LECTURE 17: CLASS AND METHODS

Hung-Yu Wei

OOP (Object-Oriented Programming)

- Class and method definitions
- Operations on objects
- Objects represent things in the real world
- Methods correspond to the ways things in the real world interact

Methods v.s. Functions

- Definition
  - *Methods are defines within a class definition*
- Invoking a method v.s. invoking a function

Example in Chapter 17

- **Time** Class
  - *Re-write with **methods***
  - ***User-defined methods** for Time class*
    - print_time()
    - increment()
    - add_time()
    - is_after()
  - ***Special methods***
    - init method (initialization)
    - str method (string print)
    - add +
      - *Operator overloading*
- Concepts
  - *Polymorphism*
  - *Separate interface from implementation*

Function v.s. Methods – print_time()

Function

```
class Time:
    """Represents the time of day."""

def print_time(time):
    print('%.2d:%.2d:%.2d' % (time.hour, time.minute, time.second))
```

```
>>> start = Time()
>>> start.hour = 9
>>> start.minute = 45
>>> start.second = 00
>>> print_time(start)
09:45:00
```

Method

```
class Time:
    def print_time(time):
        print('%.2d:%.2d:%.2d' % (time.hour, time.minute, time.second))
```

```
>>> Time.print_time(start)
09:45:00
```

```
>>> start.print_time()
09:45:00
```

Method of a class

- Definition of class

```
class Time:
    def print_time(self):
        print('%.2d:%.2d:%.2d' % (self.hour, self.minute, self.second))
```

  - *self*
    - Conventionally, you call it "self" (but not mandatory)

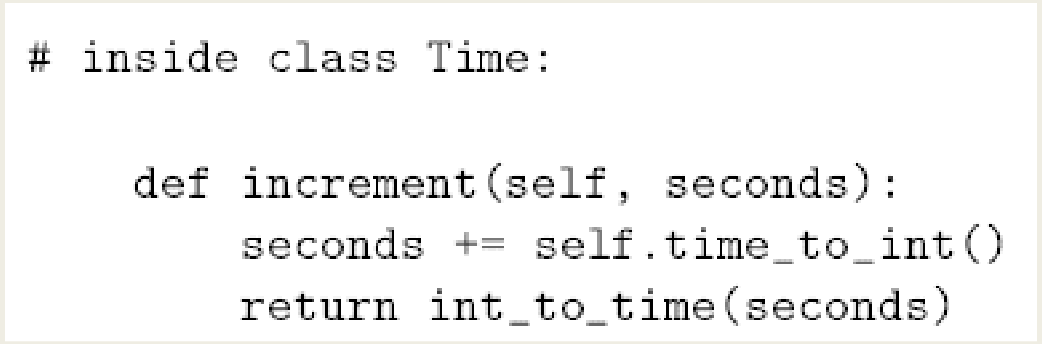
- Invoke

```
>>> start.print_time()
```

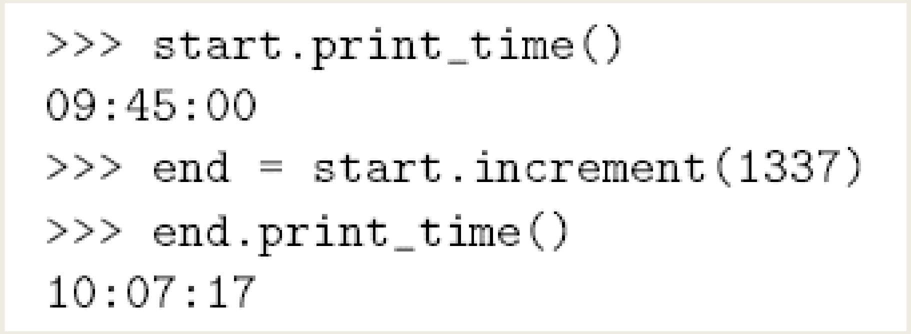
  - *dot*
  - *Object_name.method_name*
- "Hey start! Please print yourself."

Dot operation to access (1) attribute (2) method

Re-write increment() as a method



```
# inside class Time:

    def increment(self, seconds):
        seconds += self.time_to_int()
        return int_to_time(seconds)
```



```
>>> start.print_time()
09:45:00
>>> end = start.increment(1337)
>>> end.print_time()
10:07:17
```

Re-write is_after()

```
# inside class Time:

    def is_after(self, other):
        return self.time_to_int() > other.time_to_int()
```

```
>>> end.is_after(start)
True
```

init method

```
__init__
```

- Initialization: __init__()
  - *special method*
  - *Automatically invoked when an object is instantiated*
  - *Initialization*

```
# inside class Time:

    def __init__(self, hour=0, minute=0, second=0):
        self.hour = hour
        self.minute = minute
        self.second = second
```

```
>>> time = Time()
>>> time.print_time()
00:00:00
```

```
>>> time = Time (9)
>>> time.print_time()
09:00:00
```

```
>>> time = Time(9, 45)
>>> time.print_time()
09:45:00
```

str method

`__str__`

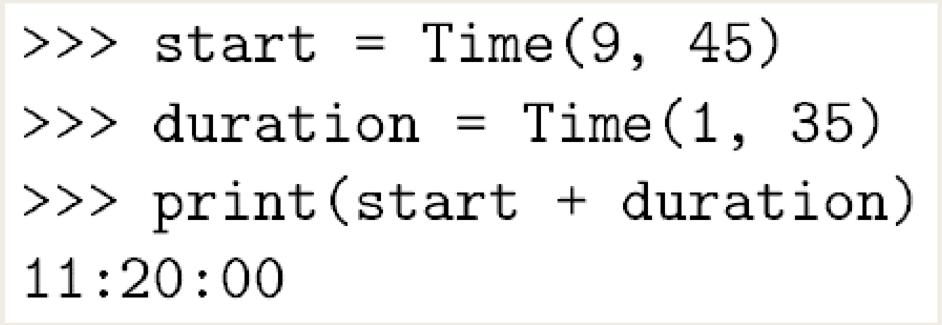
- str method
  - *Special method*
  - *Return a string*
    - Print an object

```
>>> time = Time(9, 45)
>>> print(time)
09:45:00
```

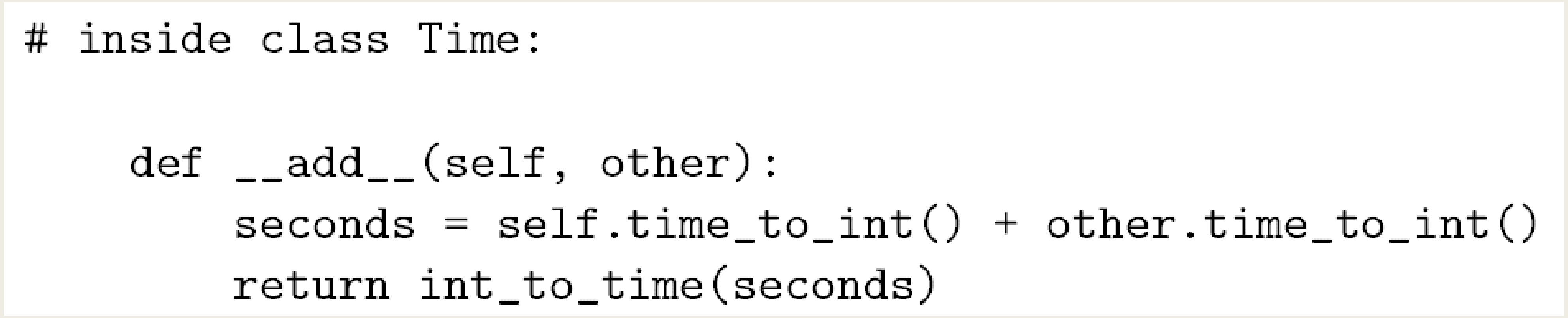
```
# inside class Time:

    def __str__(self):
        return '%.2d:%.2d:%.2d' % (self.hour, self.minute, self.second)
```

Operator overloading

- You use operator for different types
  For example, +, -
- Operator + for a new class
  __add__



```
>>> start = Time(9, 45)
>>> duration = Time(1, 35)
>>> print(start + duration)
11:20:00
```



```
# inside class Time:

    def __add__(self, other):
        seconds = self.time_to_int() + other.time_to_int()
        return int_to_time(seconds)
```

- List of special methods
  - https://docs.python.org/3/reference/datamodel.html#specialnames

Type-based dispatch

- You have seen 2 types of addition
  - *Add for time objects*
    - Add_time()
  - *Add for seconds*
    - Increment()

- isinstance(value, ClassObject)
  - *True if value is ClassObject*

- +

You want to automatically call the corresponding addition

```
# inside class Time:

    def __add__(self, other):
        if isinstance(other, Time):
            return self.add_time(other)
        else:
            return self.increment(other)

    def add_time(self, other):
        seconds = self.time_to_int() + other.time_to_int()
        return int_to_time(seconds)

    def increment(self, seconds):
        seconds += self.time_to_int()
        return int_to_time(seconds)
```

timeObj1+timeObj2

timeObj1+20

__radd__

20+timeObj1

- time+time
- time+second
- second + time
  - *Right-side add*
  - *__radd__*

___ + Time

```
>>> start = Time(9, 45)
>>> duration = Time(1, 35)
>>> print(start + duration)
11:20:00
```

```
>>> print(start + 1337)
10:07:17
```

```
>>> print(1337 + start)
TypeError: unsupported operand type(s) for +: 'int' and 'instance'
```

```
# inside class Time:

    def __radd__(self, other):
        return self.__add__(other)
```

```
>>> print(1337 + start)
10:07:17
```

Polymorphism (多型; 多種型態)

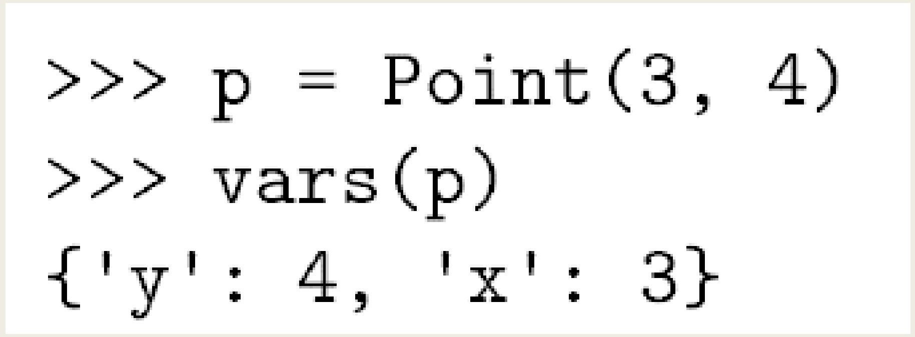
- Polymorphism is a key feature in OOP
- Functions that work with several types
- Time works with +
- Time works with sum

The new Time class behavior is similar to other types (int, floating point)

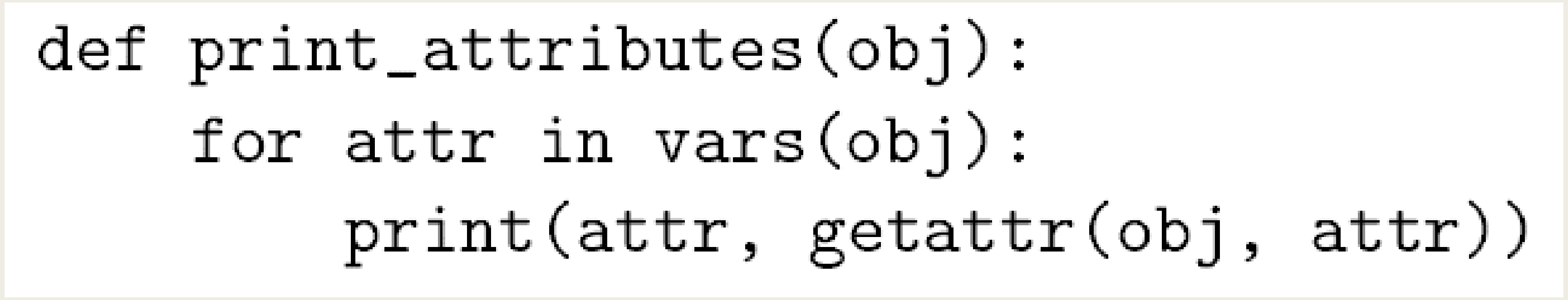
```
>>> t1 = Time(7, 43)
>>> t2 = Time(7, 41)
>>> t3 = Time(7, 37)
>>> total = sum([t1, t2, t3])
>>> print(total)
23:01:00
```

Debugging: check the attributes

- Built-in function
  - *hasattr*
  - *getattr*
  - *vars*
    - Dictionary of attributes



```
>>> p = Point(3, 4)
>>> vars(p)
{'y': 4, 'x': 3}
```



```
def print_attributes(obj):
    for attr in vars(obj):
        print(attr, getattr(obj, attr))
```

Interface and implementation

- Interface
  - *time-consuming and error-prone to change the interface after deployment*
- Implementation
  - *change the implementation without changing the interface*
  - *other parts of the program don’t have to change.*

Summary

- Chapter 17 in textbook “Think Python”
- Summary:
  - *Time Class*
    - Special methods of a class
      - *__init__*
      - *__str__*
      - *__add__*
      - *__radd__*
    - Methods
      - *print_time()*
      - *is_after()*
      - *increment()*
      - *is_valid()*
      - *add_time()*

Polymorphism

Operator Overloading