COMS 3101-3 Programming Languages – Python: Lecture 3

Kangkook Jee

jikk@cs.columbia.edu
Homework 2

1000778,
Zimmerman Farmers Market,
http://www.pzfarmersmarket.org,
"Lions Park, Main Street",
Zimmerman,
Sherburne,
Minnesota,
55398,
-93.5868786,
45.4407107

```
fields = line.split(',
for field in fields:
    if field.startswith('"'):
        ...
    elif fields.endswith('"'):
        ...
    else:
        ...
```
Review

• Advanced data types
  – Dictionary
  – String
  – Sets

• Function
  – Function basics
  – Positional parameters, named parameters
  – Default parameters
  – Scoping issues
  – Closures
Agenda

• Advanced Functions
  – Functional programming with Map, Reduce, and Filter
• Objected Oriented Programming
• Modules and Packages
• If time permits
  – Basic modules os, sys
ADVANCED FUNCTIONS

map, reduce, and filter
Map, Reduce, and Filter

- Functional programming elements in action
  - Core constructs in parallel programming paradigm
  - We’re not covering `lambda` from this lecture
- Function is a first-class citizen in Python
  - Provided as a first parameter
- Smart way to iterate over sequence data structure
  - More readable and concise syntax
    - at least for some
Map

- We want to apply a function for all elements of sequence data type

\[
\text{map(func, seq1, [seq2, ... seqN])}
\]

- `func` need to be a callable object (i.e., function)
- `func` take \( N \) arguments where \( N \) is number sequence data types
- `seq1, seq2 ... seqN` need to be in the same length

```python
def to_str(lst):
    ret = []
    for i in lst:
        ret.append(str(i))
    return ret
```

- List comprehension

\[
\text{[str(i) for i in lst]}
\]

- Map

\[
\text{map(str, lst)}
\]
Map: Adding One for All Elements

```python
def inc_one(x):
    return x + 1

def inc_one_list(lst):
    ret = []
    for i in lst:
        ret.append(inc_one(i))
    return ret
```

```python
map(inc_one, range(10))
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

```python
inc_one_list(range(10))
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

- Applying add_one() to all elements
Map: Adding X for All Elements

map(func, seq1, [seq2, ...])

- Closure: inc_x() preserved state defined from inc_some()

```python
def inc_some(x):
    def inc_x(y):
        return x + y
    return add_x

def inc_some_list(lst):
    ret = []
    inc_two = inc_some(2)
    for i in lst:
        ret.append(inc_two(i))
    return ret

>>> map(inc_some(2), range(10))
[2,3,4,5,6,7,8,9,10,11]

inc_some_list(range(10))
[2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
```
Map: Combining Two (or more) Sequence

```
map(func, seq1, [seq2, ...])
```

- `add_two()` takes two parameters
- if `func` is `None`, the identity function is assumed

```python
def add_two(x, y):
    return x + y

def add_two_lists(lst0, lst1):
    ret = []
    add_two = add_some(2)
    for i, j in zip(lst0, lst1):
        ret.append(i + j)
    return ret

add_two_list(range(5), range(5))
[0, 2, 4, 6, 8]
```

```python
def add_two(x, y):
    return x + y

>>> map(add_two, range(5), \
    range(5))
[0, 2, 4, 6, 8]
```
Filter

**Filter**

\[
\text{filter}(\text{func, seq})
\]

- Extracts each element \( x \) for which \( \text{func}(x) \) returns True
- \( \text{func} \) takes a single parameter and returns boolean type (True, False)

```python
def is_neg(x):
    return x < 0
```

```python
def filter_neg(lst):
    ret = []
    for i in lst:
        if is_neg(i):
            ret.append(i)
    return ret
```

```python
>>> filter(is_neg, range(-5, 5))
[-5, -4, -3, -2, -1]
```

```python
filter_neg(range(-5, 5))
[-5, -4, -3, -2, -1]
```
reduce

\[ \text{reduce}(\text{func}, \text{seq}) \]

- Reduce a sequence data type(\text{seq}) to a single value by combining elements via \text{func}
- \text{func} takes two parameter and returns a single value

```
add(x, y):
    return x + y
```

```
def sum(lst):
    ret = None
    for i in lst:
        if ret:
            ret = add(ret, i)
        else:
            ret = i
    return ret
```

```
>>> reduce(add, range(-4, 5))
0
```

```
sum(range(-4, 5))
0
```
OBJECT ORIENTED PYTHON
Object Oriented Programming in Python

• Object Oriented Programming (OOP) is at the core of Python
  – Everything is an object!
  – Operations are methods on objects
  – Modularizations

• We have seen examples of objects already
  – Objects of built-in data types (int, str, list, dict ...)
  – Functions

• Class allow us to create our own type
int() class

```python
>>> a = int(1)
>>> type(a)
<type 'int'>
>>> a
1
>>> dir(a)
['__abs__', '__add__', '__and__', '__class__', '__cmp__', ...
'denominator', 'imag', 'numerator', 'real']
>>> a.__add__(1)
2
>>> b = myint(2)
>>> b + a
addition: 2 + 1
3
```

Class: A Custom Type

Built-in Types Definitions

- object
  - int
    - value=0
    - __add__()
    - __sub__()
    - __mul__()
    - __div__()
    - ...
  - str
    - char[] = {'a', 'b', 'c'}
    - __add__()
    - __isalpha__()
    - ...
  - list
    - obj[] = {...}
    - append()
    - remove()
    - pop()
    - ...

... sequence

- __contains__()  
- __add__()   
- __mul__()   
- __div__()   
...

Custom Types Definitions

- MyInt
  - value=0
  - __add__()
  - __sub__()
  - __mul__()
  - __div__()
...

Instantiated Objects

- 0, 1, 2, ...
- 'abc', 'hello', ...
- [], [1, 2, 3], ['test', 'list']
- 0, 1, 2, ...

Instantiated Objects
Objects, Attributes, Methods, Classes

• Classes
  – User-defined types of objects (including their methods, attributes, relations to other objects)
  – Can be instantiated into an object / is a ‘blueprint’ that describes how to build an object

```
Knights can eat sleep, have a favorite color, and a title.
```

• Object: Grouping of state(attributes) and behavior (methods) into a functional ‘package’

```
l = Knight()
```

• Attributes: data fields of the object for state maintenance

```
l.name = “Launcelot, l.title = “The brave” ...
```

• Methods: functions that belong to the object and can access and manipulate the object’s data. All methods are attribute

```
l.eat(food), l.sleep() ...
```
Class Definition with `class`

- Class definitions contains
  - *methods*: functions defined in the class’ scope
  - *class attributes*: variables defined in the class’s scope
  - *docstring*: documentation that explains the class

```python
class Knight(object):
    """A knight with two legs, who can eat food."
    """
    legs = 2  # class attribute

    def __init__(self):
        self.stomach = []

    def eat(self, food):
        self.stomach.append(food)
        print('Yummy!')
```

- `class` definition with inheritance and docstring
- Class attribute ‘leg’
- Constructor method ‘__init__()’ defines instance attribute ‘self.stomach’
- Instance method ‘eat()’
Class Definition with \texttt{class}

- Class definitions contains
  - \textit{methods}: functions defined in the class’ scope
  - \textit{class attributes}: variables defined in the class’s scope
  - \textit{docstring}: documentation that explains the class

\begin{verbatim}
class Knight(object):
  
  """A knight with two legs, who can eat food."
  """
  legs = 2  # class attribute

  def \_\_init\_(self):
    self.stomach = []

  def eat(self, food):
    self.stomach.append(food)
    print(‘Yummy!’)
\end{verbatim}

- Classes are objects too.
  - Methods and attributes are attributes of the class object

\begin{verbatim}
>>> Knight.legs
2
>>> Knight.eat
<unbound method Knight.eat>
\end{verbatim}
Instantiating a Class to an Instance Object

```python
class Knight(object):
    """A knight with two legs, who can eat food."
    """
    legs = 2  # class attribute

    def __init__(self):
        print "init. Knight"
        self.stomach = []

    def eat(self, food):
        self.stomach.append(food)
        print('Yummy!')

>>> k = Knight()  # invoke Knight.__init__(self)
init. Knight
>>> k
<_main__.Knight object at 0x107709f90>
>>> type(k)
<class '__main__.Knight'>
```

- Functions are instantiated into instance objects by calling a class object
Calling Bound Methods on Instance Objects

```python
class Knight(object):
    """A knight with two legs, who can eat food. """
    legs = 2  # class attribute

    def __init__(self):
        print "init. Knight"
        self.stomach = []

    def eat(self, food):
        self.stomach.append(food)
        print('Yummy!')

>>> k = Knight()  # invoke Knight.__init__(self)
>>> k.eat("cheese")
Yummy!
>>> k.stomach
['cheese']
```

- Functions are instantiated into instance objects by calling a class object
- The first parameter in a method definition (‘self’) is bound to the instance object when a bound method is called
class Knight(object):
    """A knight with two legs, who can eat food."
    """
    legs = 2  # class attribute

    def __init__(self, name):
        self.stomach = []
        self.name = name

    def eat(self, food):
        self.stomach.append(food)
        print('Yummy!')

• The special method __init__() is called when an instance is created
  – ClassName(x, y) corresponds to ClassName.__init__(self, x, y)

• Main purpose:
  – Initialize instance object referred by self
  – sets up attributes of the instance

>>> k = Knight('kangkook')
>>> k.name
"""kangkook"""
self

- **self** represents instantiated object
  - Become a reference point to runtime instances
  - Created as we call constructor method `__init__()`
- **self** is an *automatically* passed as a first argument when *instances* call methods
- Instance methods must specify *self* as their first parameter

```python
>>> k.eat('bacon')
Yummy!
>>> Knight.eat(k, 'toast')
Yummy!
>>> k.stomach
['bacon', 'toast']
```
‘int’ Type

class int(object):
    def __init__(self, x=0):
        ...
    def __init__(self, x, base=10):
        ...
    def __add__(self, other):
        ...
    def __sub__(self, other):
        ...  

>>> x = int()
>>> x
0
>>> y = int('10', base=16)
>>> x.__add__(y)
16

• Two constructors for ‘int’ and ‘str’ respectively
• int.__add__() methods overrides ‘+’ operator
  – x.__add__(y) \iff x + y
‘str’ Type

```python
class str(basestring):
    def __init__(self, obj=''): ...
    def __add__(self, other): ...
    def __contains__(self, other): ...
    def __format__(self, format_spec) ...

>>> s = str(12345)
>>> s
'12345'
>>> s.__contains__('1')
True
```

- A constructors: a parameter ‘obj’ can be any type that implements `__str__()`
- int.__contains__() methods overrides ‘in’ operator
  - s.__contains__(x) <=> x in s
Class vs. Instance Attributes

```python
class Knight0(object):
    # class attribute
    inst_count = 0

    def __init__(self):
        Knight0.inst_count += 1

>>> c1 = Knight0()
>>> c2 = Knight0()
>>> c2.inst_count
2
>>> Knight0.inst_count
2

class Knight1(object):
    # class attribute
    inst_count = 0

    def __init__(self):
        self.inst_count += 1

>>> c1 = Knight1()
>>> c2 = Knight1()
>>> c2.inst_count
1
>>> Knight1.inst_count
0
```

- Class variables are accessed with class name (not with self)
- Class attributes are visible from instances
- Re-binding attribute names in an instance creates a new instance attribute that hides the class attribute
Inheritance

• Classes inherits from one or more base classes
  – Multiple inheritance allowed
• Look up methods and class attributes in base classes if not found in class

class Knight(Warrior, TitleBearer):
    
    def __init__(self, name):
        ...
    
    def go_on_quest(self, quest):
        ...

>>> k1 = Knight(“Galahad”)
>>> k2 = Knight(“Robin”)
>>> k1.fight(k2)
>>> k2.eat(“bacon”)
Multiple Inheritance – Method Resolution Order

• Problem: Which `eat` method to use?
• User first found according to method resolution order

```python
class Knight(Warrior, TitleBearer):
    def __init__(self, name):
        ...
    def go_on_quest(self, quest):
        ...
```

```python
>>> k1 = Knight(“Galahad”)
>>> k2 = Knight(“Robin”)
>>> k1.fight(k2)
>>> k2.eat(“bacon”)
```
Sub-classing ‘object’

• Base class for all class defined
  – All should begin from object
• Implements basic methods

```python
>>> dir(object)
['__class__', '__delattr__', '__doc__', '__format__',
 '__getattribute__', '__hash__', '__init__', '__new__',
 '__reduce__', '__reduce_ex__', '__repr__', '__setattr__',
 '__sizeof__', '__str__', '__subclasshook__']

>>> obj = object()
<object object at 0x10191f090>
```
class FlipDict(dict):
    """A dictionary that can be inverted
    """
    def flip(self):
        """Return a dictionary of values to sets of keys
        """
        res = {}
        for k in self:
            v = self[k]
            if not v in res:
                res[v] = set()
                res[v].add(k)
        return res

>>> import flipdict
>>> x = flipdict.FlipDict([(1,'a'), (2, 'b'), (3,'a')])
>>> x
{1: 'a', 2: 'b', 3: 'a'}
>>> x.flip()
{'a': set([1, 3]), 'b': set([2])}

Polymorphism

```python
class Shape(object):
    def perimeter(self):
        """abstract method"""
        return

class Square(Shape):
    def perimeter(self):
        return self.side ** 2

class Circle(Shape):
    def perimeter(self):
        return self.radius * \ 
        2 * math.pi

• Inheritance allows to override methods of base classes in different ways

def get_perimeter(shape):
    """method to get perimeter for shape objects"""
    if isinstance(shape, Shape):
        return shape.getPerimeter()
    else:
        # Error handling follows
```
Class Type Checking with isinstance()

```python
>>> s = Square(4)
>>> type(s)
<class 'shape.Square'>
>>> isinstance(s, Shape)
True
>>> isinstance(s, Square)
True
>>> isinstance(s, Circle)
False
```

- Two ways of testing object types: built-in functions of `type()`, `isinstance()`
- `type(object)`: returns the type of the object
- `isinstance(object, class)`:  
  - if the object is an instance of the class, or of a subclass thereof
  - recommended for testing
Calling Base Class Implementations of Overloaded Methods

- Sometimes we want to call the base class version of a method
- This is often the case for `__init__`
- Use unbound method attribute the base class

```python
class Shape(object):
    def __init__(self, dimension):
        self.dimension = dimension
        self.name = name

...

class Square(Shape):
    def __init__(self, dimension, side):
        self.side = side
        Shape.__init__(self, dimension)

>>> s = Square('2d', 4)
>>> s.dimension
'2d'
```
Polymorphism: Duck Typing

• Python is dynamically typed. Any variable can refer to any object
• Explicit type checking (\texttt{isinstance}) at runtime is considered bad style
• Instead use ‘duck typing’! (plus error handling)

Duck Typing

“When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck.”

• As long as an object implements functionality, its type does not matter
• Example: Equality (==), convert to string(str())
def get_perimeter_dt(shape):
    
    duck-typed get_perimeter()
    
    try:
        return shape.perimeter()
    except TypeError:
        print("the object does not support perimeter()")
        return 0

• isinstance() removed
  – By assuming shape parameter object support ‘perimeter()’
• We will learn about exception handling (try ... except) from the next class
Special Methods (1)

- __init__(self) is called when an instance is created
- __str__(self) returns a string representation of the object
- __repr__(self) returns the ‘official’ string representation

```python
class FarmersMarket(object):
    def __init__(self, name, state, town, zipc):
        self.name = name
        self.state, self.town, self.zipc = state, town, zipc
    def __str__(self):
        return '{0} 
{1}, {2} {3}'.format(self.name, self.town, 
        self.state, self.zipc)
    def __repr__(self):
        return 'FarmersMkt:<{0}:{1},{2} {3}>'.format(self.name, 
        self.town, self.state, self.zipc)
```

```
>> mkt = farmers.FarmersMarket('CU Greenmarket', 'New York', 'NY', 10027)
>>> print mkt  # str(mkt)
CU Greenmarket
NY, New York 10027
> mkt
FarmersMkt:<CU Greenmarket:NY,New York 10027>
```

Special Methods (2) - Comparisons

- `__eq__`(self, other) used for `==` comparisons
- `__lt__`(self, other) used for `<` comparisons
- `__le__`, `__gt__`, `__ge__`, `__ne__`

```python
class Shape(object):
    def __eq__(self, other):
        return self.area() == other.area()
    def __ne__(self, other):
        return not self.__eq__(other)

class Rectangle(Shape):
    def __init__(self, l, w):
        self.l, self.w = l, w
    def area(self):
        return self.l * self.w

>>> Rectangle(2,3) == Rectangle(1,6)
True
# equivalent by operator overloading
>>> Rectangle(2,3).__eq__(Rectangle(1,6))
True
```

Special Methods (2) - Comparisons

- If none of the previous comparisons operators are defined, __cmp__(self, other) is called
  - Return 0, if self and other are equal
  - negative integer, if self < other
  - positive integer if self > other

```python
class Shape(object):
    def __cmp__(self, other):
        return self.area() - other.area()

class Circle(Shape):
    def __init__(self, r):
        self.r = r
    def area(self):
        return math.pi * self.r ** 2

>>> Circle(1) < Rectangle(4, 2)
True
```
Special Methods (3) - Arithmetic

• `__add__`(self, other) used for + operations
  - \( x + y \Leftrightarrow x.__add__(y) \)

• `__iadd__`(self, other) used for += operations

• `__sub__`, `__mul__`, `__div__`, `__radd__`, `__rsub__` ...

```python
class Complex(object):
    def __init__(self, x, y):
        self.x, self.y = x, y
    def __add__(self, other):
        return Complex(self.x + other.x, self.y + other.y)
    def __iadd__(self, other):
        self.x += other.x
        self.y += other.y
        return self

>>> c1 = Complex(1, 2)
>>> temp = c1
>>> c2 = Complex(3, 4)
>>> c1 += c2
>>> temp.x, temp.y
(4, 6)
```

MODULES AND PACKAGES
Modules

• Module is another programming abstract that
  – Defines independent grouping of code and data
  – Contains multiple variable / function / class definitions in it

• A typical Python program consists of several source files, each correspond to a module
  – A module can include other module: can cause circular import problem

• Packages are collection of modules
  – Use ‘import’ statement to load modules or packages
Structure of a Module File

- A module corresponds to any Python source file
- The module ‘name’ is typically in file ‘name.py’
- Can contain a docstring (string in first non-empty line)

```python
""" A module to illustrate modules.
"""

class A(object):
    def __init__(self, *args):
        self.args = args

def quadruple(x):
    return x**4

x = 42

print("This is an example module.")
```

Importing and Using Modules

```
import modulename [as newname]
```

- Imports a module and creates a module objects
- All statements are executed upon import
- All defined variables/classes/functions become attributes of the module

```python
>>> import sample_module as sm
>>> sm.x
42
>>> a = sm.A(1,2,3)
>>> a
```
Importing Specific Attributes of a Module

```python
from modulename import attr [as newname]
```

- loads the module and make `attr` (a class, function, variable ..) available in the namespace of the importing modules

```python
>>> from sample_module import A
This is an example module.
>>> a = A(1,2,3)
>>> testmodule
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'testmodule' is not defined
```

- Can also import all attributes (considered bad style!)

```python
>>> x = 100
>>> from sample_module import *
>>> x
42
```
Executing Module as a Script

- Problem: Modules often contain some test code that we do not want to run every time it is imported
- A special variable ‘__name__’
  - contains module name when imported
  - contains ‘__main__’ when it is executed as a script

```python
... 
print("This is an example module. {0}".format(__name__))

if __name__ == "__main__":
    a = A(1, 2, 3)
    print("script mode")
```

script execution
```
dhcp102:Desktop jikk$ python sample_module.py
This is an example module. __main__
script mode
```
Packages

- Packages group modules
  - Packages contains modules as attributes
  - Packages therefore span trees of modules
- A package corresponds to a directory
  - coms3101.lec3.sample_module indicates coms3101/lec3/sample_module.py
- Package directories must contain a file __init__.py
  - __init__.py contains package initialization code

```python
>>> import coms3101.lec3.sample_module
__init__: coms3101 from coms3101/__init__.py
__init__: coms3101.lec3 from coms3101/lec3/__init__.pyc
This is an example module.
coms3101.lec3.sample_module
```
Backup Slides
stdin and stdout

- Can access terminal input (sys.stdin) and terminal output (sys.stdout) as file object
- These objects are defined globally in the module ‘sys’
  - ‘sys’ is loaded with import statement

```python
>>> import sys
>>> sys.stdout.write('Hello world!
')
Hello world!
>>> sys.stdin.read(4)
COMS3101-3
'COMS'
```
File Operation with ‘os’

• ‘os’ module defines interfaces that enable interactions with operating systems
  – Most frequently used component of standard library
  – Implements majority subset of OS system call API

```python
>>> import os
>>> os.system('date')  # OS specific command
Wed Sep 9 22:16:59 EDT 2013
0

>>> os.path.isdir("/tmp")  # some folder
True
```

• ‘os.path’ sub-module defines interfaces for filename manipulation
os.path Module – manipulate pathnames

- `os.path.abspath(path)` – Returns the absolute pathname for a relative path

```python
>>> os.path.abspath('python')
'/opt/local/bin/python'
```

- `os.path.basename(path)` – Returns the absolute pathname for a relative path

```python
>>> os.path.abspath('python')
'/opt/local/bin/python'
```

- `os.path.getsize(path)` – Returns the size of path in byte

```python
>>> os.path.getsize("python")
13404
```

- `os.path.isfile(path)` – Returns True if the path points to a file
- `os.path.isdir(path)` – Returns True if the path points to a directory
os Module – list, walk content of a directory

- os.listdir(path) lists files in a directory

  ```python
  >>> os.listdir("/tmp")
  ['.font-unix', '.ICE-unix', ... , android-jikk']
  ```

- os.walk(path) returns generator object to traverse sub-directories in depth-first fashion

  ```python
  >>> w = os.walk('/tmp')
  >>> loc = w.next()
  >>> while w:
  ...     print loc
  ...     loc = w.next()
  ```