Objects and Classes II

Thomas Schwarz, SJ
Marquette University

Classes and Objects 2

- Classes usually define objects, but they can also used in isolation
  - Assume that you want to use a number of global variables
    - This is dangerous, since you might be reusing the same name
  - Solution: Use a class that contains all these variables

A Globals Class

- We call the class Gl — short for global
- Store constants as class variables
- Easy to identify in program

```
class Gl:
    gr2gr = 0.06479891
    dr2gr = 1.7718451953125
    oz2gr = 28.349523125
    lb2gr = 453.59237
    st2gr = 6350.29318
```

```
def translate(number, measure):
    if measure == "gr":
        return "{0:.3f} {1}".format(number*Gl.gr2gr, "gram")
    if measure == "dr":
        return "{0:.3f} {1}".format(number*Gl.dr2gr, "gram")
    if measure == "oz":
        return "{0:.3f} {1}".format(number*Gl.oz2gr, "gram")
    if measure == "lb":
        return "{0:.3f} {1}".format(number*Gl.lb2gr, "gram")
    if measure == "st":
        return "{0:.3f} {1}".format(number*Gl.st2gr/1000, "kg")
    raise ValueError
```

Class and Instance Variables

- Class variable
  - belong to the class
  - shared by all objects
  - defined without prefix in the class
- Instance variable
  - belong to the instance
  - not shared by objects
  - defined by using an object or self prefix

Self Test

- Identify the type of the bold-faced variables in the following code

```
import math

class Example:
    exists = False
    def __init__(self, x, y):
        self.radius = math.sqrt(x*x+y+y)
        self.x = x
        self.y = y
        Example.exists = True

print(Example.exists)
e = Example(2, 3)
print(e.x)
print(Example.exists)
print(e.radius)
```

Answer

```
import math

class Example:
    exists = False
    def __init__(self, x, y):
        self.radius = math.sqrt(x*x+y+y)
        self.x = x
        self.y = y
        Example.exists = True

print(Example.exists)
e = Example(2, 3)
print(e.x)
print(Example.exists)
print(e.radius)
```

This is an instance variable. It belongs to the (one and only) object of type Example.

It happens to be defined in __init__. However, it is defined with the self prefix.

Answer

```
import math

class Example:
    exists = False
    def __init__(self, x, y):
        self.radius = math.sqrt(x*x+y+y)
        self.x = x
        self.y = y
        Example.exists = True

print(Example.exists)
e = Example(2, 3)
print(e.x)
print(Example.exists)
print(e.radius)
```

This is a class variable. It is specified by using the class name “Example.”

It is defined without a prefix within the class.

Answer

```
import math

class Example:
    exists = False
    def __init__(self, x, y):
        self.radius = math.sqrt(x*x+y+y)
        self.x = x
        self.y = y
        Example.exists = True

print(Example.exists)
e = Example(2, 3)
print(e.x)
print(Example.exists)
print(e.radius)
```

This is an instance variable. It is defined with the prefix self.

It is used by referring to an object e.

Class and Instance Methods

- The same distinction can be made for methods
  - Methods are functions related to an object
- A class method depends only on the class.
  - It is defined in the class, but has no argument self
  - It is called by giving the class-name
- An instance method depends on an instance
  - It is defined in the class with first argument `self`
  - It is called by prefacing it with an instance.
    - The instance is called the implicit argument

Class and Instance Methods

```
class Example:
    def foo():
        print("foo")
    def __init__(self):
        pass
    def bar(self):
        print("bar")

Example.foo()
e = Example()
e.bar()
```

A method definition without argument self: Class Method

It is called using the class-name to call it

Class and Instance Methods

```
class Example:
    def foo():
        print("foo")
    def __init__(self):
        pass
    def bar(self):
        print("bar")

Example.foo()
e = Example()
e.bar()
```

A method definition with
argument self:
Instance Method

It is called using the
Instance.
Without an object e, we
cannot call it.

Self Test

- Identify the type of methods in the following code

```
import math
class Vector3D:
    def __init__(self, x, y, z):
        self.x = x
        self.y = y
        self.z = z
    def zeroes():
        return Vector3D(0,0,0)
    def ones():
        return Vector3D(1,1,1)
    def __add__(self, other):
        return Vector3D(self.x+other.x,
                        self.y+other.y,
                        self.z+other.z)
    def __str__(self):
        return "({}, {}, {})".format(self.x, self.y, self.z)
    def length(self):
        return math.sqrt(self.x**2+self.y**2+self.z**2)
```

Answers

Dunder (double under) method:
Hard to tell

```
import math
class Vector3D:
    def __init__(self, x, y, z):
        self.x = x
        self.y = y
        self.z = z
    def zeroes():
        return Vector3D(0,0,0)
    def ones():
        return Vector3D(1,1,1)
    def __add__(self, other):
        return Vector3D(self.x+other.x,
                        self.y+other.y,
                        self.z+other.z)
    def __str__(self):
        return "({}, {}, {})".format(self.x, self.y, self.z)
    def length(self):
        return math.sqrt(self.x**2+self.y**2+self.z**2)
```

Answers

```
import math
class Vector3D:
    def __init__(self, x, y, z):
        self.x = x
        self.y = y
        self.z = z
    def zeroes():
        return Vector3D(0,0,0)
    def ones():
        return Vector3D(1,1,1)
    def __add__(self, other):
        return Vector3D(self.x+other.x,
                        self.y+other.y,
                        self.z+other.z)
    def __str__(self):
        return "({}, {}, {})".format(self.x, self.y, self.z)
    def length(self):
        return math.sqrt(self.x**2+self.y**2+self.z**2)
```

Class Method, even though it generates an object

Answers

```
import math
class Vector3D:
    def __init__(self, x, y, z):
        self.x = x
        self.y = y
        self.z = z
    def zeroes():
        return Vector3D(0,0,0)
    def ones():
        return Vector3D(1,1,1)
    def __add__(self, other):
        return Vector3D(self.x+other.x,
                        self.y+other.y,
                        self.z+other.z)
    def __str__(self):
        return "({}, {}, {})".format(self.x, self.y, self.z)
    def length(self):
        return math.sqrt(self.x**2+self.y**2+self.z**2)
```

Class Method, even though it generates an object

Answers

```
import math
class Vector3D:
    def __init__(self, x, y, z):
        self.x = x
        self.y = y
        self.z = z
    def zeroes():
        return Vector3D(0,0,0)
    def ones():
        return Vector3D(1,1,1)
    def __add__(self, other):
        return Vector3D(self.x+other.x,
                        self.y+other.y,
                        self.z+other.z)
    def __str__(self):
        return "({}, {}, {})".format(self.x, self.y, self.z)
    def length(self):
        return math.sqrt(self.x**2+self.y**2+self.z**2)
```

Instance method

Answers

```
import math
class Vector3D:
    def __init__(self, x, y, z):
        self.x = x
        self.y = y
        self.z = z
    def zeroes():
        return Vector3D(0,0,0)
    def ones():
        return Vector3D(1,1,1)
    def __add__(self, other):
        return Vector3D(self.x+other.x,
                        self.y+other.y,
                        self.z+other.z)
    def __str__(self):
        return "({}, {}, {})".format(self.x, self.y, self.z)
    def length(self):
        return math.sqrt(self.x**2+self.y**2+self.z**2)
```

Dunder instance method

Answers

```
import math
class Vector3D:
    def __init__(self, x, y, z):
        self.x = x
        self.y = y
        self.z = z
    def zeroes():
        return Vector3D(0,0,0)
    def ones():
        return Vector3D(1,1,1)
    def __add__(self, other):
        return Vector3D(self.x+other.x,
                        self.y+other.y,
                        self.z+other.z)
    def __str__(self):
        return "({}, {}, {})".format(self.x, self.y, self.z)
    def length(self):
        return math.sqrt(self.x**2+self.y**2+self.z**2)
```

Instance method

Dunder Methods

- Python reserves special names for functions that allows the programmer to emulate the behavior of built-in types
  - For example, we can create number like objects that allow for operations such as addition and multiplication
  - These methods have special names that start out with two underscores and end with two underscores

- Aside: If you preface a variable / function / class with a single underscore, you indicate that it should be treated as reserved and not used outside of the module / class

Dunder Method

- A class for playing cards:
  - A card has a suit and a rank
    - We define this in the constructor __init__

```
class Card:
    def __init__(self, suit, rank):
        self.suit = suit
        self.rank = rank
```

Dunder Method

- We want to print it
  - Python likes to have two methods:
    - __repr__   for more information, e.g. errors
    - __str__ for the print-function
    - Both return a string

```
class Card:

    def __str__(self):
        return self.suit[0:2]+self.rank[0:2]
    def __repr__(self):
        return "{}-{}".format(self.suit, self.rank)
```

Dunder Method

- __repr__ is used when we create an object in the terminal

```
>>> Card("Heart", "Queen")
Heart-Queen
```

- __str__ is used within print or when we say `str(card)`

```
>>> print(Card("Heart", "Queen"))
HeQu
>>> str(Card("Heart", "Queen"))
'HeQu'
```

Dunder Method

- We now create a carddeck class
    - Consists of a set of cards
    - Constructor uses a list of ranks and a list of suits

```
class Deck:
    def __init__(self, los, lov):
        self.cards = [Card(suit, rank) for suit in los
                                       for rank in lov]
```

Dunder Method

- We create the string method. Remember that it needs to return a string.

```
class Deck:
    def __init__(self, los, lov):
        self.cards = [Card(suit, rank) for suit in los
                                       for rank in lov]
    def __str__(self):
        result = []
        for card in self.cards:
            result.append(str(card))
        return " ".join(result)
```

Dunder Method

- In order to allow python to check whether a deck exists, we want to have a length class. Besides, it is useful in itself.
  - `if deck:` works by checking `len(deck)`

```
class Deck:

    def __len__(self):
        return len(self.cards)
```

Dunder Method

- Given a deck, we want to be able to access the i-th element.
- We do so by defining __getitem

```
class Deck:

    def __getitem__(self, position):
        return self.cards[position]
```

Dunder Method

- This turns out to be very powerful:

```
french_deck = Deck(['Spade', 'Diamonds', 'Hearts', 'Clubs'],
                   ['Ace', 'King', 'Queen', 'Jack', '10', '9',
                    '8', '7', '6', '5', '4', '3', '2'])
```

  - We can print out the i-th element of the deck

```
>>> str(french_deck[5])
'Sp9'
```

  - But we can also **slice** the deck

```
>>> print(french_deck[6:12])
[Spade-8, Spade-7, Spade-6, Spade-5, Spade-4, Spade-3]
```

Dunder Method

- We can use random.choice() to select a card

```
>>> random.choice(french_deck)
Diamonds-9
```

- Only for random.sample do we need to go to the underlying instance field

```
>>> random.sample(french_deck.cards, 5)
[Hearts-8, Hearts-2, Hearts-Ace, Hearts-6, Diamonds-Ace]
>>> random.sample(french_deck.cards, 5)
[Hearts-5, Clubs-Queen, Diamonds-Ace, Clubs-3, Clubs-King]
```

  - But this is ugly and we better write a class method for it.