301AA - Advanced Programming

Lecturer: Andrea Corradini
andrea@di.unipi.it
http://pages.di.unipi.it/corradi

Course pages:
http://pages.di.unipi.it/corradi/Didattica/AP-18/

AP-2017-27: More on Python
We have seen:

• Basic and Sequence Datatypes
• Dictionaries
• Control Structures
• List Comprehension
• Function definition
• Positional and keyword arguments of functions
Next topics

• Namespaces and Scopes
• Object Oriented programming in Python
• Inheritance
• Iterators and generators
• Functions as objects
• Higher-order functions
• Importing modules
Namespaces and Scopes

• A **namespace** is a mapping from names to objects: typically implemented as a dictionary. Examples:
  – **builtins**: pre-defined functions, exception names,...
    • Created at interpreter's start-up
  – global names of a **module**
    • Created when the module definition is read
    • Note: names created in interpreter are in module **__main__**
  – local names of a **function invocation**
    • Created when function is called, deleted when it completes
  – and also names of a **class**, names of an **object**...
• Name **x** of a module **m** is an **attribute of m**, accessible with **m.x**. If writable, it can be deleted with **del**.
Namespaces and Scopes (2)

• A scope is a textual region of a Python program where a namespace is \textit{directly accessible}, i.e. reference to a name attempts to find the name in the namespace.

• Scopes are determined statically, but are used dynamically.

• During execution at least three namespaces are directly accessible, searched in the following order:
  – the scope containing the local names
  – the scopes of any enclosing functions, containing non-local, but also non-global names
  – the next-to-last scope containing the current module’s global names
  – the outermost scope is the namespace containing built-in names

• Assignments to names go in the local scope

• Non-local variables can be accessed using \texttt{nonlocal} or \texttt{global}
Example of scoping rules

def scope_test():
    def do_local():
        spam = "local spam"

    def do_nonlocal():
        nonlocal spam
        spam = "nonlocal spam"

    def do_global():
        global spam
        spam = "global spam"

    spam = "test spam"
    do_local()
    print("After local assignment: ", spam)
    do_nonlocal()
    print("After nonlocal assignment: ", spam)
    do_global()
    print("After global assignment: ", spam)

scope_test()
print("In global scope: ", spam)

After local assignment: test spam
After nonlocal assignment: nonlocal spam
After global assignment: nonlocal spam
In global scope: global spam
OOP in Python

- Typical ingredients of the Object Oriented Paradigm:
  - **Encapsulation**: dividing the code into a public interface, and a private implementation of that interface;
  - **Inheritance**: the ability to create subclasses that contain specializations of their parent classes.
  - **Polymorphism**: The ability to override methods of a Class by extending it with a subclass (inheritance) with a more specific implementation (inclusion polymorphism)

From [https://docs.python.org/3/tutorial/classes.html](https://docs.python.org/3/tutorial/classes.html):

- "Python classes provide all the standard features of Object Oriented Programming: the class inheritance mechanism allows multiple base classes, a derived class can override any methods of its base class or classes, and a method can call the method of a base class with the same name. Objects can contain arbitrary amounts and kinds of data. As is true for modules, classes partake of the dynamic nature of Python: they are created at runtime, and can be modified further after creation."
Defining a class

- A class is a blueprint for a new data type with specific internal **attributes** (like a struct in C) and internal operations (**methods**).
- To declare a class in Python the syntax is the following:

  ```
  class className:  
    statements  
    statements 
  ```

- **statements** are assignments or function definitions.
- A new namespace is created, where all names introduced in the statements will go.
- When the class definition is left, a **class object** is created, bound to **className**, on which two operations are defined: **attribute reference** and **class instantiation**.
- Syntax of class instantiation is **className()**, unless a constructor is defined.
Accessing instance attributes

- Class **Points** declares a collection of objects (**instances**) with two **instance attributes**.

- The first statement `p1=Point()` is the call to the **implicit constructor** of the class.

- `p1` is an **instance** (object of class Point)

- Instance attributes can be declared directly inside class or in **constructors** (more common), as we will see later.
Adding instance attributes

- Python objects are dynamic. You can add instance attributes any time!

- The instance attribute \( z \) has been added to the object \( p1 \) (and only to that instance of class \( \text{Point} \)).
Instance methods

- A class can define a set of *instance methods*, which are just functions:

  ```
  def methodname(self, parameter_1, ..., parameter_n):
      statements
  ```

- **Self** must be the first parameter to any instance method. It represents the *implicit parameter* (*this* in Java)

- A method **must** access the object's attributes through the *self* reference

- The **self** parameter must not be passed when the method is called. It is bound to the target object. Syntax:

  ```
  obj.methodname(arg_1, ..., arg_n):
  ```

- But it can be passed explicitly. Alternative syntax:

  ```
  Class. methodname(obj, arg_1, ..., arg_n):
  ```
Example: class **Point** with three instance methods

```python
from math import sqrt

class Point:
    #Attributes:
    x = 0
    y = 0

    #Methods:
    def set_location(self, x, y):
        self.x = x
        self.y = y

    def distance_from_origin(self):
        return sqrt(self.x * self.x + self.y * self.y)

    def distance(self, other):
        dx = self.x - other.x
        dy = self.y - other.y
        return sqrt(dx * dx + dy * dy)

#Main:
P1=Point()
P1.set_location(2, 3)
P2=Point()
P2.set_location(5, 10)
print "Distanza dall'origine di P1: ", P1.distance_from_origin()
print "Distanza tra i due punti P1-P2: ", P1.distance(P2)
```

Equivalent calling syntax:

```python
Pl=Point()
Point.set_location(Pl, 5, 10)
Pl.set_location(5, 10)
```
Constructors

- A constructor is a *special instance method* of a class with name `__init__`. Syntax:

  ```python
def __init__(self, parameter_1, ..., parameter_n):
    statements
  ```

- Invocation: `obj = className(arg_1, ..., arg_n)`

- Returns an object of the class. Parameter `self` is bound to the new object.

- Note: at most ONE constructor (no overloading in Python!)
Defining attributes in constructor

In this version, the instance attributes are not defined in the class but in the constructor.

def _init_(self, x, y):
    self.x = x  # Attribute x is created in the constructor!
    self.y = y  # Attribute y is created in the constructor!

def distance_from_origin(self):
    return sqrt(self.x * self.x + self.y * self.y)

def distance(self, other):
    dx = self.x - other.x
    dy = self.y - other.y
    return sqrt(dx * dx + dy * dy)

def translate(self, dx, dy):
    self.x += dx
    self.y += dy

# Metodo di stampa:
def __str__(self):
    return "(" + str(self.x) + ", " + str(self.y) + ")"
String representation

- It is often useful to have a textual representation of an object with the values of its attributes. This is possible with the following instance method:

```python
def __str__(self):
    return <string>
```

- This is equivalent to Java's `toString` (converts object to a string) and it is invoked automatically when `str` or `print` is called.
Other special methods

- **Method overloading**: you can define special instance methods so that Python's built-in operators can be used with your class.

## Binary Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Class Method</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-</code></td>
<td><code>__sub__(self, other)</code></td>
</tr>
<tr>
<td><code>+</code></td>
<td><code>__add__(self, other)</code></td>
</tr>
<tr>
<td><code>*</code></td>
<td><code>__mul__(self, other)</code></td>
</tr>
<tr>
<td><code>/</code></td>
<td><code>__truediv__(self, other)</code></td>
</tr>
</tbody>
</table>

### Unary Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Class Method</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-</code></td>
<td><code>__neg__(self)</code></td>
</tr>
<tr>
<td><code>+</code></td>
<td><code>__pos__(self)</code></td>
</tr>
</tbody>
</table>

## Similar to C++
Example: special method summing two points componentwise, overloading +.
Private attributes and methods

- Actually in Python is possible to declare `private` instance attributes and methods, i.e. instance attributes and methods that can be used only from a code inside the class!
- All the attributes and methods with a name starting with `__` (except the ones with a name also finishing in `__`) are `private`!
- Example (the interpreter throws an error – `id` can not be used outside the class):

```python
Punto P1_ Id: (2, 2, -2) with __id  
Traceback (most recent call last):
  File "/Users/theuser/Desktop/prova.py", line 19, in <module>
    print "Punto P1_ Id: (2, 2, -2) with __id"
AttributeError: Point instance has no attribute '__id'
```
Encapsulation (and "name mangling")

- **Private** instance variables (not accessible except from inside an object) don’t exist in Python.

- **Convention**: a name prefixed with underscore (e.g. `_spam`) is treated as non-public part of the API (function, method or data member). It should be considered an implementation detail and subject to change without notice.

**Name mangling** ("storpiatura")

- Sometimes class-private members are needed to avoid clashes with names defined by subclasses. Limited support for such a mechanism, called *name mangling*.

- Any name with at least two leading underscores and at most one trailing underscore like e.g. `__spam` is textually replaced with `_classname__spam`, where `classname` is the current class name.
Example for name mangling

- Name mangling is helpful for letting subclasses override methods without breaking intraclass method calls.

```python
class Mapping:
    def __init__(self, iterable):
        self.items_list = []
        self.__update(iterable)

    def update(self, iterable):
        for item in iterable:
            self.items_list.append(item)

    __update = update  # private copy of update() method

class MappingSubclass(Mapping):
    def update(self, keys, values):
        # provides new signature for update()
        # but does not break __init__()
        for item in zip(keys, values):
            self.items_list.append(item)
```
Class attributes

- Python classes can define "**Instance attributes**", associated with each object, and also "**class attributes**" associated with the class (similar to **static variables** in Java).

- Their usage is as follows:
  - Outside the class they are used by referring to the class name: "**classname.attribute**";
  - Inside an instance method they are referred as "**self.__class__.attribute**" or "**classname.attribute**".

- **Example**: we have a class "**Node**" with an instance attribute "**name**". We want to keep tracking on how many times this class is instantiated. We use a class attribute "**count**".
Class vs. instance attributes

- Note that the syntax for introducing an instance attribute or a class attribute is identical. The difference is in the way they are conventionally used.
  - One can mix-up the two concepts, accessing the same attribute in both ways
  - Better to define instance attributes in constructors

- Everything is handled using namespaces:
  - Each class introduces a new namespace
  - Each object introduces a new namespace, nested in the one of its class

- Inspect in the interpreter what happens when the following class is defined. Use `dir` and `__dict__`

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

Static and Class methods

- Sometimes we need to process data associated with classes instead of instances.

- Example: keeping track of instances created or instances currently in memory. This data are associated to class.

- It is worth noting that simple functions written outside the class can suffice for this purpose!

- However, the code is not well associated with class, cannot be inherited and the name of the method is not localized.

- Hence, Python offers us static and class methods.
Static methods

- **Static methods** are simple functions with no `self` argument, preceded by the `@staticmethod` annotation (which is a decorator, see later...)

- They are defined inside a class but they cannot access instance attributes and methods (they cannot access the variable `sender` in the example)

- They can be called through both the class and any instance of that class!

- **Benefits of static methods**: they allow subclasses to customize the static methods with inheritance. Classes can inherit static methods without redefining them. The output of the program is “2” for all the print statements.
Class methods

- Similar to static methods but they have a first parameter which is the class name.
- Definition must be preceded by the `@classmethod` decorator.
- Can be invoked on the class or on an instance.

```python
class Spam:
    numInstances = 0;  # class attribute
    def __init__(self, sender):
        Spam.numInstances = Spam.numInstances + 1
        self.sender = sender #sender is an instance attribute

    @classmethod
    def getInstances(cls):
        return cls.numInstances

#Main:
first = Spam("Marco Rossi")
second = Spam("Pinco Pallino")
print "Numero di istanze di Spam: ", first.getInstances()
print "Numero di istanze di Spam: ", second.getInstances()
print "Numero di istanze di Spam: ", Spam.getInstances()
```
(Multiple) Inheritance, in one slide

• A class can be defined as a derived class

```python
class derived(baseClass):
    statements
    statements
```

• No need of additional mechanisms: when leaving the definition, the class object remembers the baseClass, and uses it as the next non-local scope to resolve names

• All instance methods are automatically virtual

• Python supports multiple inheritance

```python
class derived(base1,..., basen):
    statements
    statements
```

• Diamond problem solved by an algorithm that linearizes the set of all (directly or indirectly) inherited classes: the Method resolution order (MRO)

• https://www.python.org/download/releases/2.3/mro/
What are iterators?

- An **iterator** is an object which allows a programmer to traverse through all the elements of a collection (**iterable** object), regardless of its specific implementation. In Python they are used implicitly by the **FOR** loop construct.

- Python iterator objects required to support two methods:
  - `__iter__` returns the iterator object itself. This is used in **FOR** and **IN** statements.
  - `next` method returns the next value from the iterator. If there is no more items to return then it should raise **StopIteration** exception.

- Remember that an iterator object can be used only once. It means after it raises **StopIteration** once, it will keep raising the same exception.

- Example:

```python
for element in [1, 2, 3]:
    print element
```

```python
>>> lista= [1,2,3]
>>> it = iter(lista)
>>> it
<listiterator object at 0x00A1DB50>
>>> it.next()
1
>>> it.next()
2
>>> it.next()
3
>>> it.next() -> raises StopIteration
```
Iterator objects

- This example shows how to create a class which is an iterator. The iterator can be used in a FOR loop!

```python
class Counter():
    def __init__(self, low, high):
        self.current = low
        self.high = high

    def __iter__(self):
        # Returns itself as an iterator object
        return self

    def next(self):
        # Returns the next value till current is lower than high
        if self.current > self.high:
            raise StopIteration
        else:
            self.current = self.current + 1
        return self.current - 1

# Main:
c = Counter(5, 10)
iterator = iter(c)
while True:
    try:
        x = iterator.next()
        print(x)
    except StopIteration as e:
        break
```

- Output: 5 6 7 8 9 10! Similar to range but slightly different...
Generators are a simple and powerful tool for creating iterators.

They are written like regular functions but use the `yield` statement whenever they want to return data.

Each time the `next()` is called, the generator resumes where it left-off (it remembers all the data values and which statement was last executed).

An example shows that generators can be trivially easy to create.

Anything that can be done with generators can also be done with class based iterators as described in the previous section (not vice-versa).

What makes generators so compact is that the `__iter__()` and `next()` methods are created automatically.

Another key feature is that the local variables and execution state are automatically saved between calls.

This made the function easier to write and much more clear than an approach using class variables like `self.index` and `self.data`. 
In addition to automatic method creation and saving program state, when generators terminate, they automatically raise `StopIteration`.

In combination, these features make it easy to create iterators with no more effort than writing a regular function.

The output is: 0 2 4 6 8!
Back to functions - Recap

• All functions return some value (possibly `None`)
• Function call creates a new namespace
• Parameters are passed by object reference
• Functions can have optional keyword arguments
• Functions can take a variable number of args and kwargs
Function documentation

- The comment after the functions header is bound to the \_\_doc\_\_ special attribute

```python
def my_function():
    """Summary line: do nothing, but document it.
    Description: No, really, it doesn't do anything.
    """
    pass

print(my_function.__doc__)
# Summary line: Do nothing, but document it.
# # Description: No, really, it doesn't do anything.
```
Functions are objects

• As everything in Python, also functions are object, of class `function`

```python
def echo(arg): return arg
type(echo)    # <class 'function'>
hex(id(echo)) # 0x1003c2bf8
print(echo)   # <function echo at 0x1003c2bf8>
foo = echo
hex(id(foo))  # '0x1003c2bf8'
print(foo)    # <function echo at 0x1003c2bf8>
isinstance(echo, object)     # => True
```
Lambdas (anonymous functions)

- They exist! But very limited...
  
  \texttt{lambda arguments: expression}

- The body can only be a single expression
Higher-order functions

• Functions can be passed as argument and returned as result
• Main combinators (map, filter) predefined: allow standard functional programming style in Python
• Heavy use of iterators, which support laziness
Map

```python
>>> print(map.__doc__) % documentation
map(func, *iterables) --> map object
Make an iterator that computes the function using arguments from each of the iterables. Stops when the shortest iterable is exhausted.
```

```python
>>> map(lambda x:x+1, range(4)) % lazyness: returns <map object at 0x10195b278> % an iterator
>>> list(map(lambda x:x+1, range(4)))
[1, 2, 3, 4]
>>> list(map(lambda x, y : x+y, range(4), range(10)))
[0, 2, 4, 6] % map of a binary function
>>> z = 5 % variable capture
>>> list(map(lambda x : x+z, range(4)))
[5, 6, 7, 8]
```
Import and Modules

- Programs will often use classes & functions defined in another file
- A Python module is a single file with the same name (plus the .py extension)
- Modules can contain many classes and functions
- Access using `import`

Where does Python look for module files?

- The list of directories where Python looks: `sys.path`
- When Python starts up, this variable is initialized from the PYTHONPATH environment variable
- To add a directory of your own to this list, append it to this list.
  
  ```python
  sys.path.append('/my/new/path')
  ```

- Oops! Operating system dependent....
Defining Modules

- **Modules** are files containing definitions and statements. A module defines a new namespace.
- Modules can be organized hierarchically in packages.

```python
# File fibo.py - Fibonacci numbers module
def fib(n):    # write Fibonacci series up to n
    a, b = 0, 1
    while b < n:
        print(b, end=' ')
        a, b = b, a+b
    print()

def fib2(n):   # return Fibonacci series up to n
    result = []
    a, b = 0, 1
    while b < n:
        result.append(b)
        a, b = b, a+b
    return result
```
Importing a module

```python
>>> import fibo # imports module from local file 'fibo.py'
>>> fibo.fib(6)  # dot notation
[1, 1, 2, 3, 5]
>>> fibo.__name__ # special attribute __name__
'fibo'
>>> fibo.fib.__module__  # special attribute __module__
'fibo'
```

```python
>>> from fibo import fib, fib2
    # or from fibo import *
>>> fib(500)
>>> fib.__module__    # special attribute __module__
'fibo'
>>> fibo.__name__    # NameError: name 'fibo' is not defined
```
Executing a module as a script

- A module can be invoked as a script from the shell as

```
> python fibo.py 60
```

- Executed with `__name__` set to "`__main__`".

```python
# File fibo.py - Fibonacci numbers module
def fib(n):    # write Fibonacci series up to n
    ...
def fib2(n):  # return Fibonacci series up to n
    ...
if __name__ == "__main__":    # added code
    import sys
    fib(int(sys.argv[1]))
```

```
> python fibo.py 60
1 1 2 3 5 8 13 21 34
> 
```
The **dir()** Function

- The built-in function **dir()** returns a sorted list of strings containing all names defined in a module

```python
>>> import sys
>>> dir(sys)  # Prints names defined in sys
['__displayhook__', '__doc__', '__excepthook__', '__loader__', '__name__', '__package__', '__stderr__', '__stdin__', ...

>>> dir()  # Prints names defined currently
...

>>> import builtins
>>> dir(builtins)  # Prints built-in functions and variables
```