

# Python objects

Reuven M. Lerner, PhD  
[reuven@lerner.co.il](mailto:reuven@lerner.co.il)

# Objects

- We use a lot of objects in Python.
- But wait: What is an object in Python?
  - It has a type ( = “class”)
  - It has attributes (data + functions)
- Nearly everything in Python is an object!

# Everything has a type

```
type('abc')
    str
type(100)
    int
type(type)
    type
type(len)
    builtin_function_or_method
def hello():
    pass
type(hello)
    function
```

# What is a type?

- A type is a factory for objects with predefined attributes
- Attribute names are shared across all instances of that type
- Attribute values, of course, can be different
- Each instance can have custom attribute name-value pairs, as well

# Everything has attributes

```
dir('abc')[:4]
['__add__', '__class__', '__contains__', '__delattr__']
dir(100)[:4]
['__abs__', '__add__', '__and__', '__class__']
dir(type(type))[:3]
['__abstractmethods__', '__base__', '__bases__']
dir(len)[:4]
['__call__', '__class__', '__cmp__', '__delattr__']
def hello(): pass
dir(hello)[:4]
['__call__', '__class__', '__closure__', '__code__']
```

# Getting attributes

- Python's dot notation is shorthand for the built-in "getattr" function:

```
>>> import os
```

```
>>> os.pathsep
```

```
'.'
```

```
>>> getattr(os, 'pathsep')
```

```
'.'
```

# Setting attributes

- Similarly, we can change attributes with setattr:

```
>>> os.foo = 'abc'
```

```
>>> os.foo
```

```
'abc'
```

```
>>> setattr(os, 'foo', 'def')
```

```
>>> os.foo
```

```
'def'
```

# Python classes

- User-defined types are known as “classes”
- Use the reserved word “class” to define it

```
class Foo(object):
```

```
    pass # empty class
```

```
type(Foo)
```

```
type
```



# Old-style classes

- Python 2 supports two styles of classes
- New-style classes inherit from “object” (i.e., we name object inside of the parentheses)
- Old-style classes don't specify any base class
- You should only use new-style classes, but that means remembering to inherit from object

# Instances

- “Execute” a class to get an instance:

```
s = set()                                # gives us a set
```

```
mylist = list()                          # gives us a list
```

```
i = int('0x123abc', 16)                 # gives us an int
```

# Instances

```
class Person(object):
```

```
    pass
```

```
p = Person() # create an instance
```

```
type(p)      # what is p's class?
```

```
    __main__.Person
```

```
p           # p's representation
```

```
<__main__.Person at 0x931c0>
```

# Docstrings for classes

```
class Person(object):
```

```
    "Describes a person"
```

```
    pass
```

```
help(Person)    # Shows the docstring
```

# Docstrings

- There are three levels of docstrings: Module, class, and function
- If you use docstrings in all of your files, the help and pydoc commands will generate documentation, in a set format

# Constructor

- Python has a constructor, called `__new__`
- Under almost no circumstances should you actually call this!
- Instead, we define `__init__`, which is effectively our constructor

# `__init__`

- Method run after the instance is created
- This is where you should define attributes (and their default values) — perhaps from parameters passed to `__init__`
- `__init__` (like all methods) always takes one parameter, traditionally called `self`, which is the newly created object (instance)

# `__init__` example

```
class Person(object):  
  
    "Describes a person"  
  
    def __init__(self):  
  
        self.first_name = ""  
  
        self.last_name = ""  
  
        self.email_address = ""  
  
p = Person()  
  
p.first_name          # ""
```



# self

- self is a variable (not a reserved word)
- It is the current instance
- Like “this” in some other languages
- Always, always use self — an extremely strong convention

# Init parameters

```
class Person(object):  
  
    "Describes a person"  
  
    def __init__(self, first_name, last_name):  
  
        self.first_name = first_name  
  
        self.last_name = last_name  
  
p = Person("John", "Smith")  
  
p.first_name                # "John"
```

# Variables vs. attributes

```
class Person(object):
```

```
    def __init__(self, first_name, last_name):
```

```
        self.first_name = first_name
```

```
        self.last_name = last_name
```

- self, first\_name, and last\_name are variables
- self has two (explicitly defined) attributes, first\_name and last\_name

# Instance attributes

- Any attribute set on self is attached to the instance
- We can, at any point, set new instance attributes — but it's traditional to set them all in `__init__`
- Use default parameter values in `__init__` to give your instances default attribute values

# Instance attributes

- Instance attributes can be set or retrieved from anywhere in the program
- No setters or getters are needed!

```
p.first_name = 'xxx'
```

```
p.first_name
```

```
'xxx'
```

# Private and protected

- Everything is public in Python!
- We dissuade people via conventions:

```
self.regular = 'hello'
```

```
self._semi_private = 'shh'
```

```
self.__name_mangled = 'shh!!'
```

```
self.__reserved__ = 'careful'
```

# Private and protected

- Attributes defined inside of an instance as `__name` will be available outside as `__classname_name`.
- Attributes of the form `__internal__` are treated specially by Python, and should only be set if you know what you're doing.

# Attribute types

- As always, Python's dynamic typing allows us to store anything in an attribute
- It's most common to store integers and strings
- But it's not at all unusual to have lists, tuples, and dicts — as well as other objects



# Creating an instance

```
p = Person()
```

```
type(p)
```

```
__main__.Person()
```

# Instance methods

- Functions defined within the class definition are instance methods
- As with `__init__`, `self` (i.e., the object itself) is silently passed as the first parameter
- Additional parameters can also be passed

# Simple getter

```
class Person(object):  
    def __init__(self, first_name, last_name):  
        self.first_name = first_name  
        self.last_name = last_name  
    def get_first_name(self):  
        return self.first_name  
  
p = Person("John", "Smith")  
p.get_first_name()  
    'John'
```

# Simple setter

```
class Person(object):
    def __init__(self, first_name, last_name):
        self.first_name = first_name
        self.last_name = last_name
    def get_first_name(self):
        return self.first_name
    def set_first_name(self, new_name):
        self.first_name = new_name
p = Person("John", "Smith")
p.set_first_name("Foo")
p.get_first_name()
    "Foo"
```

# Do we need them?

- Normally, we don't create setters and getters in Python
- Rather, we just set and retrieve directly, using the attribute name
- You can, in theory, create new attributes from outside of a method — but that's considered to be in bad taste

# One person's opinion

“Getters and setters are evil. Evil, evil, I say! Python objects are not Java beans. Do not write getters and setters. This is what the ‘property’ built-in is for. And do not take that to mean that you should write getters and setters, and then wrap them in ‘property’.

— Philip J. Eby, <http://dirtsimple.org/2004/12/python-is-not-java.html>

# Another method

```
class Person(object):  
    def __init__(self, first_name, last_name):  
        self.first_name = first_name  
        self.last_name = last_name  
    def get_full_name(self):  
        return '{}{}'.format(self.first_name,  
                               self.last_name)  
  
p = Person("John", "Smith")  
p.get_full_name()  
    'John Smith'
```

# So...

```
p = Person('Reuven', 'Lerner')  
  
p.first_name = 'foo'  
  
p.last_name = 'bar'  
  
print(p.fullname())      # 'foo bar'
```



# return

- As with all functions, you must explicitly return a value
- Otherwise, the method returns None
- You can return complex types from methods, just as you could from functions
- (Methods are simply functions, bound to a class)

# Method rewrite

- Why do we get self as the first parameter?
- Because Python takes

```
p.set_first_name("Foo")
```

- and turns it into

```
Person.set_first_name(p, "Foo")
```

# Bad error messages

```
class Math(object):
```

```
    def mult(first, second):  # No self!
```

```
        return first * second
```

```
m = Math()
```

```
m.mult(3,5)
```

```
TypeError: mult() takes exactly 2 arguments (3 given)
```

# Explore an object

```
dir(p)[-4:]
```

```
['__weakref__', 'first_name',
```

```
 'get_first_name', 'set_first_name']
```

# Destructors

- Nothing!
  - (Or almost nothing)
- You can define `__del__`, if you want. But you probably won't need to in most cases.
- `__del__` executes when an object is garbage collected (or when the program exits), not when the variable is deleted

# Class attributes

- We can define an attribute on a class, by defining it at the class level
- There is no “self” in a class
- Just assign to the variable, and it will be defined on the class

# Class attribute

```
class Person(object):  
    population = 0  
    def __init__(self, first_name, last_name):  
        self.first_name = first_name  
        self.last_name = last_name  
  
print(Person.population)  # no instance needed!  
0
```

# Is that assignment?

- Yes, it is!
- You can execute arbitrary code inside of the class definition
- As a general rule, you only want to assign to class attributes



# Retrieving class attributes

- Classes are objects (instances of type)
- Class attributes have the same rules as all other attributes — just name the class

```
Person.population = 5
```

```
print(Person.population)    # 5
```

# Instances and class attributes

- This is where things get tricky!
- If you try to get an attribute that doesn't exist on the instance, Python will look on the object's class. So:

```
p = Person( 'Reuven' , 'Lerner' )
```

```
Person.population = 5
```

```
print(p.Population) # 5
```

# Instance and class attributes

- Setting an attribute on an always works!
- If you set an object's attribute, and the class has one of the same name:

```
p = Person('Reuven', 'Lerner')
Person.population = 100
p.population = 5
print(p.Population)           # 5
print(Person.Population)      # 100
```

# More fun

```
class Person(object):  
    population = 0  
    def __init__(self, name):  
        self.name = name  
        self.population += 1  
Person.population  
0  
p = Person('first')  
Person.population  
0  
p.population  
1
```

# Moral: Don't do that!

- Don't use the same names for instance and class attributes
- Set class attributes via the class, not via the instance — unless you want "dynamic class attributes"
- Take advantage of this “look at the class” functionality for more flexible classes and inheritance (more on this in a bit)

# Python scoping rules

- Variables
  - Local
  - Enclosing func
  - Global (module)
  - Built-in
- Attributes
  - Object
  - Object's class
  - Object's class parents

# OK, a bit more complex:

- From an object:
  - Look in the object
  - Look in the object's class
  - Look in up the inheritance tree of the object's class (i.e., the superclasses)

# As for classes:

- From an object:
  - Look in the class
  - Look in up the inheritance tree of the object's class (i.e., the superclasses)



# Inheritance

- New-style objects inherit from “object”
- What does inheritance mean in Python?
- One thing: If a method isn't found in a class, we look in its parent class
- Example: Didn't define `__init__`? That's fine, because “object” did

# Simple inheritance

```
class Employee(Person):
    def __init__(self, first_name,
                  last_name, employee_id):
        self.first_name = first_name
        self.last_name = last_name
        self.employee_id = employee_id
e = Employee('first', 'last', 100)
print(e.fullname())                # Prints 'first last'
```

# Why does this work?

- We invoke `e.fullname()`
- Python looks on `e`'s class (`Employee`)
- `Employee.fullname()` doesn't exist
- So it goes to `Employee`'s parent class
- This continues until we get to object (i.e., the ultimate base class)

# `__bases__` tuple

`Employee.__bases__`

`(__main__.Person,)`

`Person.__bases__`

`(object,)`

`object.__bases__`

`()`

# Checking is-a

- `issubclass(C, B)` — returns True if C is a subclass of B
- `isinstance(O, C)` — returns True if O is an instance of C

# Not very DRY

- Notice that `Employee.__init__` had to repeat code from `Person.__init__`:

```
class Employee(Person):  
  
    def __init__(self, first_name,  
  
                  last_name, employee_id):  
  
        self.first_name = first_name  
  
        self.last_name = last_name  
  
        self.employee_id = employee_id
```

# Data inheritance

- Python doesn't offer data inheritance
- If we want Employee to have “first\_name” and “last\_name” fields, we need to define them
- But duplicating code in each `__init__` is asking for trouble

# Easiest solution

```
class Employee(Person):  
  
    def __init__(self, first_name,  
  
                  last_name, employee_id):  
  
        Person.__init__(self, first_name, last_name)  
  
        self.employee_id = employee_id
```



# Why does this work?

- self already exists, as a naked instance of Employee, when Employee.\_\_init\_\_ is invoked
- We explicitly invoke Person.\_\_init\_\_, passing self as its first parameter
- Person.\_\_init\_\_ defines its attributes
- Employee.\_\_init\_\_ adds attributes

# super()

- In other languages, we would use `super()` to invoke the same method in the parent class
- Not in Python!
- `super()` doesn't execute a method — it returns an object, on which we then invoke a method
- You pass `super()` a class and an instance

# Using super()

```
class Employee(Person):  
  
    def __init__(self, first_name,  
  
                  last_name, employee_id):  
  
        super(Employee, self).__init__(first_name,  
  
                                       last_name)  
  
        self.employee_id = employee_id
```

# Why?

- `super()` works this way because Python supports multiple inheritance
- Thus, you can get yourself into a situation where there is no one “superclass”
- Thus, Python has to figure out which object should receive a method call
- `super()` returns a proxy to that object

# Overloading operators

- We can add numbers:

5 + 3

- We can concatenate strings:

"foo" + "bar"

- What if I want to use + with my own object?
- This is known as “operator overloading”

# Overriding len()

```
class Person(object):  
    def __init__(self):  
        self.height = 180  
    def __len__(self):  
        return self.height  
  
p = Person()  
  
print(len(p))    # Prints 180
```

# Overriding +

```
class Person(object):  
    def __init__(self, height=180):  
        self.height = height  
    def __add__(self, other):  
        return self.height + other.height  
  
p = Person()  
  
print(p + p)      # prints 360
```

# Overriding [ ]

```
class Person(object):  
    def __init__(self, name='Reuven'):  
        self.name = name  
    def __getitem__(self, i):  
        return self.name[i]  
  
p = Person()  
print(p[3])          # v  
print(p[50])         # IndexError
```



# `__repr__` and `__str__`

- There are two ways to turn your object into a string:
  - `__repr__` is what the interactive Python shell shows for an object. It should be a valid Python expression in the string.
  - `__str__` is what the object returns when passed to `str()`
- If `__str__` isn't defined, `__repr__` is called

# Many overloading methods

- Python provides an overwhelming number of methods that you can define to overload built-in operators
- Use the ones that make your objects work in ways that seem logical
- Ignore the rest of them!