Introduction to Python – Part III

Hamid Zarrabi-Zadeh
Web Programming – Fall 2013
Outline

• More on Functions
  – Anonymous Functions
  – Variable Arguments
• Decorators
• Classes and Objects
  – Inheritance
  – Inner Classes
Anonymous Functions

- Anonymous functions are defined using `lambda` keyword
  - Syntax: `lambda args: expression`

```python
f = lambda x: x ** 2
# is equivalent to
def f(x):
    return x ** 2

# another example
f = lambda x, y: x + y
```
Packed Arguments

- Upon function calls, we can unpack lists and tuples with *, and unpack dictionaries with **

```python
person = ['Ali', 'Orouji']
greet(person[0], person[1])
greet(*person)

person = {
    'name': 'Ali',
    'family': 'Orouji'
}
greet(**person)
greet(name='Ali', family='Orouji')
```
Variable Arguments

- We can define variable arguments using a single * representing a tuple, or ** representing a dictionary

```python
def average(*args):
    sum = 0
    for x in args:
        sum += x
    return sum / len(args)

average(2, 3, 5)

def f(*args, **kwargs):
    print(args, kwargs)
f(2, 3, k=5, n=10)
```
Decorators
Decorators

- A **decorator** is a “function wrapper” that "decorates" the behavior of a function to perform somewhat differently than designed, or to do something in addition to its native task.

```python
@deco
def foo():
    pass

# is equivalent to
foo = deco(foo)
```
Defining Decorators

- Decorators are simply functions that take a function and return a decorated version of it.

```python
def log(func):
    def wrapped_func():
        print("-- Inside %s --" % func.__name__)
        return func()
    return wrapped_func

@log
def foo():
    pass

foo()
```
Decorators with Arguments

- Decorators can take arguments
- In this case, a “decorator-maker” takes the arguments and returns a decorator that takes foo as the function to wrap

```python
@decomaker(deco_args)
def foo():
    pass

# is equivalent to:
func = decomaker(deco_args)(foo)
```
Here is a sample decorator with arguments

```python
def div_wrap(class_name):
    def decorator(function):
        def wrapper(*args, **kwargs):
            text = function(*args, **kwargs)
            tmpl = '<div class="{0}">{1}</div>'
            return tmpl.format(class_name, text)
        return wrapper
    return decorator

@div_wrap('my_class')
def foo():
    return 'Some text!'
```
Classes and Objects
We can define a class using `class` keyword

```python
class Person:
    version = 1.2

    def __init__(self, name, phone):
        self.name = name
        self.phone = phone

    def update_phone(self, phone):
        self.phone = phone
```
Instantiation

- Instances are created by simply calling the class as if it was a function.
- This creates the object, automatically calls `__init__` with the given arguments, and then returns the newly created object back to you.

```python
a = Person('Ali', '0912-112-2030')
b = Person('Mahsa', '6616-6645')

b.update_phone('6616-6695')
```
Inheritance

• For creating a subclass, we only need to provide one or more base classes upon defining the subclass

```python
class Student(Person):
    def __init__(self, name, phone, id):
        Person.__init__(self, name, phone)
        self.id = id

a = Student('Reza', '4460-3220', '901002188')
```
Inner Classes

- We can define classes inside other classes
- The inner class is only visible to instances of the outer class

```python
class MainClass(object):
    class InnerClass:
        pass
```
References

• Python Web Development with Django
  – By Jeff Forcier, Paul Bissex, Wesley Chun

• Python 3 Documentation
  – http://docs.python.org/3/

• Python Official Website
  – http://python.org/