Secure Programming via Libraries

Python in a Nutshell

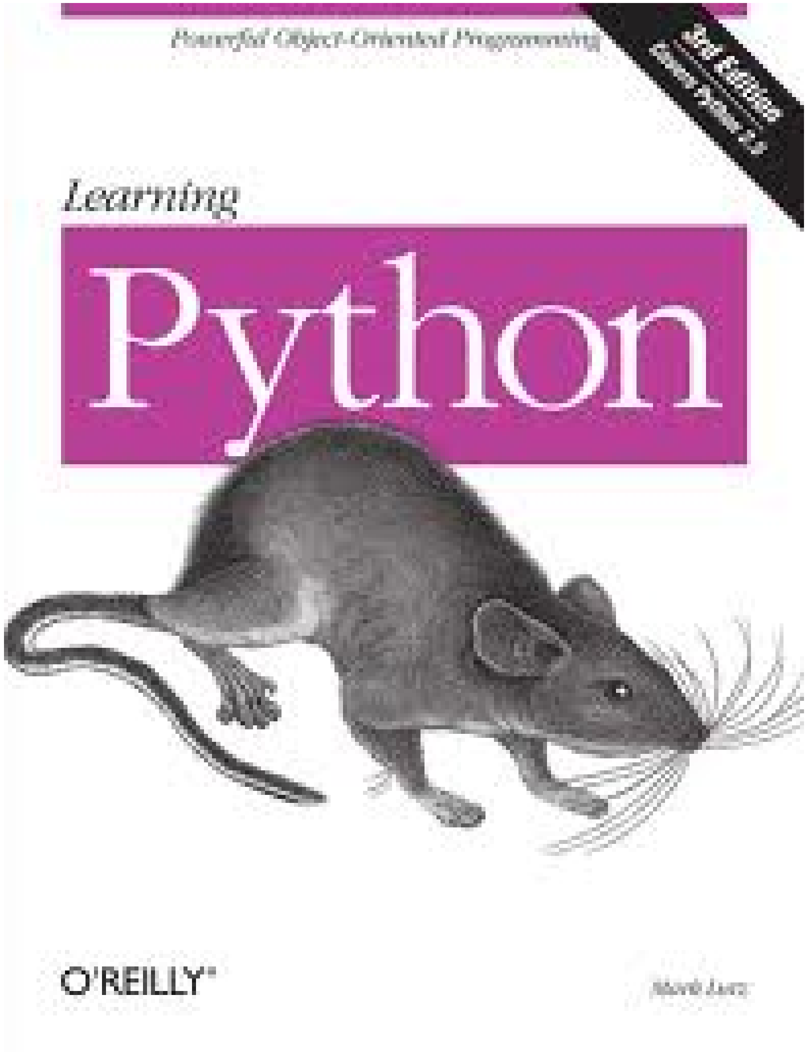
Alejandro Russo (russo@chalmers.se)

Escuela de Ciencias Informáticas (ECI) 2011
UBA, Buenos Aires, Argentina

CHALMERS

Learning Python

- By Mark Lutz
- Available online
- Learn it on demand
- We will see Python in a Nutshell
- Great programming language
- Highly used by Google



Python

- Programming language
  - Dynamically typed
  - Imperative
  - Object-oriented
  - Functional
- It does not force you to use a feature or programming paradigm that you do not want
- Open source, clean syntax, easy to learn
- There are several flavors of Python
- We use the one provided by the Python Software Foundation [Python]

Python: Relevant Features

- ***Very dynamic language***
  - **You can modify the behavior of almost any entity dynamically**
- *Everything* is an object
  - They have dictionaries indicating the supporting operations
- Variables are references to objects
- Types are associated with objects, not variables
- Multiple-inheritance
- Overloading
- Decorators

Everything is an Object

```
x = "Hello word!"
y = "... Goodbye!"

def f(x,y):
    print "You are calling function f"
    print "..."
    return x+y
```

```
$ python -i objects.py
>>> x
'Hello word!'
>>> y
'... Goodbye!'
>>> f(x,y)
You are calling function f
...
'Hello word!... Goodbye!'
>>> dir(x)
['__add__', '__class__', '__contains__', '__delattr__', '__doc__', '__eq__',
'__format__', '__ge__', '__getattribute__', '__getitem__', '__getnewargs__',
'__getslice__', '__gt__', '__hash__', '__init__', '__le__', '__len__',
'__lt__', '__mod__', '__mul__', '__ne__', '__new__', '__reduce__',
'__reduce_ex__', '__repr__', '__rmod__', '__rmul__', '__setattr__',
'__sizeof__', '__str__', '__subclasshook__', '_formatter_field_name_split',
'_formatter_parser', 'capitalize', 'center', 'count', 'decode', 'encode',
'endswith', 'expandtabs', 'find', 'format', 'index', 'isalnum', 'isalpha',
'isdigit', 'islower', 'isspace', 'istitle', 'isupper', 'join', 'ljust',
'lower', 'lstrip', 'partition', 'replace', 'rfind', 'rindex', 'rjust',
'rpartition', 'rsplit', 'rstrip', 'split', 'splitlines', 'startswith',
'strip', 'swapcase', 'title', 'translate', 'upper', 'zfill']
>>> x.isdigit()
False
>>>
```

Everything is an Object

```
x = "Hello word!"
y = "... Goodbye!"

def f(x,y):
    print "You are calling function f"
    print "..."
    return x+y
```

```
>>> dir(f)
['__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']
>>> f.__call__("Buenos ", "Aires")
You are calling function f
...
'Buenos Aires'
>>>
```

Variables are References

```
x = "Hello word!"
y = x
print "x is: ", x
print "y is: ", y
x = "... Goodbye!"
print 'After x = "... Goodbye!"'
print "x is: ", x
print "y is: ", y
```

```
$ python -i references.py
x is:  Hello word!
y is:  Hello word!
After x = "... Goodbye!"
x is:  ... Goodbye!
y is:  Hello word!
>>>
```

Types and Variables

```
$ python -i types.py
>>> x.__class__
<type 'str'>
>>> y.__class__
<type 'int'>
>>> f.__class__
<type 'function'>
>>> x
'Hello word!'
>>> y
3
>>> x = y
>>> x.__class__
<type 'int'>
>>> x
3
>>>
```

```
x = "Hello word!"

y = 3

def f(x):
    return x
```

Classes (classic style)

```python
class Klass:
      def setdata(self, value):
          self.data=value
      def display(self):
          print self.data
```

```
python -i classes.py
>>> obj = Klass()
>>> dir(obj)
['__doc__', '__module__', 'display', 'setdata']
>>> obj.setdata(42)
>>> dir(obj)
['__doc__', '__module__', 'data', 'display', 'setdata']
>>> obj.display()
42
>>> type(obj)
<type 'instance'>
>>>
```

Classes (new-style)

```
class Klass1(object):
      def setdata(self, value):
          self.data=value
      def display(self):
          print self.data
```

```
python -i classes.py
>>> obj = Klass1()
>>> dir(obj)
['__class__', '__delattr__', '__dict__', '__doc__', '__format__',
'__getattribute__', '__hash__', '__init__', '__module__', '__new__',
'__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__',
'__str__', '__subclasshook__', '__weakref__', 'display', 'setdata']
>>> obj.setdata(42)
>>> dir(obj)
['__class__', '__delattr__', '__dict__', '__doc__', '__format__',
'__getattribute__', '__hash__', '__init__', '__module__', '__new__',
'__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__',
'__str__', '__subclasshook__', '__weakref__', 'data', 'display',
'setdata']
>>> obj.display()
42
>>> type(obj)
<class '__main__.Klass1'>
>>>
```

Unify types and classes. It also add some support for meta-programming

Inheritance

```
class Klass2(Klass1):
      def display(self):
          print "Current value = %s"%self.data
```

```
python -i classes.py
>>> obj = Klass2()
>>> obj.setdata(42)
>>> obj.display()
Current value = 42
>>>
```

It supports multiple-inheritance. For that, it uses the C3 Method Resolution algorithm

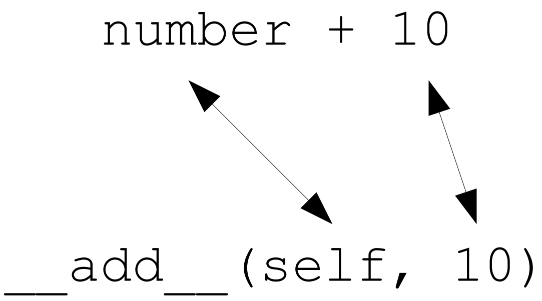
Overloading

Special functions that are not intended to be called directly

```
class X:
      def __init__(self, n):
          self.n = n

      def __add__(self, other):
          print "Doing some addition?"
          return (self.n + other)
```

```
python -i overload.py
>>> number = X(42)
>>> number+10
Doing some addition?
52
>>>
```



number + 10

__add__(self, 10)

Methods of the form __X__ can be seen as special hooks

Dynamic Dispatch

- What happen when combining Inheritance and Overloading?

```
class Y(X):
      def __add__(self, other):
          print "It is in fact an addition!"
          return (self.n + other)
```

```
python -i overload.py
>>> number = Y(42)
>>> number + 10
It is in fact an addition!
52
>>>
```

At this point, Python decides to call the most specific class

Decorators

- It allows to insert code (wrappers) into functions and classes definitions
- It allows to modularly augment functionality
- From a functional perspective, they are just high order functions! (with some differences)

High Order Functions

```
def debug(func):
    def inner  (*args):
        for a in args:
            print "The received arguments are:"
            print a

        result = func (*args)
        print "The result is:" , result

    return inner

def id(x):
    return x
```

```
python -i decorators.py
>>> id(1)
1
>>> id_debug = debug(id)
>>> id_debug(1)
The received arguments are:
1
The result is: 1
>>>
```

Decorators

```
def debug(func):
    def inner (*args):
        for a in args:
            print "The received arguments are:"
            print a

        result = func(*args)
        print "The result is:", result

    return inner

@debug
def id(x):
    return x
```

Decorator

```
python -i decorators2.py
>>> id(1)
The received arguments are:
1
The result is: 1
>>>
```

This is equivalent to:

```
def id(x):
    return x

id = debug(id)
```

More about Python?

- It is lot of fun programming with it
- If you are functional programmer, you will probably use Python differently from regular Python programmers
- Great opportunity to take functional programming results into Python!