an attribute • Has builtin getter, setter, and delete methods

Class - Functions

- dir: return the list of names in the current local scope
- hasattr(object, name): check if the name (str) is one of the object's attribute
- getattr(object, name): Return the value of the named (str) attribute of *object*
- setattr(object, name, value): programmingly assign the value to the name (str) of the object's attribute

Class - Dunder Methods

- __init__: initialisation method of an object, called when the object is created
- __str__: the string representation of an object
- __class__: an attributes that stores the class/type of an object (i.e. this returns when type is called)
- __eq__: used in object comparison (i.e. when == is called)
- __add__: arithmetic operation, allows this object to be added to another object
- __iter__: returns an iterator over the object (i.e. used in loops such as for, while, etc)
- __len__: calculates the length of object (i.e. when len is called)
- __dict__: a dictionary contains all the instance attributes of an object. It does not include any methods, class attributes or special default attributes

Object Oriented Programming (OOP)

- Group relevant variables and functions into logical subgroups
- Valuable when the program grows in size and complexity
- Allows code to be reused so the development time is reduced
- Easier to maintain

Object Oriented Programming (OOP)

- *Encapsulation*: the idea that data inside the object should only be accessed through a public *interface* – the object's method
  - the functionality is defined *in one place* and not in multiple places.
  - it is defined in a logical place – the place where the data is kept.
  - data inside our object is not modified unexpectedly by external code in a completely different part of our program.
  - when we use a method, we only need to know what result the method will produce
- Encapsulation is not enforced by Python, but convention indicates that a property is intended to be private and is not part of the object's public interface begins its name with an *underscore*

OOP – Inheritance

- *Inheritance* is a way of arranging objects in a hierarchy from the most general to the most specific.
- We often say that a class is a *subclass* or *child class* of a class from which it inherits, or that the other class is its *superclass* or *parent class*.
- We can refer to the most generic class at the base of a hierarchy as a *base class*.

Inheritance

- super(): used to call parent class methods
  - __mro__ and mro(): **method resolution order**, it traces back the order in which base classes are searched for a member during lookup
  - __init__ with super(): used in subclasses to initialize both the subclass and parent class attributes.
    - Preferred syntax in Python 3: super(~~ChildB, self~~).__init__()
- Custom Exception: define custom classes for exceptions which we want to raise in our code
  - We can efficiently write except blocks which handle groups of related exceptions

Multi - Inheritance

- *diamond problem:* if classes B and C inherit from A and class D inherits from B and C, and both B and C have a method called do_something, which do_something will D inherit?
- Mixins: a class which is not intended to stand on its own
  - It exists to add extra functionality to another class through multiple inheritance.
  - Many mixins just provide additional methods and don't initialize anything
- Abstract classes: a class that we can't use to create an object directly, we can only inherit from the class and use the subclasses to create objects
  - It is useful to create a class which serves as a *template* for suitable objects by defining a list of methods that these objects must implement
  - A common way is to use NotImplementedError inside our method definitions
- *Polymorphism:* a way to define methods in the child class with the same name as defined in their parent class
  - It allows us to have different behaviour for a method in different classes (*Method Overriding*)

Reference

- Object-Oriented Programming in Python
- Inheritance and Polymorphism in Python
- Understanding Python MRO - Class search path
- Python Object Oriented Programming
- Object-Oriented Programming in Python 3

Questions?

Thank you!