12_OOP(1)

April 5, 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:

- Object-oriented programming (OOP)
- Objects in Python
- Class
  - attribute
  - method

1.1 Objects in Python

- Python supports many different types of data

```
1234, 3.14159, "Hello", True, [1, 5, 7, 11, 13], (1,2,3), {1,2,3}, {"CA": "California","TX": "
```

- In Python, everything is an object. All objects have:
  - an unique ID, or location in the computer's memory;
  - a set of properties that describe the object;
  - a set of methods, or things that the object can do.
- Every object is an **instance** of a type
  - `1234` is an instance of an int
  - `"hello"` is an instance of a string

[1]:
```
type(int)
```

[1]:
```
type
```

[2]:
```
type(1234)
```

[2]:
```
int
```

[3]:
```
id(1234)
```

[3]:
```
140637772250704
```

[4]:
```
type("hello")
```

[4]:
```
str
```

[5]:
```
type(str)
```

[5]:
```
type
```

1.2 Object-oriented programming (OOP)

Object-oriented programming (OOP) is a programming paradigm based on the concept of "objects", which can contain data and code. The data is in the form of fields (often known as attributes or properties), and the code is in the form of procedures (often known as methods).

- **EVERYTHING IN PYTHON IS AN OBJECT (and has a type)**
- can **create new objects** of some type
- can **manipulate objects**
- can **destroy objects**
  - explicitly using `del` or just "forget" about them
    - `del obj_name` # delete an object

[6]:
```
90
```

[6]:
```
90
```

[7]:
```
list_a = [1,2,3]
```

[8]:
```
list_a.pop()
```

[8]:
```
3
```

[9]:
```
list_a
```

[9]:
```
[1, 2]
```

[10]:
```
del list_a
```

[11]:
```
list_a
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[11], line 1
----> 1 list_a

NameError: name 'list_a' is not defined
```

1.3 What are objects?

Objects are a data abstraction that captures:

1. an internal representation
   - through data **attributes**
2. an interface for interacting with object
   - through **methods** (aka procedures/functions)
   - defines behaviors but hides implementation

1.3.1 EXAMPLE of Objects:

`["rich", "lisa", "andy", "zach", "tia"]` has type `list`

- how are lists represented internally?
  - linked list of cells
  - Dynamic Array (python)
- how to manipulate lists?
  - `L[i], L[i:j], +`
  - `len(), min(), max(), del L[i]`
  - `L.append(),L.extend(),L.count(),L.index(), L.insert(),L.pop(),L.remove(),L.reverse(), L.sort()`
- Internal representation should be private

1.4 Other Potential Objects in Python

- An entity of "thing" in your program

Example	attributes/data/properties	behaviors/procedures/methods
person	name, age, address	walk, talk, breathe
employee	id, name, age, position, salary	raise salary, promote
door	height, color, locked	open, close, lock
point	coordinates of the point	move to a different location, distance to another point
polygon	coordinates of the points comprising the polygon, coordinate system	relationship with other polygons (overlap, contain)

1.4.1 Car as a class in python

1.4.2 Advantages of OOP

- bundle **data** into packages together with **procedures** that work on them through well-defined **interfaces** - **classes** and their **instances**
- **divide-and-conquer** development
  - implement and test behavior of each class separately
  - increased modularity and reduces complexity
- classes make it easy to **reuse** code
  - each class has a separate environment (no collision on function names)
  - inheritance allows subclasses to redefine or extend a selected subset of a superclass' behavior
  - many Python modules/packages define new classes

1.5 Classes and instances

- We can create many instances/objects from a single class
- Classes define a **type**
- The process of creating an object from a class is called **instantiation**
- An object is an instance of a class

1.5.1 Example of `Door` class and its instances

	properties	behaviors	
Door	height, color, is_locked	open(), close(), lock()	Class
door1	height:80, color:"red",is_locked:False	open(), close(), lock()	instance
door2	height:95, color:"gray",is_locked:True	open(), close(), lock()	instance

- Class:
  - a blueprint for how something should be defined
  - like a form or questionnaire
  - doesn't actually contain any data
- Instance:
  - an object that is built from a class
  - like a form that has been filled out with information
  - contains real data

1.5.2 Defining a type/Class in Python

```
class Point:
    pass #pass is often used as a placeholder indicating where code will eventually go
```

- Keyword `class`
- Python class names are written in CapitalizedWords notation by convention (e.g., `MarkovChains`).
- indentation

[12]:
```
class Point:
    pass
```

[13]:
```
help(Point)
```

```
Help on class Point in module __main__:

class Point(builtins.object)
 |  Data descriptors defined here:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)
```

[14]:
```
type(int)
```

[14]: type

[15]:
```
type(Point)
```

[15]: type

1.5.3 Defining a Class in Python with attributes

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

- Two attirbutes:
  - `x`: x-coordinate
  - `y`: y-coordinate
- `self`:
  - a special parameter living inside the class
  - represents the instance of the class
  - used to access the attributes and methods of the class: `self.coordinates`, `self.coordinate_system`
- `__init__()`:
  - constructor: a special method to create an instance
  - When a new Polygon object is created, `__init__()` is called to set the initial state of the object by assigning the values of the object's properties
  - `.__init__()` initializes an instance of the `Point` class.
  - accepts any number of parameters
    - the first parameter will always be `self`, representing the new object/instance that is going to be created

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

[17]:
```
point1 = Point(12, 2)
```

[18]:
```
point1.x
```

[18]: 12

[19]:
```
point1.y
```

[19]: 2

[20]:
```
help(Point)
```

```
Help on class Point in module __main__:

class Point(builtins.object)
 |  Point(x, y)
 |
 |  Methods defined here:
 |
 |  __init__(self, x, y)
 |      Initialize self.  See help(type(self)) for accurate signature.
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors defined here:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)
```

1.5.4 Defining a Class in Python with attributes and methods

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

    def translate(self, dx, dy):
        '''
        Translate the point dx units to the right and dy units up
        '''
```

```
        self.x = self.x + dx
        self.y = self.y + dy
```

- Two attributes
- One method in addition to the constructor method:
  - One method `translate` for translating the point
  - `self` is required as an argument in each of the class methods
  - method is a function that works only with this class

[21]:
```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def translate(self, dx, dy):
        '''
        Translate the point dx units to the right and dy units up
        '''
        self.x = self.x + dx
        self.y = self.y + dy
```

[22]:
```
help(Point)
```

```
Help on class Point in module __main__:

class Point(builtins.object)
 |  Point(x, y)
 |
 |  Methods defined here:
 |
 |  __init__(self, x, y)
 |      Initialize self.  See help(type(self)) for accurate signature.
 |
 |  translate(self, dx, dy)
 |      Translate the point dx units to the right and dy units up
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors defined here:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)
```

1.5.5 Creating an instance from a class

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

    def translate(self, dx, dy):
        '''
        Translate the point dx units to the right and dy units up
        '''
        self.x = self.x + dx
        self.y = self.y + dy

p1 = Point(1,2)
```

- instantization
- `class_name(arguments for the __int__ method)`
- `Point(x, y)`
  - `self` is ignored in instantization
  - the rest of the method has the some syntax

[23]:
```
p1 = Point(1,2)
```

[24]:
```
type(p1)
```

[24]:
```
__main__.Point
```

1.5.6 Accessing attributes and reassigning attribute values

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

    def translate(self, dx, dy):
        '''
        Translate the point dx units to the right and dy units up
        '''
        self.x = self.x + dx
        self.y = self.y + dy
p1 = Point(1,2)
print(p1.x)
print(p1.y)
```

- `instance_name.`: dot operator
- `p1.x`: accessing instance `p1`'s attribute `x`
- `p1.x = 10`: change the value of instance `p1`'s attribute `x`

[25]:
```
p1 = Point(1,2)
```

[26]:
```
p1.x
```

[26]: 1

[27]:
```
p1.y
```

[27]: 2

[28]:
```
p1.x = 10
```

[29]:
```
p1.x
```

[29]: 10

class is mutuble, but write to data from outside class definition is not recommended - does not maintain data encapsulation

read more on data encapsulation and ways to ensure that in python https://python-course.eu/oop/properties-vs-getters-and-setters.php

[30]:
```
p1.y
```

[30]: 2

1.5.7 Calling Methods

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

    def translate(self, dx, dy):
        '''
        Translate the point dx units to the right and dy units up
        '''
        self.x = self.x + dx
        self.y = self.y + dy
p1 = Point(1,2)
```

- one method `translate` in addition to the constructor method
  - other than `self` and dot notation, methods behave just like functions (take params, do operations, return)

Two ways of calling the method of an object:

- `self` is implict: `object.method(parameters)`:

```
p1.translate(0.5, 0.5)
```

- `self` is explict: `Class.method(object, parameters)`:

```
Point.translate(p1, 0.5, 0.5)
```

[31]:
```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def translate(self, dx, dy):
        '''
        Translate the point dx units to the right and dy units up
        '''
        self.x = self.x + dx
        self.y = self.y + dy
```

[32]:
```
p1 = Point(1,2)
```

[33]:
```
p1.translate(0.5, 0.5)
```

[34]:
```
print(p1.x, p1.y)
```

```
1.5 2.5
```

[35]:
```
Point.translate(p1, 0.5, 0.5)
```

[36]:
```
print(p1.x, p1.y)
```

```
2.0 3.0
```

[37]:
```
a = [1,2,3]
a.pop()
```

[37]: 3

[38]:
```
a = [1,2,3]
list.pop(a)
```

[38]: 3

Exercise:

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

    def translate(self, dx, dy):
        '''
```

```
        Translate the point dx units to the right and dy units up
        '''
        self.x = self.x + dx
        self.y = self.y + dy
```

Add another method `distance` to class `Point` which calcuates and returns the distance from the point to the origin `(0,0)`. (hint: distance = $\sqrt{(x-0)^2 + (y-0)^2}$

Raise your hand when you are done

[39]:
```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def translate(self, dx, dy):
        '''
        Translate the point dx units to the right and dy units up
        '''
        self.x = self.x + dx
        self.y = self.y + dy

    def distance(self):
        return (self.x**2 + self.y **2) ** 0.5
```

[40]:
```
p1 = Point(1,2)
```

[41]:
```
p1.distance()
```

[41]:
```
2.23606797749979
```

[42]:
```
p1 = Point(3,4)
```

[43]:
```
p1.distance()
```

[43]:
```
5.0
```

Exercise:

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

How would we make it possible to construct a Point with no parameters/arguments (users do not need to provide any inputs) to get `3,4` as coordinates?

[1]:
```
class Point:
    def __init__(self, x=3, y=4):
```

```
        self.x = x
        self.y = y
```

[2]:
```
p2 = Point()
```

[3]:
```
print(p2.x, p2.y)
```

```
3 4
```

[4]:
```
p3 = Point(1,2)
```

[5]:
```
print(p3.x, p3.y)
```

```
1 2
```

[6]:
```
p4 = Point(y=1,x=2)
```

[7]:
```
print(p4.x, p4.y)
```

```
2 1
```