Think Python 2e, Chapter 17 Notes

Classes and Methods

November 1, 2022

Methods

- Need tighter relationship between classes and the functions that deal with them.
- Methods are semantically the same as functions.
- The syntax for methods is different from functions.
- Methods are defined inside a class definition.
- This makes the relation between class and method explicit.

NON object-oriented way

```
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
```

Only way to call the function:

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

Object-oriented way

```
class Time:
    """Represents the time of day."""
    def print_time(time):
        print(’%.2d:%.2d:%.2d’ %
              (time.hour, time.minute, time.second))
```

There are now two ways to call the function:

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

- The second is more concise.
- `start` is the actual parameter bound to `time`
- `start` is called the **subject**

self

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

- By convention, the formal parameter is usually called `self`

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

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

Function-oriented *vs.* object-oriented programming

Function is focus:

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

Object is focus:

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

Function-oriented *vs.* object-oriented programming

Function is focus:

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

Object is focus:

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

- Notice you can write `time_to_int` as a method, but not `int_to_time`.
- Why not?

increment

```python
# inside class Time:

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

- This is a pure function

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

- `increment` is defined with two formal parameters
- `increment` is called with one subject and one actual parameter

Error message can be confusing

```
>>> end = start.increment(1337, 460)
TypeError: increment() takes 2 positional arguments but 3 were given
```

- But I only gave two parameters!

Error message can be confusing

```
>>> end = start.increment(1337, 460)
TypeError: increment() takes 2 positional arguments but 3 were given
```

- But I only gave two parameters!
- Wrong! You gave the subject and two parameters.
- That's three

Positional arguments

- A **positional argument** is an argument that doesn't have a parameter name; that is, it is not a keyword argument.

```
sketch(parrot, cage, dead=True)
```

- `parrot` and `cage` are positional, and `dead` is a keyword argument.

Methods with two objects

```python
# inside class Time:

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

- `self` and `other` are conventional names.

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

__init__

```python
# inside class Time:

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

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

__str__

```
# inside class Time:

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

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

Operator overloading

- Every operator in Python has a dunder method to overload it.
- Here we overload addition, *i.e.* the + operator.

```
# inside class Time:

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

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

Type based dispatch

```
# 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)
```

Type based dispatch

We can now use this as follows

```
>>> start = Time(9, 45)
>>> duration = Time(1, 35)
>>> print(start + duration)
11:20:00
>>> print(start + 1337)
10:07:17
```

Unfortunately it is not commutative. Solution:

```
# inside class Time:

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

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

Polymorphism

Functions that work with several types are **polymorphic**

```python
def histogram(s):
    d = dict()
    for c in s:
        if c not in d:
            d[c] = 1
        else:
            d[c] = d[c]+1
    return d
```

```python
>>> t = ['spam', 'egg', 'spam', 'spam', 'bacon', 'spam']
>>> histogram(t)
{'bacon': 1, 'egg': 1, 'spam': 4}
>>> histogram('banana')
{'b': 1, 'a': 3, 'n': 2}
```

Polymorphism and code reuse

sum will work with any items which support addition

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

Polymorphism frequently surprises us and works for types we didn't even know they would.

Debugging

- It is legal to add attributes anywhere in the execution of a program.
- It is still a bad idea: same types should have same attributes.
- Add attributes only in the `__init__` method.
- If you have to check if an object has an attribute use `hasattr`
- Can also use `vars` which takes an object and returns a dictionary mapping from attribute names to values.
- `getattr` takes an object and an attribute name and returns the attribute's value.

```
item.x == getattr(item, 'x')
```

Interface and Implementation

- Separate interfaces from implementations
- Methods should not depend on attributes
- Example:
    - `Time` used attributes: `hour`, `minute`, `second`
    - Instead, it could have used just: `seconds`
    - Different methods are easier with different representations.
- The user of the interface should not know the implementation.
- You can change the implementation, to make it faster, smaller, whatever, without changing the interface.
- Code that uses the class should not change when the implementation changes.

Vocabulary

object-oriented language: A language that provides features, such as programmer-defined types and methods, that facilitate object-oriented programming.

object-oriented programming: A style of programming in which data and the operations that manipulate it are organized into classes and methods.

method: A function that is defined inside a class definition and is invoked on instances of that class.

subject: The object a method is invoked on.

Random Warning!

Don't initialize objects with mutables!

Vocabulary

positional argument: An argument that does not include a parameter name, so it is not a keyword argument.

operator overloading: Changing the behavior of an operator like $+$ so it works with a programmer-defined type.

type-based dispatch: A programming pattern that checks the type of an operand and invokes different functions for different types.

polymorphic: Pertaining to a function that can work with more than one type.