Objects and classes

CMPT 145

Learning Objectives

After studying this chapter, a student should be able to:

- Explain the differences between Procedural and Object Oriented Programming (OOP).
- Explain the difference between a class and an object.
- Explain what attributes and methods are in terms of object oriented programming.
- Define simple classes, including data and methods, in Python.

Procedural programming

- In CMPT 141 and CMPT 145 (so far), our programs consisted of
  - data: variables, list, dictionaries.
  - computation: loops, conditionals, functions
- Procedural programming uses functions (procedures) to encapsulate (contain) algorithms.

Object Oriented concepts: Object

- An *object* consists of
  - **data** stored in the object (similar to a record defined by a record type)
  - **operations** on the data (in the form of functions)
- The data in an object are stored using variables local to the object. These variables are called attributes or fields or instance variables.
- The operations in an object are called member functions or methods or messages.
- An object is self-contained. A well-designed object contains data and has methods to operate on that data.

Object Oriented concepts: Class

- A class is like a blue-print for objects.
- An object is created from its class.
  - You can create many objects from the same class.
  - The class name is also the object's type.
- A class defines the attributes and the functions that the object will have.
  - The class doesn't usually do work; objects do work.
  - The class doesn't store attributes; the objects do.

Classes you already know about

- String (immutable)

```
1 alist = 'Jan Feb Mar Apr May'.split()
```

- List

```
1 astring = alist.append('Jun')
```

- Dictionary

```
1 addict = {'one': 1}
2 print(addict.keys())
```

A simple class

```
1 class Hero(object):
2     def __init__(self, nn, pp):
3         self.name = nn
4         self.power = pp
```

Class definitions:

- A class definition starts with the keyword `class`
- Everything in the class is indented relative to `class`
  - (rather like internal functions)
- The class name is conventionally capitalized
- The class name is followed by `(object):`
  - Looks like a function-parameter list, but it's not
  - More about this later!

Class definitions: `__init__()`

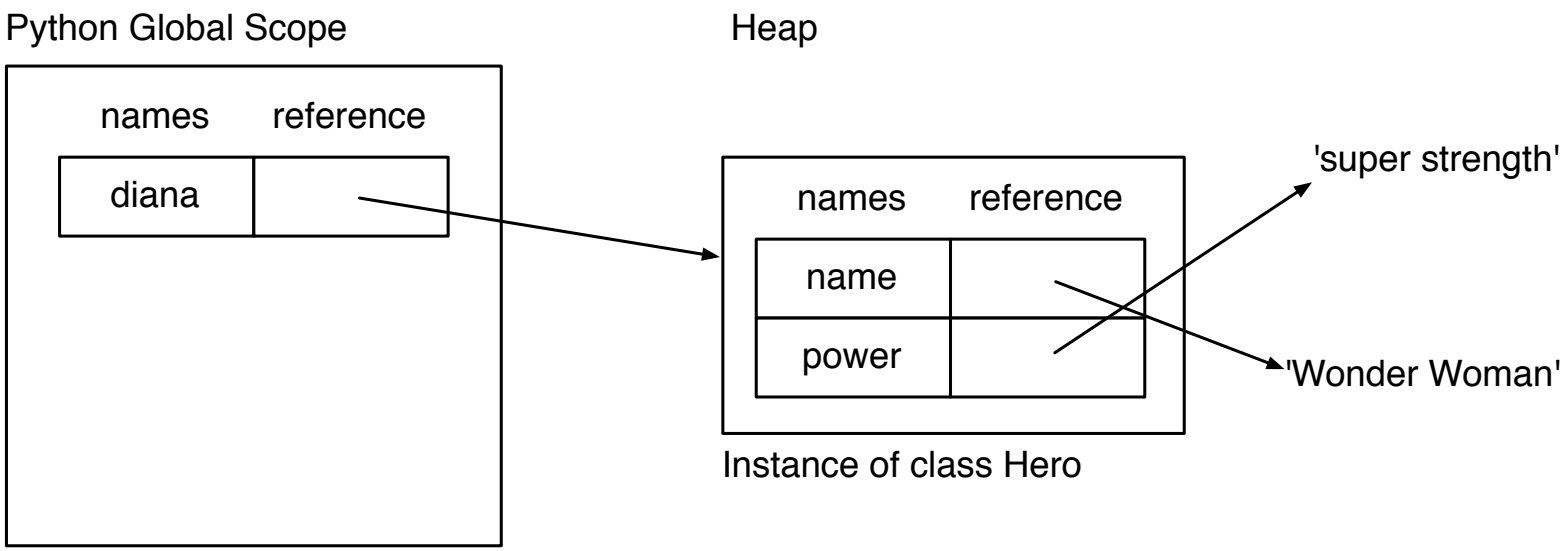
- A class definition should always have an `__init__()` method
- When an object is created, Python calls `__init__()` implicitly
- The first parameter for `__init__()` is always `self`
- `__init__()` initializes the object `self` by creating attributes using assignment statements.
- `__init__()` has no return statement

A hero is born

```
1  class Hero(object):
2      def __init__(self, nn, pp):
3          self.name = nn
4          self.power = pp
5
6  diana = Hero('Wonder Woman', 'super strength')
```

There are two attributes, `self.name` and `self.power` are created by the assignment statements.

A hero is born

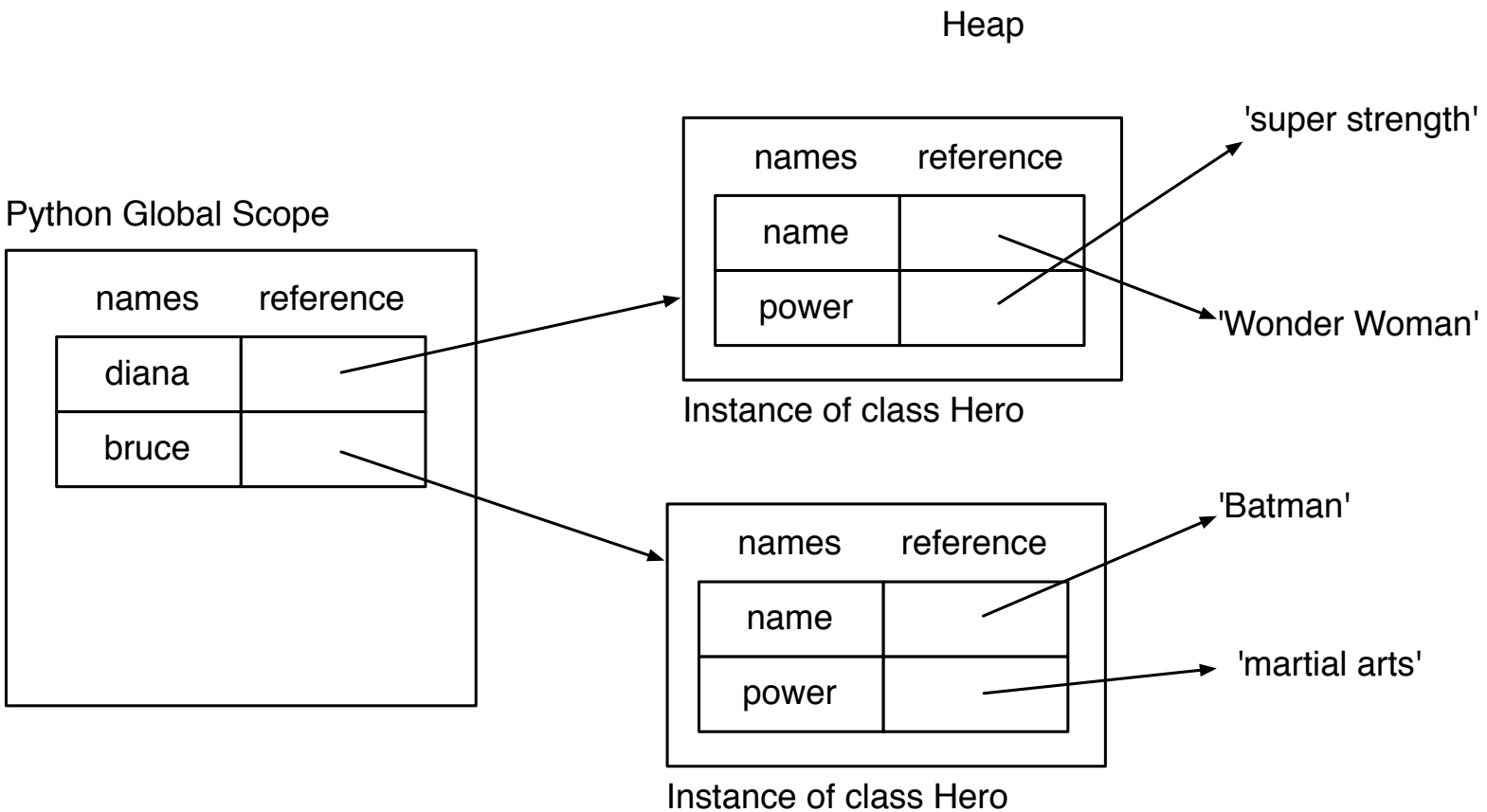


Towards a league

```
1  class Hero(object):
2      def __init__(self, nn, pp):
3          self.name = nn
4          self.power = pp
5
6  diana = Hero('Wonder Woman', 'super strength')
7  bruce = Hero('Batman', 'martial arts')
```

There are now two objects, each has two attributes. The attributes have the same names, but different values.

Towards a league



Object attributes

- The `__init__()` method should initialize attributes
- Attributes are variables local to the object `self`
- Attributes are accessed using the dot-notation, e.g., `self.name`
- Many objects can be created from the same class:
  - All the objects have the same attribute names
  - The attribute values can be different

Object methods

The class defines what objects do by defining methods:

```
1  class Hero(object):
2      def __init__(self, nn, pp):
3          self.name = nn
4          self.power = pp
5
6      def say_hello(self):
7          print('Hello, evil-doers! My name is', self.name)
8          print('My super power is', self.power)
```

- The function `say_hello()` is a method for the class Hero.
- Every method's first parameter is always `self`.
- More parameters are allowed, after `self`. All the parameters are normal function parameters.

Calling Object methods

```
1  class Hero(object):
2      def __init__(self, nn, pp):
3          self.name = nn
4          self.power = pp
5
6      def say_hello(self):
7          print('Hello, evil-doers! My name is', self.name)
8          print('My super power is', self.power)
9
10
11 diana = Hero('Wonder Woman', 'super strength')
12 bruce = Hero('Batman', 'martial arts')
13 bruce.say_hello()
14 diana.say_hello()
```

Calling a method uses the dot-notation: `var.method(args)`

`var` is a variable or expression that refers to an object.

Calling Object Methods

- The class defines what objects do by defining methods
- In a definition, a method's first parameter is always `self`
- Calling a method uses the dot-notation.
- Calling a method never gives an argument for `self`
  - We write `bruce.say_hello()`
  - Python calls the `say_hello()` method, giving `bruce` as the value of the first parameter, `self`.

Classes provide data hiding

- Our ADTs were designed to hide data behind operations.
  - E.g., the Statistics ADT.
- Classes provide extra safety for data by restricting access to attributes.
- Python does this by a convention:
  - `self.attribute1`: public. Anyone can access `attribute1`
  - `self._attribute2`: protected. Anyone can access `_attribute2` but doing so is considered ill-advised.
  - `self.__attribute3`: private. Leave `__attribute3` alone.

Access to attributes

```
1  class Hero(object):
2      def __init__(self, nn, pp, sid):
3          self.name = nn
4          self.power = pp
5          self.__secret = sid
6
7
8  bruce = Hero('Batman', 'martial arts', 'Bruce Wayne')
9  print(bruce.name)
10 print(bruce.__secret)
```

There are two public attributes, `self.name` and `self.power`
There is one private attribute, `self.__secret`.

Public attributes

- All languages allow access to public attributes.
- Public attributes can be accessed in any script.
- Class designers decide to make attributes public because:
  - Access does not put data at risk.
  - Access simplifies coding for scripts using the class.

Protected attributes

- Python leaves protected attributes public.
- Protected attributes are accessible by any script.
  - But the programmer doesn't really think you should be using them.
  - "Don't touch, but go ahead if you think you know what you're doing."
- In other languages (e.g., Java, C++), the term protected carries a bit more weight. Access to protected attributes is limited to modules in the same library.

Private attributes

- All languages prevent access to private attributes.
- In Python, trying to access a private attribute naively raises a run-time error.
- If you work hard enough, you can find a way to access private attributes in Python.
- Private attributes are used when the programmer knows you'll only mess things up.

Private attributes: getters and setters

- Making attributes protected or private allows programmers to control access
- A getter is a method that returns the value of an attribute.
  - The attribute can be public, private, or protected.
- A setter is a method that modifies the value of an attribute.
  - The attribute can be public, private, or protected.

Access advice

- For ADTs, when data should be hidden, use private
- For simple data structures, allow public if there's no chance that the encapsulated data can be messed up.
- Use private for everything else.
- Don't be optimistic. Better to protect your data than to open your code up to errors.