MONASH University

MONASH
INFORMATION
TECHNOLOGY

FIT9133 Semester 1 2019
Programming Foundations in Python

Week 6:
Classes and Variable Scope

Chunyang Chen
Gavin Kroeger

GROUP OF EIGHT AUSTRALIA

Module 3 Synopsis

- Module 3 is aimed to introduce you with:
  - Concepts of decomposition
    - Functions and methods
    - Modules in Python
  - Concepts of classes and methods
    - Implementation
    - Object instantiation

- Upon completing this module, you should be able to:
  - Identify how to decompose a computational program into manageable units of functions and/or classes

Object-Oriented Programming: Objects, Classes and Methods

Objects

- Python supports many different kinds of data:
  - 1234 3.1415 “Python” [1,2,3]
    {“AU”: ”Australia”, “US”: ”United States”}
- Each is an object, and every object has
  - A type
  - An internal data representation (primitive or composite)
  - A set of procedures for interacting with the object
- An object is an instance of a type
  - 1234 is an instance of an int
  - “Python” is an instance of a string

Objects

- **Everything in Python is an Object** with a type
  - Can create new objects of some type
  - Can manipulate objects
  - Can destroy objects
    - Explicitly using del or just “forget” about them
    - Python system will reclaim destroyed or inaccessible objects – called “garbage collection”

Objects

- What are objects?
  - Object is a data abstraction including:
  - an internal representation
    - through data attributes
  - an interface for interacting with object
    - by methods (functions)

- Object-oriented programming (OOP):
  - Conceptualise a real-world scenario as to how multiple groups of objects interact to build an application
  - Each type of objects represents one specific kind of concept in the real world

- Fundamental concepts of OOP:
  - Creation of objects
  - Encapsulate both the attributes and the behaviours of the objects (i.e. the ways how objects interact with each other)

Advantages of OOP

- Divide-and-conquer development:
  - Implement and test behaviour of each class separately
  - Increased modularity reduces complexity

- Easy reuse of code
  - Many python modules define new classes
  - 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

Classes

Object-oriented programming

- Definition of classes:
  - Designed to represent only one concept within an application
  - Defined as a template ("blue-print") to create objects of a specific type
  - Multiple classes are integrated to build a complete application

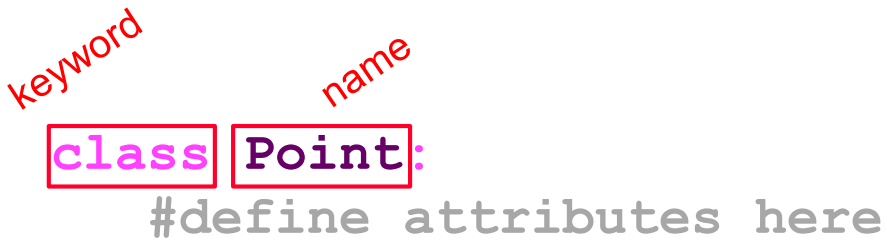
- Instances of a class:
  - Objects constructed from a specific class
  - Each instance is assigned to a variable name (reference) to access its internal data values and the associated methods

Each class defines a set of "instance variables" (data values to be represented for each object) and a set of "methods" (operations) that can be applied on the objects.

Class Implementation

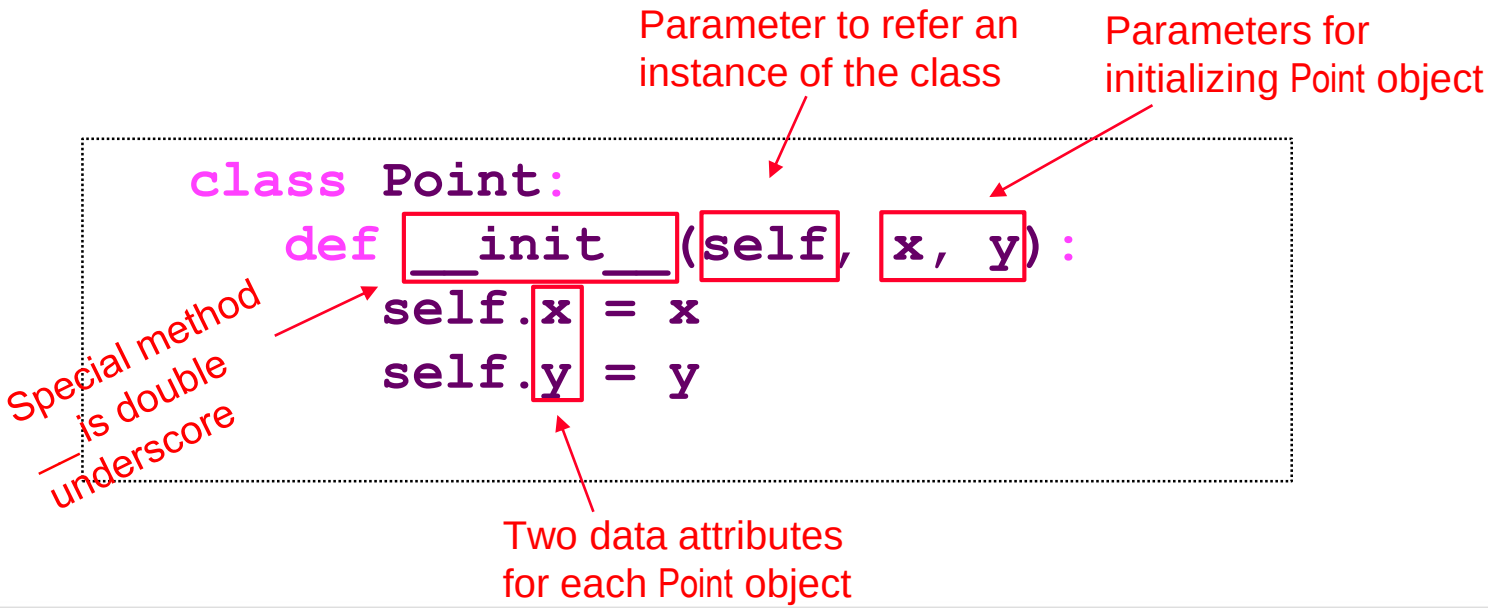
- Class header:
  - Starts with the keyword **`class`** and followed by a class name
  - Naming convention: CapWords

- Example: The Point class



Class Implementation

- Constructor: `__init__(self)`
  - The essential method for object creation
  - Initialise the values of the instance variables of each object
  - Invoked based on the class name
  - Instance variables: x and y values for representing each point in a two-dimensional space



Using the class

- Data attributes of an instance are called instance variables

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

```
>>> p = Point(1,2)
>>> print(p.x)
>>> print(p.y)
```

Class Implementation

- method
  - procedural attribute, like a function, but works only with this class
  - Operations to interact with the class
  - Invoked based on the class name

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

    def distance(self, other):
        x_diff = (self.x - other.x) ** 2
        y_diff = (self.y - other.y) ** 2
        distance = (x_diff + y_diff) ** 0.5
        return distance
```

Use it to refer to the instance

Another object as the method parameter

Using the class

- Use a method in the class
  - Conventional way

```
>>> p = Point(3,4)
>>> orig = Point(0, 0)
>>> print(p.distance(orig))
```

self is implied to be p

  - Equivalent to

```
>>> p = Point(3,4)
>>> orig = Point(0, 0)
>>> print(Point.distance(p, orig))
```

Parameters, including an object to call the method on, presenting self

Using the class

- Print representation of an object

```
>>> p = Point(3,4)
>>> print(p)
<__main__.Student object at 0x00000190F419F748>
```

  - Define a `__str__` method for a class for nice printing
  - Python automatically calls the __str__ method when used with `print()` on the class object

Using the class

- Define your own print method

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

    def distance(self, other):
        x_diff = (self.x - other.x) ** 2
        y_diff = (self.y - other.y) ** 2
        distance = (x_diff + y_diff) ** 0.5
        return distance

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

Name of special method

Must return a string

Using the class

- Define your own print method

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

    def distance(self, other):
        x_diff = (self.x - other.x) ** 2
        y_diff = (self.y - other.y) ** 2
        distance = (x_diff + y_diff) ** 0.5
        return distance

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

Name of special method

Must return a string

```
>>> p = Point(1, 2)
>>> print(p)
<1, 2>
```

- SPECIAL OPERATORS
  - +, -, ==, len(), print, and many others
  - 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
  - https://docs.python.org/3/reference/datamodel.html#basic-customization

Class Implementation

- **`self`:**
  - Each method defined within the class must have **`self`** as the first argument
  - Does not need to be specified during method invocation
  - Automatically set to *reference* the object on which the method is invoked

```
class Point:
    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y

    def get_x(self):
        return self.x
    def get_y(self):
        return self.y

    def set_x(self, x = 0):
        self.x = x
    def set_y(self, y = 0):
        self.y = y
```

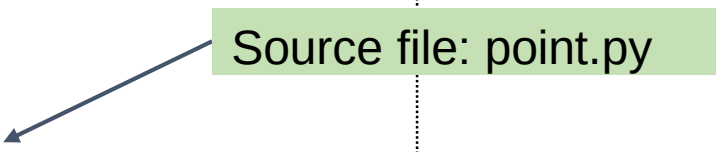
Instance variables

Accessors/getter

Mutators/setter

(More on) Class Implementation

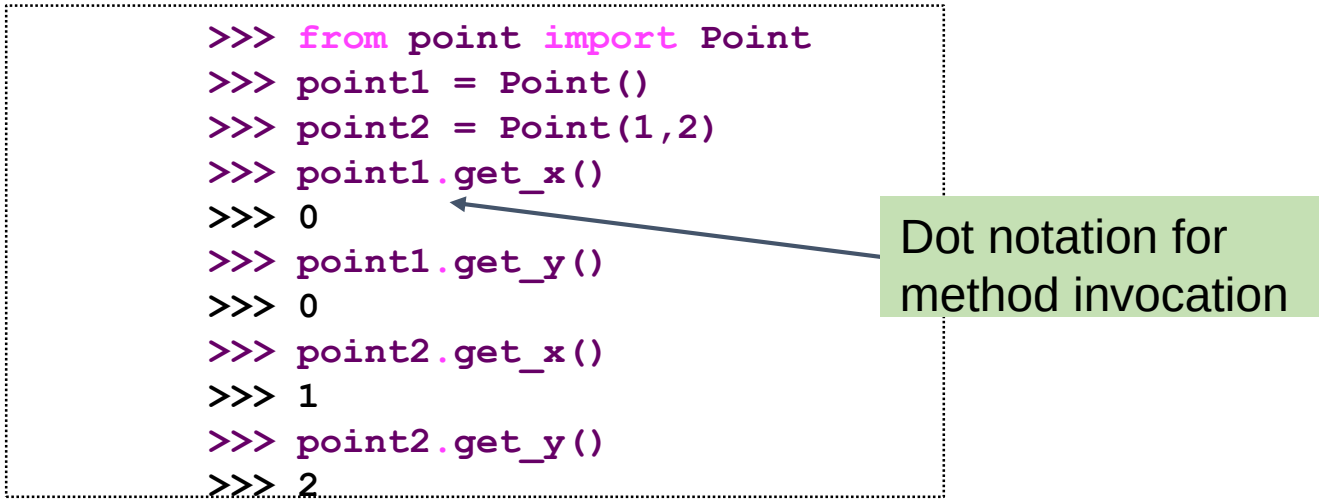
```
class Point:
    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y

    def get_x(self):
        return self.x
    def get_y(self):
        return self.y

    def set_x(self, x = 0):
        self.x = x
    def set_y(self, y = 0):
        self.y = y

    def distance(self, other):
        x_diff = (self.x - other.x) ** 2
        y_diff = (self.y - other.y) ** 2
        distance = (x_diff + y_diff) ** 0.5
        return distance
```



Object Instantiation

- To use a class for object creation in another program:
  - Must first import the class:
    `from <module_name> import <ClassName>`
- To construct a new object:
  - Syntax: `object_name = ClassName(arg1, arg2, …)`
  - E.g.: `a_point = Point(1,0)` or `a_point = Point()`
  - Note that `self` is not passed as an argument

```
>>> from point import Point
>>> point1 = Point()
>>> point2 = Point(1,2)
>>> point1.get_x()
>>> 0
>>> point1.get_y()
>>> 0
>>> point2.get_x()
>>> 1
>>> point2.get_y()
>>> 2
```

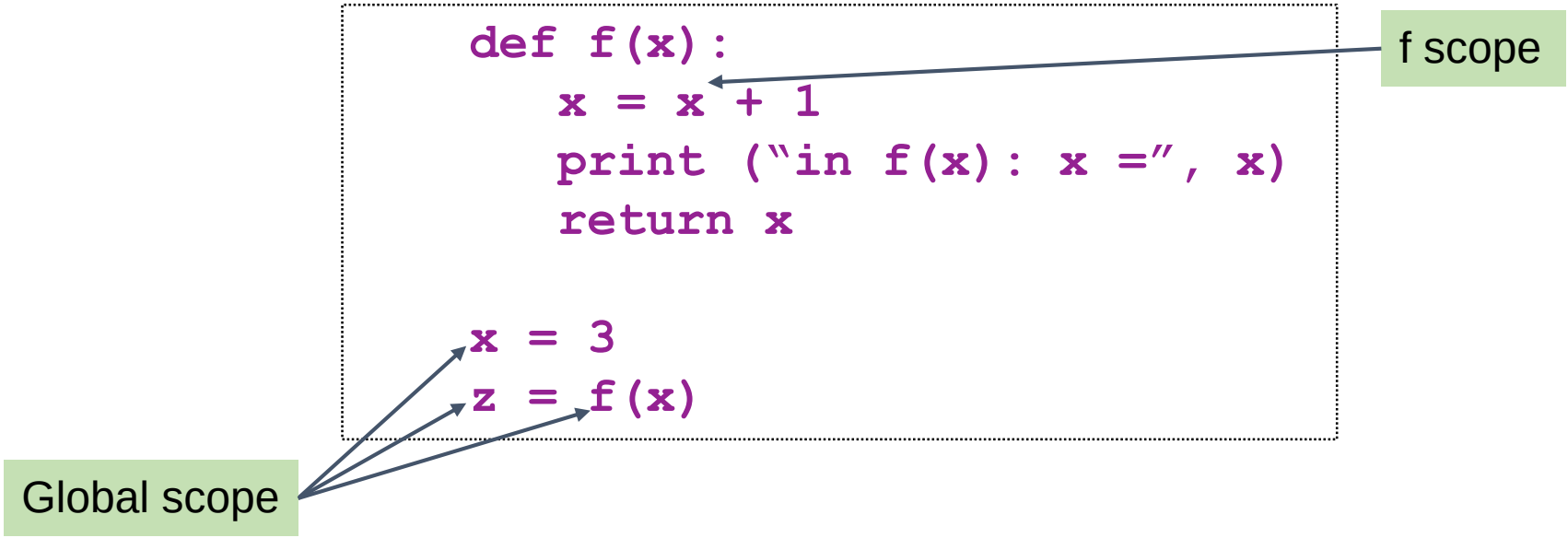


Object-Oriented Programming: Variable Scope

Variable Scoping and Lifetime

- Scoping:
  - Define the part of the program where a variable is accessible
- Lifetime:
  - Define the duration for which a variable exists during the program execution
- Global variables:
  - Can be accessed throughout the entire program
  - Exists until the execution of the program terminated
- Local variables:
  - Can only be accessed within the function it was defined
  - Exists until the function exists

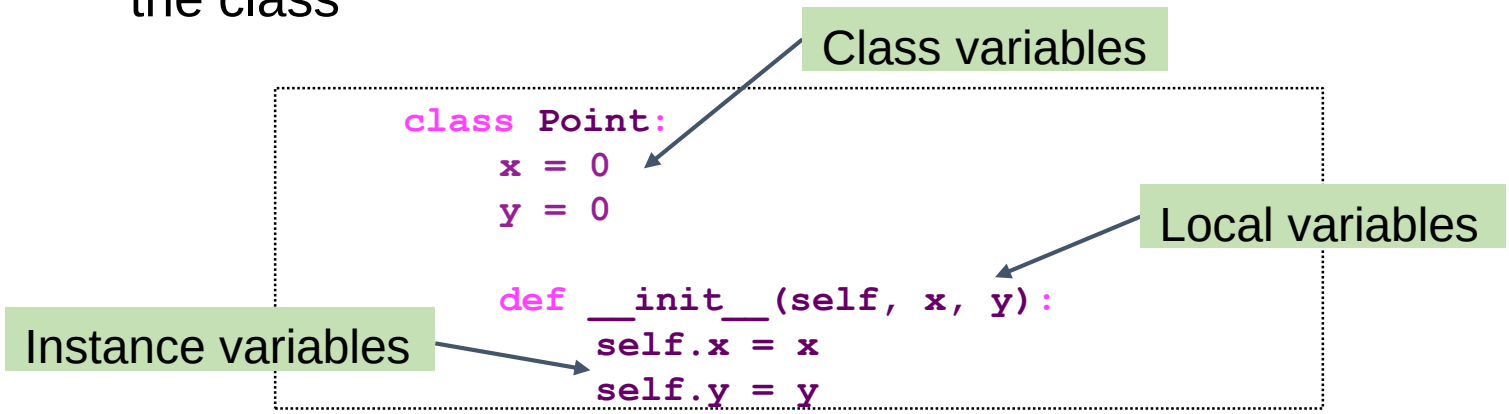
Variable Scoping and Lifetime in Function



Variable Scoping and Lifetime in Class

- Instance variables:
  - Associated to individual objects and are *unique* to each other
  - Local to the class and cannot be accessed outside of the class

- Class variables:
  - Define outside the body of any methods in a class
  - Global in scope and can be accessed both inside and outside of the class



Review Question 1

What would be the output for the given program?

```
class Point:
    x = 0
    y = 0
    def __init__(self, x, y):
        self.x = x
        self.y = y

>>> point1 = Point(1,2)
>>> print(point1.x, "and", Point.x)
```

A. 0 and 1

B. 1 and 0

C. 1 and 1

D. 0 and 0

Review Question 2

Which module are you struggling with?

A. Module 1: basic grammar (variables, basic operations, control statements like loop, if-condition...)
B. Module 2: data structure (list, tuple, set, dictionary...)
C. Module 3: decomposition (function, class)
D. Not at all

Week 6 Summary

- So far, we have discussed:
  - Classes (and methods)
  - Variable scoping and lifetime
- Next week:
  - Data structure by class: Stack & Queue
  - Summary of last six weeks

Reminder: The assignment due this Sunday (**11:55pm**).
Please come to next-week lab for **interview**.
Mid-Semester Test will be held on **16st September**.