Software Development
Python (Part B)

Davide Balzarotti

Eurecom

List Comprehension

- It is a short way to construct a list based on the content of other existing lists
  - Efficient
  - Elegant
  - Concise
- List comprehensions consist of an expression followed by a `for`

```
[word.upper() for word in "python magic".split()]
```

List Comprehension

- It is a short way to construct a list based on the content of other existing lists
  - Efficient
  - Elegant
  - Concise
- List comprehensions consist of an expression followed by a `for`

```
[word.upper() for word in "python magic".split()]
```

- It can contains more than one `for` instruction

```
[x*y for x in [2, 4, 6] for y in [4, 3, -9]]
```

List Comprehension

- It is a short way to construct a list based on the content of other existing lists
  - Efficient
  - Elegant
  - Concise
- List comprehensions consist of an expression followed by a `for`

```
[word.upper() for word in "python magic".split()]
```

- It can contains more than one `for` instruction

```
[x*y for x in [2, 4, 6] for y in [4, 3, -9]]
```

- It can contains `if` clauses as well

```
[x for x in vector if (10 < x <20)]
```

List Comprehension

- Each element can also be a list

```
words = ['The', 'quick', 'brown', 'Fox', 'jumps',
         'over', 'the', 'lazy', 'Dog']

wlen = [[w.lower(), len(w)] for w in words]

   [['the', 3], ['quick', 5], ['brown', 5],
   ['fox', 3], ['jumps', 5], ['over', 4],
   ['the', 3], ['lazy', 4], ['dog', 3]]
```

- Or a condition

```
[x > 5 for x in range(10)]

[False, False, False, False, False,
   False, True, True, True, True]
```

Builtin Functions for Iterable Objects

- `any(iterable)` – return True if any element of the iterable is true

```
if any([x < 0 for x in numbers]):
```

- `all(iterable)` – return True if all elements of the iterable are true (or if the iterable is empty)

```
if all([x>0 for x in numbers]):
```

- `len(s)` – return the number of elements of an object (list, tuple, dictionary..)
- `max(iterable)` – return the largest item of a non-empty iterable
- `min(iterable)` – return the smallest item of a non-empty iterable
- `sum(iterable)` – return the sum of the items of an iterable from left to right

Anonymous Functions

lambda parameters: expression

- The Lambda construct can be used to define on the fly nameless, short functions
- The body of an anonymous function is syntactically restricted to a single expression

```
>>> f = lambda x,y: x+y
>>> f(3,6)
9
```

- Lambda functions are commonly used to create a temporary functions to pass as parameters to other functions

```
>>> l = [[1,3], [7,2], [15,1], [2,4], [3,6], [9,5]]
>>> l.sort(key=lambda x: x[1])
>>> print l
[[15,1], [7,2], [1,3], [2,4], [9,5], [3,6]]
```

Functional Programming

- `filter(function, sequence)`
  - Returns a list containing the elements of sequence for which function(element) is `True`

```
filter(lambda x: x%2==0, range(20))
                ==
 [x for x in range(20) if x%2==0]
```

Functional Programming

- `filter(function, sequence)`
  - Returns a list containing the elements of sequence for which function(element) is `True`

```
filter(lambda x: x%2==0, range(20))
                ==
[x for x in range(20) if x%2==0]
```

- `map(function, sequence)`
  - Returns a list obtained by applying function to each element in sequence

```
map(lambda x: x**2, range[10])
            ==
  [x**2 for x in range(10)]
```

Functional Programming

- `filter(function, sequence)`
  - Returns a list containing the elements of sequence for which function(element) is `True`

```
filter(lambda x: x%2==0, range(20))
                ==
[x for x in range(20) if x%2==0]
```

- `map(function, sequence)`
  - Returns a list obtained by applying function to each element in sequence

```
map(lambda x: x**2, range[10])
             ==
  [x**2 for x in range(10)]
```

- `reduce(function, sequence)`
  - Returns a single value constructed by calling the binary `function` on the first two items of sequence, then on the result and the next item, and so on

```
reduce(list.__add__, [[1, 2, 3], [4, 5], [6, 7, 8]])
```

Functions

Functions are objects
- they may be passed as arguments to other functions
- they may be assigned dynamically at runtime
- they may return other functions as their result
- they may be keys in a dictionary or items in a sequence

```
def function-name(parameters):
      """ documentation string """
      statement1
      statement2
      return value
```

Optional docstring
Accessible through
function-name.__doc__

No value: `return, return None`
Single value: `return expression`
Multiple values: `return value1, value2,..`

Example

```
>>> def average(par1, par2):
...    """ Calculate the average """
   ...    return (par1+par2)*1.0/2
>>> average
   <function average at 0xb7eacd14>
>>> average(2,3)
2.5
>>> dir(average)
['__call__', '__class__', '__closure__', '__code__', '__defaults__',
   '__delattr__', '__dict__', '__doc__', '__format__', '__get__',
   '__getattribute__', '__globals__', '__hash__', '__init__',
   '__module__', '__name__', '__new__', '__reduce__',
   '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__',
   '__str__', '__subclasshook__', 'func_closure', 'func_code',
   'func_defaults', 'func_dict', 'func_doc', 'func_globals',
   'func_name']
>>> print a.__doc__
Calculate the average
```

Function Scope

- By default, any variable in a function body is a local variable of the function
  - It is possible to access (use) global variable from inside a function but...
  - If a variable with the same name is defined in the function, it hides the corresponding global variable
- If a function want to rebind a global variable (usually because it wants to assign to the variable name), it has to declare it first:

```
def f():
    global x
    x = 5
```

- This is not elegant and in general it must be avoided

Nesting Functions

- It is possible to define a function inside another function
  - The local function can then be returned or passed to other functions
  - Free variables used in the nested function can access (but not rebind) the local variables of the outer function
- Examples:

```
def percent2(a, b, c):
    def pc(x):
      return (x*100.0) / (a+b+c)
    print "Percentages are:", pc(a), pc(b), pc(c)
```

```
def make_adder(step):
    def add(value):
       return value+step
    return add
```

Arguments

- Are parameters passed by value or by reference?
  - It's quite complicated.. and again it boils down to the difference between mutable and immutable objects
- In practice, python always pass a reference by value
  - Not so clear?

Arguments

- Are parameters passed by value or by reference?
  - It's quite complicated.. and again it boils down to the difference between mutable and immutable objects
- In practice, python always pass a reference by value
  - Not so clear?
- Let's make it easier: Parameters are passed by assignment
  - Think of each parameter as being assigned (with the python assignment semantic we saw last time) when entering the function
  - The function cannot re-bind the parameters but it can modify them if they are mutable

Arguments

- There is no function overloading in Python
  - Two functions cannot have the same name
  - Seriously, not even if they have different parameters
- Invocation:
  - With no parameters: `function()`
  - With parameters: `function(v1, v2, ... vn)`
  - With parameters taken from a list:

```
plist = [v1, v2, .. vn]
function(*plist)
```

  - With parameters taken from a dictionary:

```
pdict = {p1:v1, p2:v2, .. pn:vn]
function(**pdict)
```

Keywords and Default Parameters

- During a function invocation, the name of the parameters can be used to explicitly set their value
- A function definition can contains optional parameters
  - They must have default values
  - They must be placed after the mandatory ones

```
>>> def p(what, header="@ "):
...     print header + str(what)

>>> p("hello")

@ hello

>>> p("hello", "% ")

% hello

>>> p(header = "- ", what="hello")

- hello
```

More on Default Parameters

- Note that each default value gets computed when the `def` statement evaluates, not when the function is called
  - The object that represents the default value is always the same whenever the caller does not supply a corresponding argument
  - Side-effect are possible if the default value is mutable

```
def f(x, y=[]):
    y.append(x)
    return y


>>> f(4)
[4]
>>> f(5)
[4,5]
```

More on Default Parameters

- Note that each default value gets computed when the `def` statement evaluates, not when the function is called
  - The object that represents the default value is always the same whenever the caller does not supply a corresponding argument
  - Side-effect are possible if the default value is mutable

```
def f(x, y=[]):
    y.append(x)
    return y


>>> f(4)
[4]
>>> f(5)
[4,5]
```

```
def f(x, y=None):
    if y is None:
        y = []
    y.append(x)
    return y

>>> f(4)
[4]
>>> f(5)
[5]
```

Arbitrary Arguments

- Arbitrary positional arguments
  - Initialized as a tuple containing all the positional arguments supplied (no keyword arguments are allowed)

```
def fun(*args):
   print args

>>> fun(1,2,3)
(1,2,3)

>>> fun(1,2,x=3)
TypeError: fun() got an keyword argument 'x'
```

- Arbitrary keyword arguments

```
def fun(normal, *args, **kargs):
  print normal
  print args
  print kargs

>>> fun(1,2,3,4,x="red", y="blue")
1
(2,3,4)
{'x':'red', 'y':'blue'}
```

Generators

- Generators are a simple and powerful tool to create iterators
- They are written like regular functions but use the `yield` statement instead of the `return` statement
  - When the `next()` method is called on the generator, the execution resumes from the instruction following the yield
  - Local variables and execution state are automatically saved and restored between calls

```
def gen():
    yield 1
    yield 2

>>> g = gen()
>>> g.next()
1
>>> g.next()
2
>>> g.next()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

Generators

- Generators are a simple and powerful tool to create iterators
- They are written like regular functions but use the `yield` statement instead of the `return` statement
  - When the `next()` method is called on the generator, the execution resumes from the instruction following the yield
  - Local variables and execution state are automatically saved and restored between calls

```
def gen():
    yield 1
    yield 2

>>> for x in gen():
...     print x

1
2

>>>
```

Generators

- Generators are a simple and powerful tool for creating iterators
- They are written like regular functions but use the `yield` statement instead of the `return` statement
  - When the `next()` method is called on the generator, the execution resumes from the instruction following the yield
  - Local variables and execution state are automatically saved and restored between call

```
def dfs(node):
    """ Depth-first exploration """
    if node:
        for x in dfs(node.left):
            yield x
        yield node.label
        for x in dfs(node.right):
            yield x
```

Generator Expressions

- It has the same exact syntax of a list comprehension expression but it is surrounded by normal parenthesis () instead of square brackets []

```
(x**2 for x in range(100))
```

- The difference is that a generator expression returns an iterator that computes the values as necessary
  - Useful when the list is very long and you don't want to pre-compute it all at the beginning

Decorators

```
@decorator1
@decorator2
def f():
    pass
```

It is somehow equivalent to

```
f = decorator1(decorator2(f))
```

- Decorators are functions that wrap another function
- Decorators alter the functionality of a function or method without having to directly use subclasses or change the source code of the function being decorated
- In the example, the function `f()` is compiled and the resulting function object is passed to the `decorator2` code, which does something to produce a callable object that is then substituted for the original f(). The same process is then repeated for `decorator1`

Decorators

```
@return_string
def add(x,y):
    return x+y

>>> add(5,6)
'11'
```

Decorators

```
def return_string(f):
    def wrapper(*params):
            return str(f(*params))
    return wrapper

@return_string
def add(x,y):
    return x+y

>>> add(5,6)
'11'
```

Classes

Define a new class Foo

```
class Foo:
    """A simple example class"""
    i = 12345
    def hello(self):
        return 'Hello world'
```

Each function definition inside the class scope defines a method

The first parameter of a method must be a reference to the current instance of the class (by convention it is always called `self`)

Classes

```
class Foo:
    """A simple example class"""
    i = 12345
    def hello(self):
        return 'Hello world'
```

```
>>> print Foo.i
12345

>>> f = Foo()
>>> print f.hello()
Hello world

>>> f.value = 2
>>> del f.value
```

Foo is a **class** object

f is an **instance** object

Object attributes can be added and removed at any time

During a method invocation, the object is automatically passed as a first parameter

Classes

- Classes have five predefined attributes:

`__dict__`	dictionary	R/W	Class name space
`__name__`	string	RO	Name of the class
`__bases__`	tuple of classes	RO	Parents classes
`__doc__`	string	R/W	Class docstring
`__module__`	string	R/W	Name of the module containing the class

- Instances also have two predefined attributes

`__dict__`	dictionary	R/W	Instance name space
`__class__`	class	R/W	The class of the instance

- When an attribute of an instance is referenced via the dot operator, Python first checks the instance name space, and then, if the attribute is not found, it checks the class's name space

Everything is Public

- All class members are public
  - There is nothing in Python that can be used to enforce data hiding. It is all based upon convention
  - Attributes that starts with _ are private and they are not supposed (by convention) to be used or called from outside the class
  - If an attribute starts with __ (two underscores at the beginning but **no** underscores at the end) is textually replaced with `_classname__attribute`

```
class Bar:
    _x  = 2
    __y = 3
    def method(self):
        print Bar.__y

>>> b = Bar()
>>> dir(b)
[__doc__, __module__, _x, _Bar__y]
>>> b.method()
3
```

Data vs. Class Attributes

- Data attributes
  - Variable owned by a particular instance of a class
  - Each instance has its own value for it
  - Stored in the instance namespace
- Class attributes (static fields)
  - Owned by the class as a whole
  - Defined within a class definition and outside of any method
  - All class instances share the same value for it
  - Stored in the class namespace
  - (don't use them if it is not strictly required)

Easy to Get Confused

```
>>> class C:
...     dangerous = 2

>>> c1 = C()
>>> c2 = C()

>>> print c1.dangerous
2

>>> c1.dangerous = 3
>>> print c1.dangerous
3
>>> print c2.dangerous
```

Class attribute

Easy to Get Confused

```
>>> class C:
...     dangerous = 2

>>> c1 = C()
>>> c2 = C()

>>> print c1.dangerous
2

>>> c1.dangerous = 3
>>> print c1.dangerous
3
>>> print c2.dangerous
2

>>> del c1.dangerous
>>> print c1.dangerous
2
>>> C.dangerous = 3
>>> print c2.dangerous
3
```

Class attribute

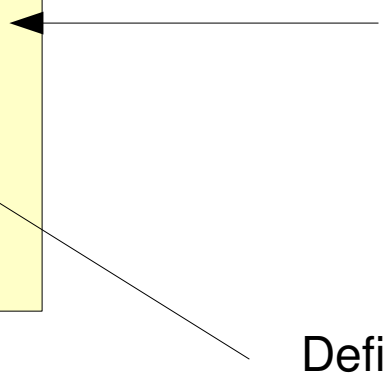
Defines a new data attribute with the same name of the class attribute

To access a class attributes, always user the class name - not the instance name!

Easy to Get Confused

```
>>> class C:
...     def __init__(self):
...         self.safe = 2

>>> c = C()
>>> print c.safe
2
```



Constructor

Defines a data attribute for every new object of this class

Class and Static Methods

- Class methods receive the class instead of the instance as a first argument
- Static methods do not automatically receive any particular information
- Both requires to use special decorators to be defined

```
class Bar:
    @classmethod
    def cmethod(cls):
        print cls

    def method(self):
        print self

    @staticmethod
    def smethod():
        print '??'
```

```
>>> b = Bar()
>>> b.method()
<__main__.Bar instance at 0x94456cc>

>>> b.cmethod()
__main__.Bar

>>> b.smethod()
??
```

Inheritance

```
class DerivedClassName(BaseClass1, BaseClass2,..):
```

- Derived classes may override methods of their base classes
  - In Python all methods are “virtual”
- The base class method can still be called by explicitly invoking:

```
BaseClassName.methodname(self, arguments)
```

- `issubclass()` - check class inheritance
  `issubclass(bool, int)` is True
- In case of multiple inheritance, attributes are searched depth-first, left-to-right

Magic Names

- `__init__(self, …)` – class constructor (invoked after the object is created)
- `__str__(self)` – called to convert an object to a string
- `__len__(self)` – called to answer to len(object)
- `__iter__(self)` – called to iterate over the object (e.g., by a for loop)
- `__getattr__(self, name)` – called when an attribute lookup has not found the attribute in the usual places
- `__setattr__(self,name, value)` – called for every attribute assignment
- `__call__(self, [args])` – called when the instance is “called” as a function
- `__getitem__(self, key)` – called to implement evaluation of object[key]
- `__add__(self, other)` – called to add together two object

New and Old Style Classes

- Old Style:
  - Three types:
    - types (list, tuple, string, ...)
    - classes (any classes that we defined)
    - instances (any instance of a class)
  - Default until python 3.0

- New-style classes were introduced in Python 2.2 to unify classes and types and provide an unified object model
  - A new-style class is just a user-defined type
  - New-style classes are created by inherit from another new-style class or from `object`
  - Default in python 3.0

Exceptions

- An exception is a Python object that represents an error
- Lots of predefined exceptions

```
import exceptions; dir(exceptions)
```

- Can be raised manually

```
raise ExceptionClass [, ExceptionParameter]
raise ExceptionInstance
```

- Defining new exceptions only requires to subclass an exception

```
class NetworkError(RuntimeError):
    def __init__(self, arg):
        self.args = arg
```

Catching Exceptions

```
try:
    #some code

except (RuntimeError, TypeError):
    print 'Runtinme or Type Error'

except IOError, instance:
    print 'IOError ',instance
    raise instance

except:
    print 'Anything else'

else:
    print 'Everything was fine'

finally:
    print "In any case.."
```

Block of code to monitor

Catch two types of exceptions

Catch one exception and keep track of the instance
(from Python 2.6 use "as" instead of comma)

Catch all the other exceptions
ATTENTION: it also catches syntax errors

The else branch is executed if the try clause does not raise an exception

The finally clause is executed under all circumstances