Προγραμματισμός σε C++ & Python & Εφαρμογές στη Ναυπηγική & Ναυτική Μηχανολογία

ΣΝΜΜ 2019

Μάθημα 4A: OOP

Γεώργιος Παπαλάμπρου
Επίκουρος Καθηγητής ΕΜΠ
george.papalambrou@lme.ntua.gr

Εργαστήριο Ναυτικής Μηχανολογίας (Κτίριο Λ)
Σχολή Ναυπηγών Μηχανολόγων Μηχανικών
Εθνικό Μετσοβιο Πολυτεχνείο

March 20, 2019

Περιεχόμενα

Περιεχόμενο Μαθήματος

- Εβδομάδα 1. Α. Εισαγωγή. Η γλώσσα. Το περιβάλλον Linux. Command line. Python interpreter. Ιστοσελίδα μαθήματος. Βιβλιογραφία. Editors: Sublime, Spyder. Β. Εισαγωγή στην γλώσσα Python. Hello World.
- Εβδομάδα 2. Α. Data types. Loops. Control. Β. Παραδείγματα
- Εβδομάδα 3. Functions. Modules
- Εβδομάδα 4. OOP. Classes
- Εβδομάδα 5. **Α. Παραδείγματα: Μέτρηση και επεξεργασία δεδομένων**
  Β. Βιβλιοθήκες NymPy, SciPy. Errors-Exceptions.
  Παραδείγματα: Γραμμική άλγεβρα, Γραφικά
- Εβδομάδα 6. Εφαρμογή: Hardware. Πλατφόρμες. Πρωτόκολλα. Βασικό I/O Εφαρμογή: Neural Networks. Machine Learning
- ΑΝΑΠΛΗΡΩΣΗ ? (2 ΩΡΕΣ, Εβδομάδα 5: File I/O)

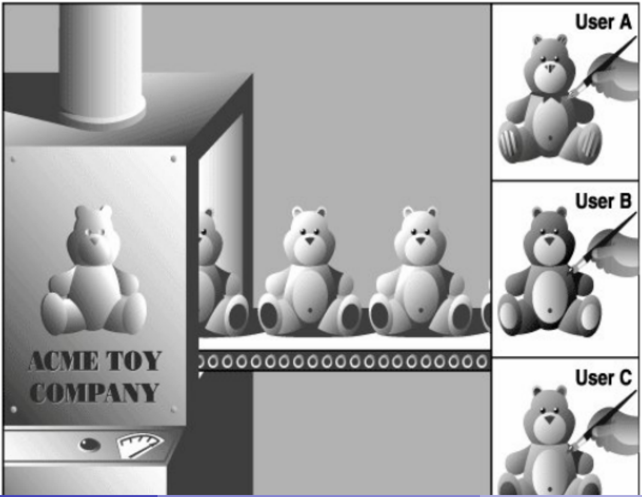
Εισαγωγή

Για σήμερα η παράδοση προέρχεται από:

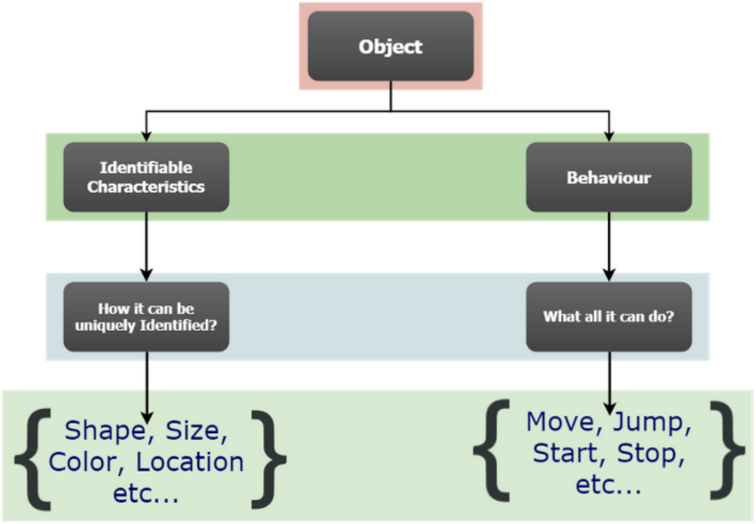
- An Introduction To Python For Scientific Computing, M. Scott Shell
  [Διαθέσιμο στο : http://www.freetechbooks.com/an-introduction-to-python-for-scientific-computing-t885.html]
  Classes, p. 57-59
- Core Python Programming, Wesley Chun
  Chapter 13. OOP

Classes - [Βιβλίο: Wesley Chun]

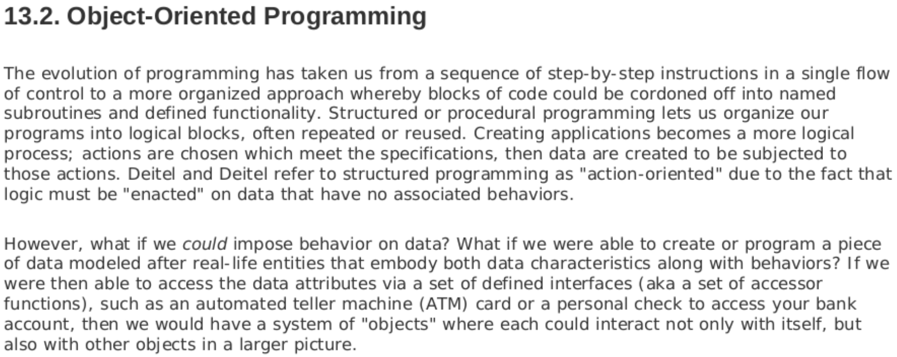
Figure 13-1. The factory manufacturing machines on the left are analogous to classes, while the toys produced are instances of their respective classes. Although each instance has the basic underlying structure, individual attributes like color or feet can be changedthese are similar to instance attributes.



Κλάσεις (Classes) και Αντικείμενα (Objects)


Object
Identifiable Characteristics
Behaviour
How it can be uniquely Identified?
What all it can do?
Shape, Size, Color, Location etc...
Move, Jump, Start, Stop, etc...

OOP - [Βιβλίο: Wesley Chun]

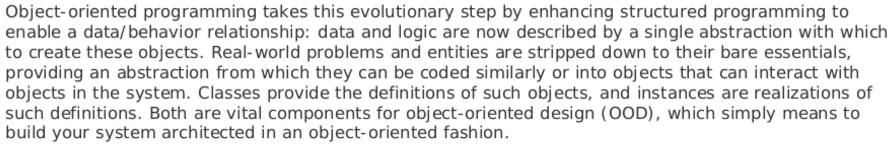


13.2. Object-Oriented Programming

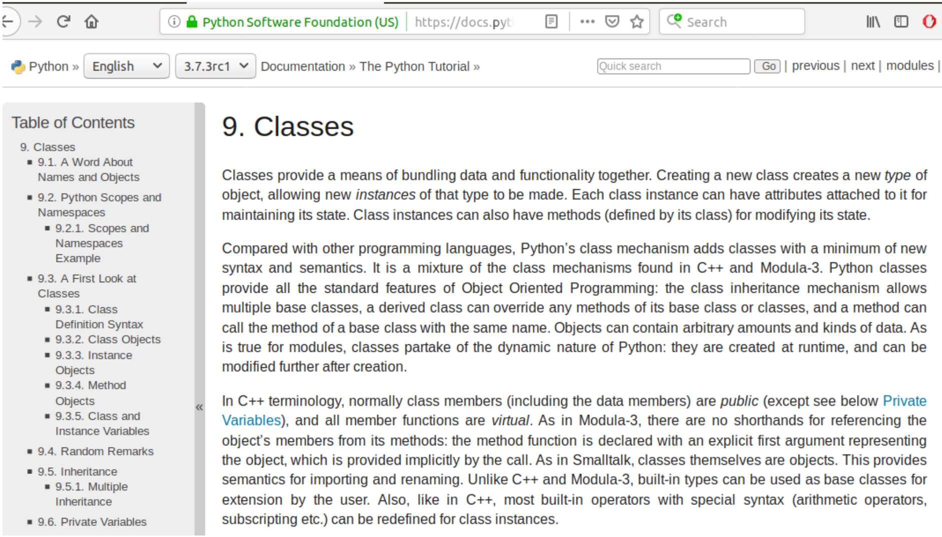
The evolution of programming has taken us from a sequence of step-by-step instructions in a single flow of control to a more organized approach whereby blocks of code could be cordoned off into named subroutines and defined functionality. Structured or procedural programming lets us organize our programs into logical blocks, often repeated or reused. Creating applications becomes a more logical process; actions are chosen which meet the specifications, then data are created to be subjected to those actions. Deitel and Deitel refer to structured programming as "action-oriented" due to the fact that logic must be "enacted" on data that have no associated behaviors.

However, what if we *could* impose behavior on data? What if we were able to create or program a piece of data modeled after real-life entities that embody both data characteristics along with behaviors? If we were then able to access the data attributes via a set of defined interfaces (aka a set of accessor functions), such as an automated teller machine (ATM) card or a personal check to access your bank account, then we would have a system of "objects" where each could interact not only with itself, but also with other objects in a larger picture.

OOP - [Βιβλίο: Wesley Chun]


Object-oriented programming takes this evolutionary step by enhancing structured programming to enable a data/behavior relationship: data and logic are now described by a single abstraction with which to create these objects. Real-world problems and entities are stripped down to their bare essentials, providing an abstraction from which they can be coded similarly or into objects that can interact with objects in the system. Classes provide the definitions of such objects, and instances are realizations of such definitions. Both are vital components for object-oriented design (OOD), which simply means to build your system architected in an object-oriented fashion.

Classes - [https://docs.python.org]



Python Software Foundation (US) https://docs.pyt
Search
Python » English 3.7.3rc1 Documentation » The Python Tutorial »
Quick search Go | previous | next | modules |

Table of Contents

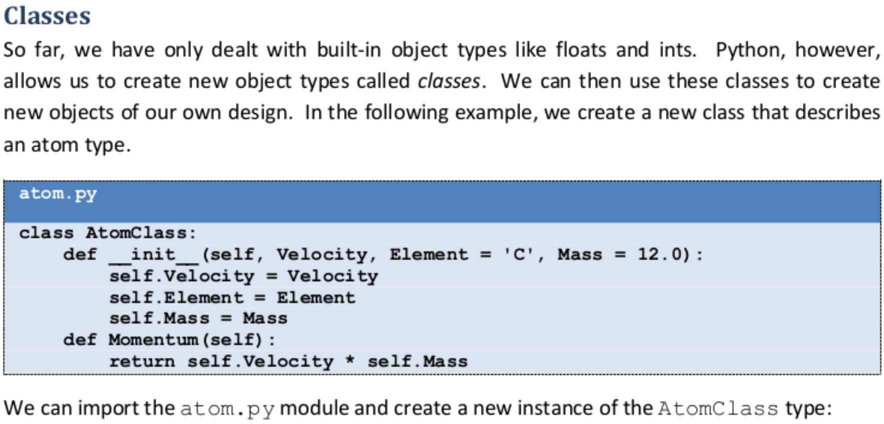
9. Classes

Classes provide a means of bundling data and functionality together. Creating a new class creates a new *type* of object, allowing new *instances* of that type to be made. Each class instance can have attributes attached to it for maintaining its state. Class instances can also have methods (defined by its class) for modifying its state.

Compared with other programming languages, Python's class mechanism adds classes with a minimum of new syntax and semantics. It is a mixture of the class mechanisms found in C++ and Modula-3. 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.

In C++ terminology, normally class members (including the data members) are *public* (except see below Private Variables), and all member functions are *virtual*. As in Modula-3, there are no shorthands for referencing the object's members from its methods: the method function is declared with an explicit first argument representing the object, which is provided implicitly by the call. As in Smalltalk, classes themselves are objects. This provides semantics for importing and renaming. Unlike C++ and Modula-3, built-in types can be used as base classes for extension by the user. Also, like in C++, most built-in operators with special syntax (arithmetic operators, subscripting etc.) can be redefined for class instances.

Classes 1 - [Βιβλίο: Scott Shell]



Classes

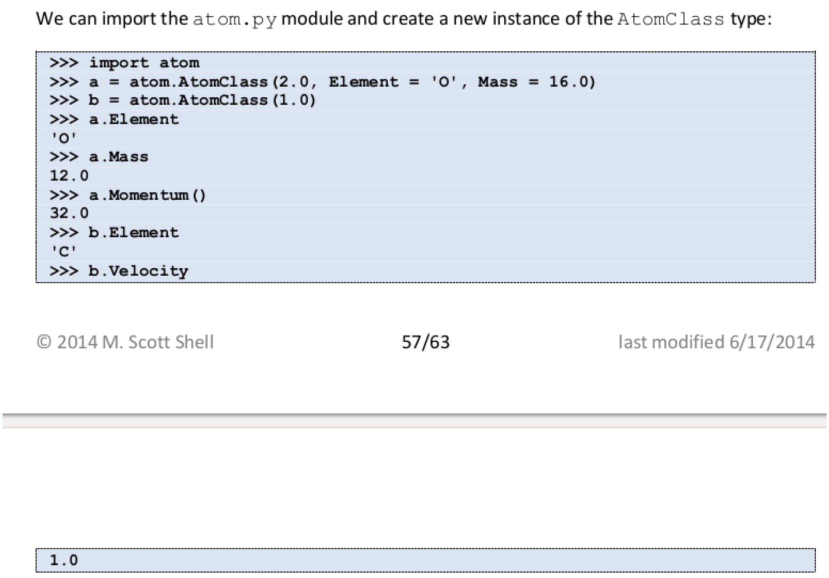
So far, we have only dealt with built-in object types like floats and ints. Python, however, allows us to create new object types called *classes*. We can then use these classes to create new objects of our own design. In the following example, we create a new class that describes an atom type.

atom.py

```
class AtomClass:
    def __init__(self, Velocity, Element = 'C', Mass = 12.0):
        self.Velocity = Velocity
        self.Element = Element
        self.Mass = Mass
    def Momentum(self):
        return self.Velocity * self.Mass
```

We can import the `atom.py` module and create a new instance of the `AtomClass` type:

Classes 1 - [Βιβλίο: Scott Shell]



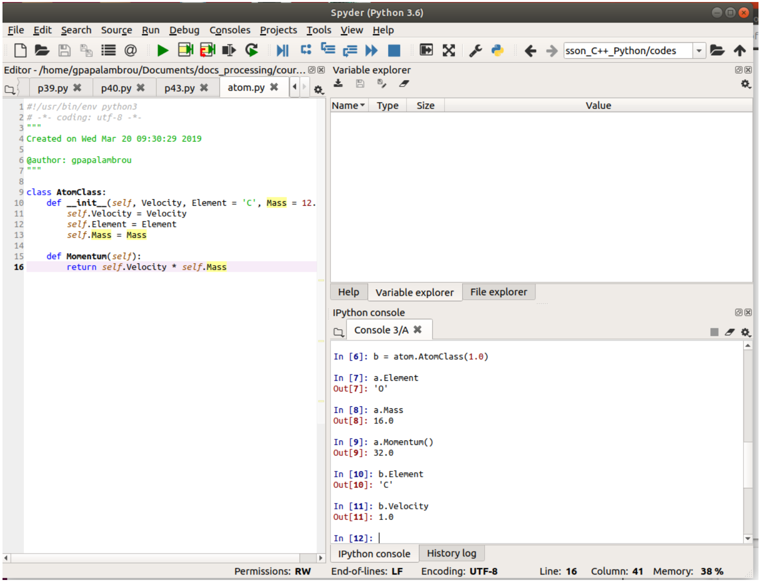
We can import the `atom.py` module and create a new instance of the `AtomClass` type:

```
>>> import atom
>>> a = atom.AtomClass(2.0, Element = 'O', Mass = 16.0)
>>> b = atom.AtomClass(1.0)
>>> a.Element
'O'
>>> a.Mass
12.0
>>> a.Momentum()
32.0
>>> b.Element
'C'
>>> b.Velocity
```

last modified 6/17/2014

```
1.0
```

Classes 1 - Spyder [atom.py]

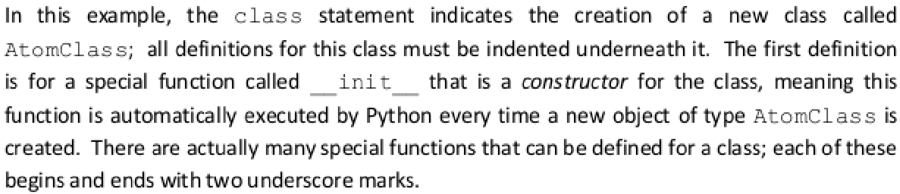

Spyder (Python 3.6)
File Edit Search Source Run Debug Consoles Projects Tools View Help
sson_C++_Python/codes
Editor - /home/gpapalambrou/Documents/docs_processing/cour...
p39.py p40.py p43.py atom.py
Variable explorer
Name Type Size Value
Help Variable explorer File explorer
IPython console
Console 3/A
IPython console History log

```python
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Wed Mar 20 09:30:29 2019

@author: gpapalambrou
"""

class AtomClass:
    def __init__(self, Velocity, Element = 'C', Mass = 12.
        self.Velocity = Velocity
        self.Element = Element
        self.Mass = Mass

    def Momentum(self):
        return self.Velocity * self.Mass
```

```
In [6]: b = atom.AtomClass(1.0)

In [7]: a.Element
Out[7]: 'O'

In [8]: a.Mass
Out[8]: 16.0

In [9]: a.Momentum()
Out[9]: 32.0

In [10]: b.Element
Out[10]: 'C'

In [11]: b.Velocity
Out[11]: 1.0

In [12]:
```

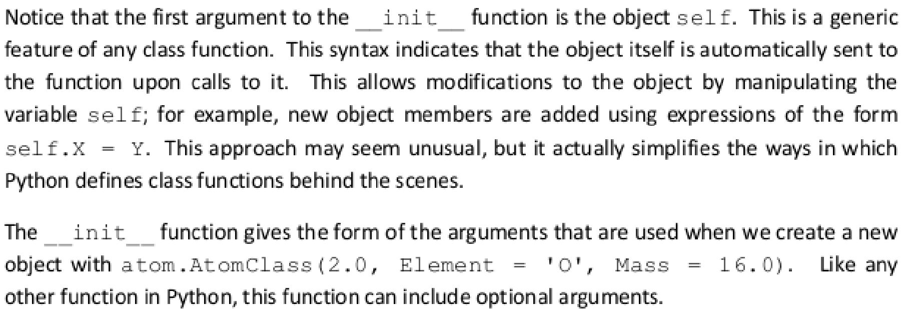
Permissions: RW End-of-lines: LF Encoding: UTF-8 Line: 16 Column: 41 Memory: 38 %

Classes 1 - [Βιβλίο: Scott Shell]



In this example, the `class` statement indicates the creation of a new class called `AtomClass`; all definitions for this class must be indented underneath it. The first definition is for a special function called `__init__` that is a *constructor* for the class, meaning this function is automatically executed by Python every time a new object of type `AtomClass` is created. There are actually many special functions that can be defined for a class; each of these begins and ends with two underscore marks.

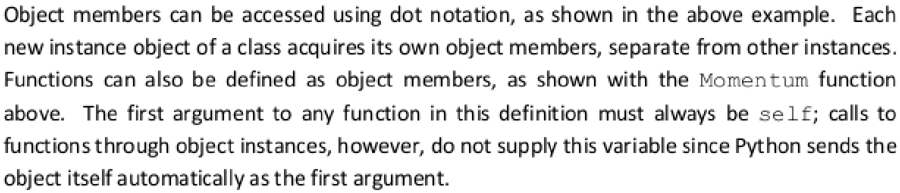
Classes 1 - [Βιβλίο: Scott Shell]



Notice that the first argument to the `__init__` function is the object `self`. This is a generic feature of any class function. This syntax indicates that the object itself is automatically sent to the function upon calls to it. This allows modifications to the object by manipulating the variable `self`; for example, new object members are added using expressions of the form `self.X = Y`. This approach may seem unusual, but it actually simplifies the ways in which Python defines class functions behind the scenes.

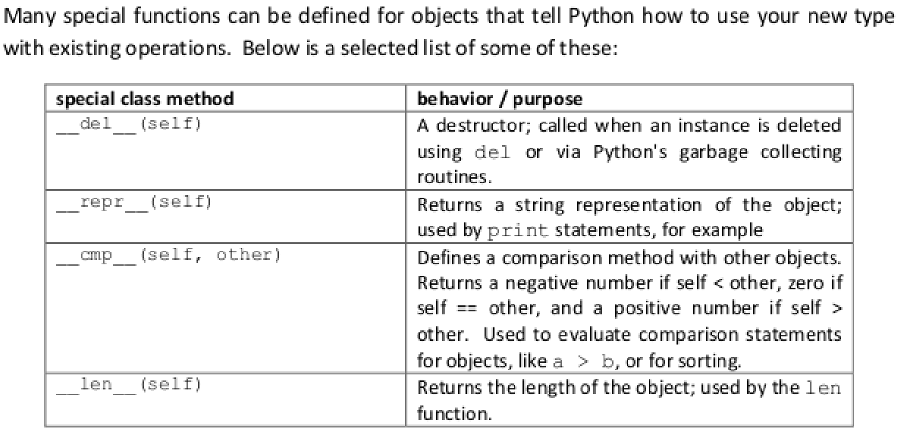
The `__init__` function gives the form of the arguments that are used when we create a new object with `atom.AtomClass(2.0, Element = 'O', Mass = 16.0)`. Like any other function in Python, this function can include optional arguments.

Classes 1 - [Βιβλίο: Scott Shell]



Object members can be accessed using dot notation, as shown in the above example. Each new instance object of a class acquires its own object members, separate from other instances. Functions can also be defined as object members, as shown with the `Momentum` function above. The first argument to any function in this definition must always be `self`; calls to functions through object instances, however, do not supply this variable since Python sends the object itself automatically as the first argument.

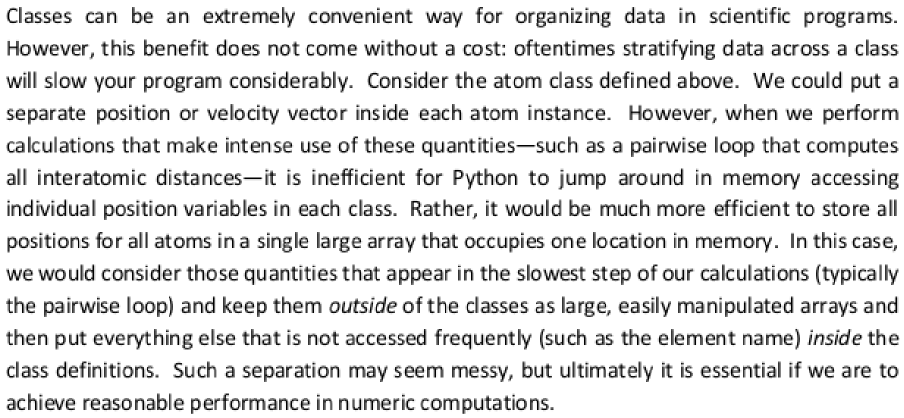
Classes 1 - [Βιβλίο: Scott Shell]



Many special functions can be defined for objects that tell Python how to use your new type with existing operations. Below is a selected list of some of these:

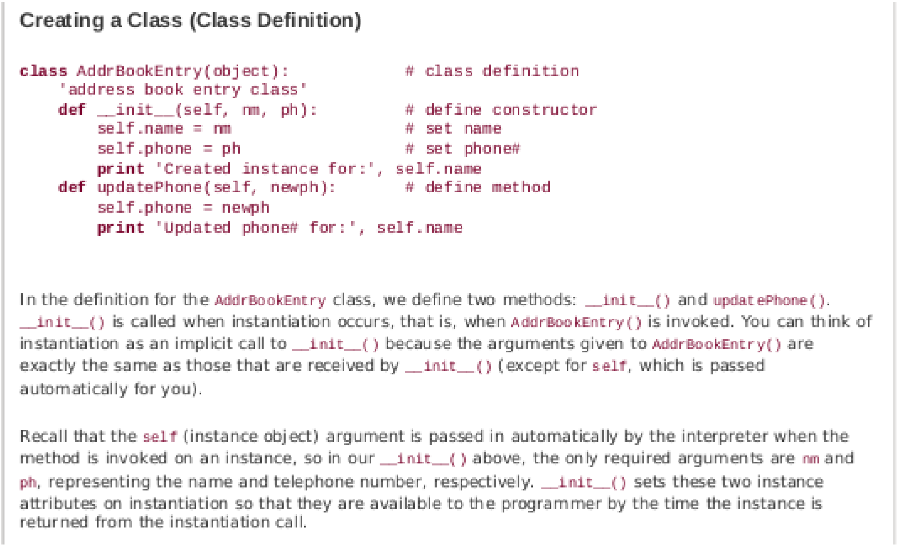
special class method	behavior / purpose
`__del__(self)`	A destructor; called when an instance is deleted using `del` or via Python's garbage collecting routines.
`__repr__(self)`	Returns a string representation of the object; used by `print` statements, for example
`__cmp__(self, other)`	Defines a comparison method with other objects. Returns a negative number if self < other, zero if self == other, and a positive number if self > other. Used to evaluate comparison statements for objects, like `a > b`, or for sorting.
`__len__(self)`	Returns the length of the object; used by the `len` function.

Classes 1 - [Βιβλίο: Scott Shell]



Classes can be an extremely convenient way for organizing data in scientific programs. However, this benefit does not come without a cost: oftentimes stratifying data across a class will slow your program considerably. Consider the atom class defined above. We could put a separate position or velocity vector inside each atom instance. However, when we perform calculations that make intense use of these quantities—such as a pairwise loop that computes all interatomic distances—it is inefficient for Python to jump around in memory accessing individual position variables in each class. Rather, it would be much more efficient to store all positions for all atoms in a single large array that occupies one location in memory. In this case, we would consider those quantities that appear in the slowest step of our calculations (typically the pairwise loop) and keep them *outside* of the classes as large, easily manipulated arrays and then put everything else that is not accessed frequently (such as the element name) *inside* the class definitions. Such a separation may seem messy, but ultimately it is essential if we are to achieve reasonable performance in numeric computations.

Classes 2 - [Βιβλίο: Wesley Chun]



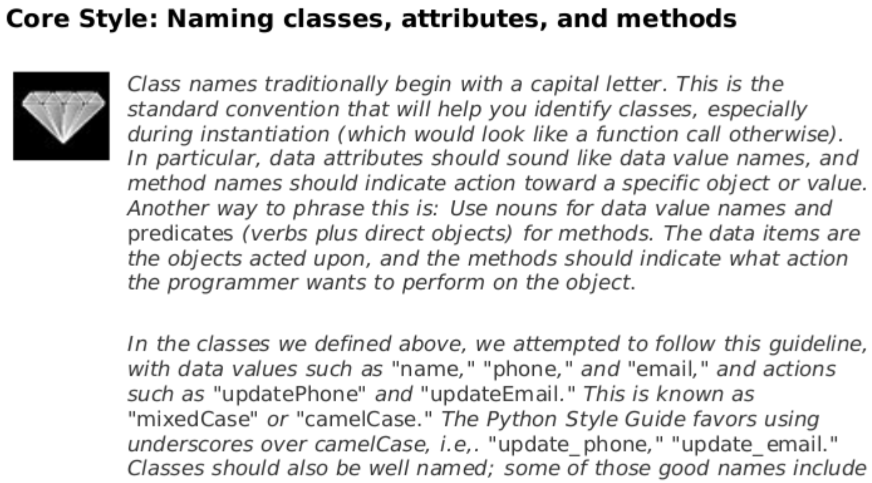
Creating a Class (Class Definition)

```
class AddrBookEntry(object):            # class definition
    'address book entry class'
    def __init__(self, nm, ph):         # define constructor
        self.name = nm                  # set name
        self.phone = ph                 # set phone#
        print 'Created instance for:', self.name
    def updatePhone(self, newph):       # define method
        self.phone = newph
        print 'Updated phone# for:', self.name
```

In the definition for the AddrBookEntry class, we define two methods: `__init__()` and `updatePhone()`. `__init__()` is called when instantiation occurs, that is, when `AddrBookEntry()` is invoked. You can think of instantiation as an implicit call to `__init__()` because the arguments given to `AddrBookEntry()` are exactly the same as those that are received by `__init__()` (except for `self`, which is passed automatically for you).

Recall that the `self` (instance object) argument is passed in automatically by the interpreter when the method is invoked on an instance, so in our `__init__()` above, the only required arguments are `nm` and `ph`, representing the name and telephone number, respectively. `__init__()` sets these two instance attributes on instantiation so that they are available to the programmer by the time the instance is returned from the instantiation call.

Classes: Naming - [Βιβλίο: Wesley Chun]

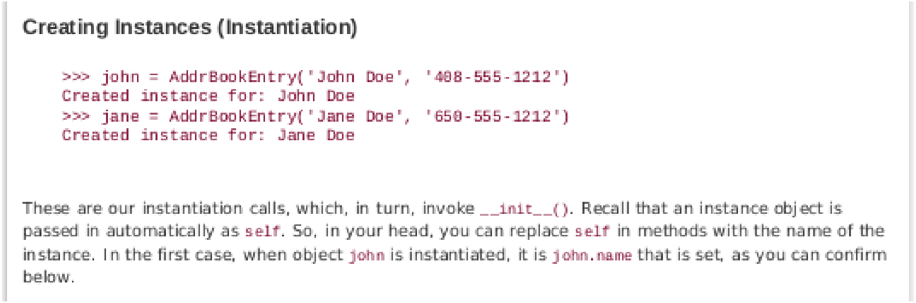


Core Style: Naming classes, attributes, and methods

Class names traditionally begin with a capital letter. This is the standard convention that will help you identify classes, especially during instantiation (which would look like a function call otherwise). In particular, data attributes should sound like data value names, and method names should indicate action toward a specific object or value. Another way to phrase this is: Use nouns for data value names and predicates *(verbs plus direct objects) for methods. The data items are the objects acted upon, and the methods should indicate what action the programmer wants to perform on the object.*

In the classes we defined above, we attempted to follow this guideline, with data values such as "name," "phone," *and* "email," *and actions such as* "updatePhone" *and* "updateEmail." *This is known as* "mixedCase" *or* "camelCase." *The Python Style Guide favors using underscores over camelCase, i.e,.* "update_phone," "update_email." *Classes should also be well named; some of those good names include*

Classes 2 - [Βιβλίο: Wesley Chun]

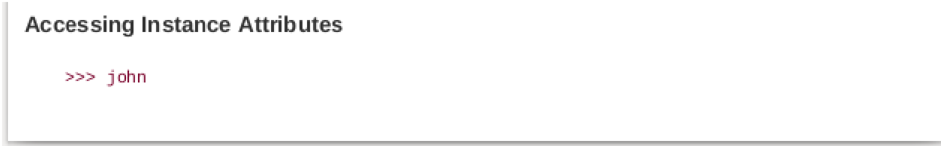


Creating Instances (Instantiation)

```
>>> john = AddrBookEntry('John Doe', '408-555-1212')
Created instance for: John Doe
>>> jane = AddrBookEntry('Jane Doe', '650-555-1212')
Created instance for: Jane Doe
```

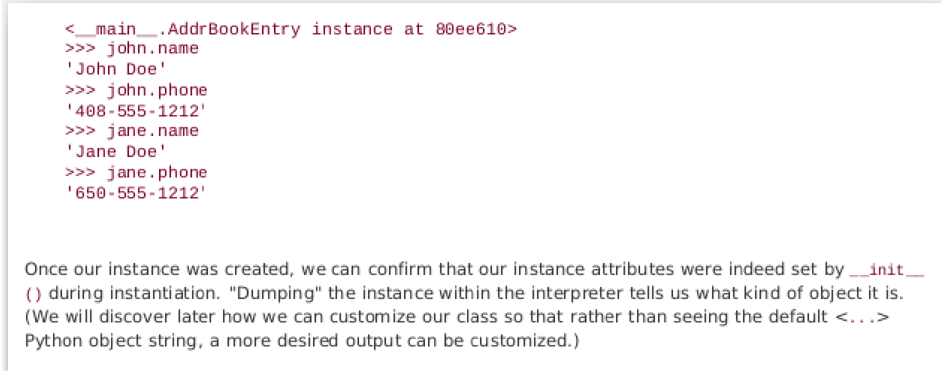
These are our instantiation calls, which, in turn, invoke `__init__()`. Recall that an instance object is passed in automatically as `self`. So, in your head, you can replace `self` in methods with the name of the instance. In the first case, when object `john` is instantiated, it is `john.name` that is set, as you can confirm below.

Classes 2 - [Βιβλίο: Wesley Chun]



Accessing Instance Attributes

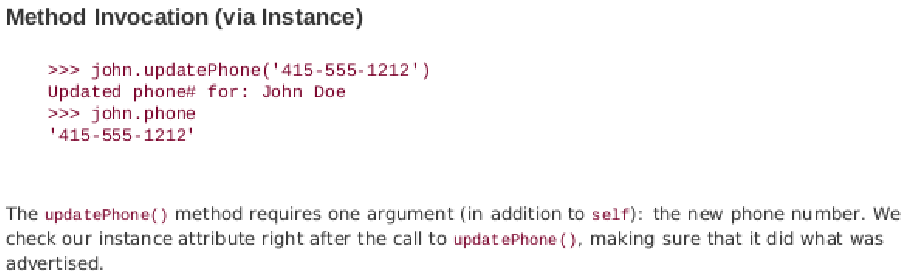
```
>>> john
```



```
<__main__.AddrBookEntry instance at 80ee610>
>>> john.name
'John Doe'
>>> john.phone
'408-555-1212'
>>> jane.name
'Jane Doe'
>>> jane.phone
'650-555-1212'
```

Once our instance was created, we can confirm that our instance attributes were indeed set by `__init__()` during instantiation. "Dumping" the instance within the interpreter tells us what kind of object it is. (We will discover later how we can customize our class so that rather than seeing the default `<...>` Python object string, a more desired output can be customized.)

Classes 2 - [Βιβλίο: Wesley Chun]



Method Invocation (via Instance)

```
>>> john.updatePhone('415-555-1212')
Updated phone# for: John Doe
>>> john.phone
'415-555-1212'
```

The `updatePhone()` method requires one argument (in addition to `self`): the new phone number. We check our instance attribute right after the call to `updatePhone()`, making sure that it did what was advertised.

Classes: TO DO ! - [Βιβλίο: Wesley Chun]

Μελετήστε τα: 13.3 - 13.11.