13.1_OOP(2)

April 11, 2023

1 Introduction to Python for Open Source Geocomputation

- Instructor: Dr. Wei Kang
- Class Location and Time: ENV 336, Mon & Wed 12:30 pm - 1:50 pm

Content:

- Polymorphism

[1]:
```
print({1:2})
```

```
{1: 2}
```

[2]:
```
print("asb")
```

```
asb
```

[3]:
```
print([12,2])
```

```
[12, 2]
```

[4]:
```
class Point:
    def __init__(self, x=3, y=4):
        self.x = x
        self.y = y
```

[5]:
```
p2 = Point()
```

[6]:
```
print(p2)
```

```
<__main__.Point object at 0x7fa36106fd30>
```

1.0.1 print representation of an object

```
class Point:
    def __init__(self, x=3, y=4):
        self.x = x
        self.y = y

    def __str__(self):
        return "<"+str(self.x)+","+str(self.y)+">"
```

- uninformative print representation by default
- define a `__str__` method for a class (special method)
- Python calls the `__str__` method when used with `print` on your class object
- you choose what it does! Say that when we print a Point object, we want to display its x,y corrdinates

[7]:
```
class Point:
    def __init__(self, x=3, y=4):
        self.x = x
        self.y = y

    def __str__(self):
        return "<"+str(self.x)+","+str(self.y)+">"
```

[8]:
```
p2 = Point()
```

[9]:
```
print(p2)
```

```
<3,4>
```

[10]:
```
print({1:2})
```

```
{1: 2}
```

Function Polymorphism `print`

1.0.2 Polymorphism: an important property of OOP

- the use of a single type entity (method, operator or object) to represent different types in different scenarios.
- Polymorphic `print()` function
  - `print("python")`: print the string
  - `print(p2)`: print the x and y coordinates in the format of `<x,y>`
- Polymorphic `len()` function
  - `len("python")`: number of characters in the string
  - `len([1,2,"python"])`: number of items in the list

  - `len({1:2, "python": 3})`: number of keys/key-value pairs in the dictionary
- Polymorphism in addition operator `+`
  - `1+2`
  - `"python"+ " "+ "good"`

[11]:
```
len("python")
```

[11]: 6

[12]:
```
len([1,2,"python"])
```

[12]: 3

[13]:
```
len({1:2, "python": 3})
```

[13]: 2

[14]:
```
1+2
```

[14]: 3

[15]:
```
"python"+ " "+ "good"
```

[15]: 'python good'

[16]:
```
class Point:
    def __init__(self, x=3, y=4):
        self.x = x
        self.y = y

    def __str__(self):
        return "<"+str(self.x)+","+str(self.y)+">"
```

[17]:
```
p1 = Point()
p2 = Point()
```

[18]:
```
p1 + p2
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[18], line 1
----> 1 p1 + p2

TypeError: unsupported operand type(s) for +: 'Point' and 'Point'
```

[ ]:
```
print(p1.x, p1.y)
```

[19]:
```
print(p1)
```

```
<3,4>
```

[20]:
```
print(p2)
```

```
<3,4>
```

[21]:
```
print(p1) == print(p2)
```

```
<3,4>
<3,4>
```

[21]:
```
True
```

[22]:
```
1 == 1
```

[22]:
```
True
```

[23]:
```
p1 == p2
```

[23]:
```
False
```

1.0.3 Special operators

- `+`, `-`, `==`, `<`, `>`, `len()`, `print`, and many others
- like `print`, we can write special methods to override these to work with your class
- define them with **double underscores before/after**
  - `__add__(self, other)`: `self + other`
  - `__sub__(self, other)`: `self - other`
  - `__eq__(self, other)`: `self == other`
  - `__lt__(self, other)`: `self < other`
  - `__len__(self)`: `len(self)`
  - `__str__(self)`: `print self`
  - … and others
    - `self` refers to the current instance
    - `other` refers to another instance of the same class

1.0.4 Customizing + for class `Point`

```
class Point:
    def __init__(self, x=3, y=4):
        self.x = x
        self.y = y

    def __str__(self):
        return "<"+str(self.x)+","+str(self.y)+">"
```

```
    def __add__(self, other):
        return Point(self.x+other.x, self.y+other.y)
```

[24]:
```
p1 = Point(2,3)
```

[25]:
```
p1 + Point(3,4)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[25], line 1
----> 1 p1 + Point(3,4)

TypeError: unsupported operand type(s) for +: 'Point' and 'Point'
```

[26]:
```
class Point:
    def __init__(self, x=3, y=4):
        self.x = x
        self.y = y

    def __str__(self):
        return "<"+str(self.x)+","+str(self.y)+">"

    def __add__(self, other):
        return Point(self.x+other.x, self.y+other.y)
```

[27]:
```
p1 = Point()
p2 = Point(4,5)
```

[28]:
```
p3 = p1 + p2
print(p3)
```

```
<7,9>
```

[29]:
```
print(p3.x, p3.y)
```

```
7 9
```

Exercise:

```
class Point:
    def __init__(self, x=3, y=4):
        self.x = x
        self.y = y
    def __add__(self, other):
        return Point(self.x+other.x, self.y+other.y)
```

Customize the operator `==` for class `Point` to compare one instance of `Point` with another. If both the x and y coordinates are equal, `True` is returned, otherwise, `False` is returned. (hint: Define the method `__eq__(self, other)` to calculate `self == other`)

Raise your hand when you are done

[30]:
```
class Point:
    def __init__(self, x=3, y=4):
        self.x = x
        self.y = y
    def __add__(self, other):
        return Point(self.x+other.x, self.y+other.y)

    def __eq__(self, other):
        if self.x == other.x and self.y == other.y:
            return True
        else:
            return False
```

[31]:
```
p1 = Point()
p2 = Point(4,5)
```

[32]:
```
p1 == p2
```

[32]:
```
False
```

[33]:
```
p1 = Point(4,5)
p2 = Point(4,5)
```

[34]:
```
p1 == p2
```

[34]:
```
True
```

1.0.5 Power of OOP

- create our own classes of objects on top of Python's basic classes
- bundle together objects that share
  - common attributes
  - procedures that operate on those attributes
- OOP models complex things as reproducible, simple structures
- Reusable, OOP objects can be used across programs
- Allows for class-specific behavior through polymorphism
- Easier to debug, classes often contain all applicable information to them