A Brief Introduction to Python by Examples

Stefano Benedettini

DEIS - Dipartimento di Elettronica Informatica e Sistemistica
University Bologna, Second Faculty of Engineering

Artificial Intelligence 2010

Outline

Topics

- Python language basics

Objectives

- Python fundamentals
- Functional Programming Python
- Python for Object-Oriented Programming
- Leveraging Python for Artificial Intelligence

Method

- This lesson is a guided tour into the language
  - Think of it as a collective interactive tutorial
- Brief introduction and then on with tutorials and exercises
  - They help to familiarize with the language
  - Not very AI oriented, but we'll have time for that...
- We will overlook a lot of details
  - For those, refer to the documentation
  - We will introduce some should the need arise

Outline

What Is Python

Python features

- Python is:
  1. a *scripting* language
  2. interpreted
  3. multi-paradigm
     - Imperative
     - Functional
     - Object-Oriented
     - Metaprogramming
  4. strongly typed
  5. dynamically typed
  6. garbage collected

What we will use Python for

- Rapid algorithm development
- Statistical algorithm analysis

Tools of the Trade

Which version?

- Two major versions are available:
  1. `Python 2.6.x`
  2. `Python 3.1.x`
- Version 3 introduces non backward-compatible changes
- We will stick with version 2.6
  - But many things still apply to 3

Which tools?

What: `Python` interpreter

Where: Documentation

How: Any text editor (with a bit of support)

- An Eclipse plug-in is available
- Several specific IDEs as well

References

- Official site: www.python.org
- Standard library documentation:
  - Also home of a tutorial
  - Contains language specifications for the most curious
- Introductory tutorials:
  - Dive Into Python for version 2.6.x and 3.1.x (they are both worth a reading)
  - Ever heard about Google?
- Idiomatic Python: a collection of programming idioms and techniques for a productive and fun Python experience
  - *Warmly recommended*
- Learning Python, 4^{th} edition

Python Appetizer

- `print` statement to...print things!

```
>>> print "Hello world!", 10, 3.14, []
Hello World 10 3.14 []
```

- Variables are just names
- Comments start with # and span to the end of line

```
x = 10
print x # prints 10
x = "some string"
print x # prints some string
```

- All kind of operators supported

```
x, y = 3, 4
z = x^2 + y^2
print z + z # prints 10
```

Outline

Numbers and Strings

Numbers

- Python natively supports:

int	Integers
float	Double-precision floating point
complex	Complex numbers of the form: $\text{Re} + \text{Im}\, j$

```
10 + 1j, 4j, 30 + 0j # but not 10 + j
```

- Type name is used as a function for conversions
- Python never converts on your behalf

Strings

- Enclosed in single or double quotes
- Also a triple quoted form that can span multiple lines
- Many built-in methods
- Subscript access to single characters

Lists and Sequences in General

Python lists

- Linear sequence of mutable object references
- Indexed from 0, subscript notation
- Also allow for slicing:

```
l = [1, 2, 3, 4, 5, 6]
x = l[1:4] # x is [2, 3, 4]
```

Python sequences

- A sequence is not an object but an object interface (called protocol in Python parlance)
- Sequences are iterable objects (method `iter`)
- Many more methods defined, like:
  - Membership
  - Slice
  - Concatenation
- Look them up here

Dictionaries

- Unordered mutable collections of key-value pairs
  - Key can be any immutable object (actually, it must be hashable)
  - Value can be anything
- Their type is `dict` (so `help(dict)` in an interpreter to learn about their methods)

```
d = {a: 1, b: 2, c: 3, d: 1}
d.keys() # returns [a, b, c, d]
d[b] # returns 2
d.values() # returns [1, 1, 2, 3]
```

Tuples

- Python tuples are immutable sequences of object references
  - Practically, like lists, but immutable
  - Cannot be shrunk or grown
  - Can't even assign to a tuple element
- Implement sequence protocol
- Many syntactic facilities to define them

```
aTuple = (1, 2, 3)
anotherTuple = tuple([1, 2, 3])
x = 1, 2, 3 # this is a tuple too!
y = x + ("ciao",) # notice the singleton tuple
    syntax
print y # prints (1, 2, 3, "ciao")
```

Outline

Type System by Slogans!

Everything is an Object

- Python is “more” object-oriented than Java
- Really everything is an object
  - Numbers, strings, booleans, . . . any datatype
  - Functions and classes
  - Code, stack-traces (OK, forget I ever wrote that!)

Example

A Prompt Session

```
>>> print "hello world".upper()
HELLO WORLD
>>> print 2 > 4
False
>>> print 3 + 4, (3).__add__(4)
7 7
```

Operators are just syntactic sugar for method invocation

Type System by Slogans!

Everything is an Object

- Python is “more” object-oriented than Java
- Really everything is an object
  - Numbers, strings, booleans, . . . any datatype
  - Functions and classes
  - Code, stack-traces (OK, forget I ever wrote that!)

Example

A Prompt Session

```
>>> print "hello world".upper()
HELLO WORLD
>>> print 2 > 4
False
>>> print 3 + 4, (3).__add__(4)
7 7
```

Operators are just syntactic sugar for method invocation

Type System by Slogans!

Strong Type System

- Will not allow not well-defined operations
- Almost never Python will coerce an object for you

Example

```
>>> print "3" + 4 # this is an error
>>> print "3" + str(4) # explicit casting
34
>>> print True + 3 # ''booleans are integers
4
```

There are no booleans in Python: True and False are just constants with value 1 and 0 respectively.

Type System by Slogans!

Strong Type System

- Will not allow not well-defined operations
- Almost never `Python` will coerce an object for you

Example

```
>>> print "3" + 4 # this is an error
>>> print "3" + str(4) # explicit casting
34
>>> print True + 3 # ''booleans are integers
4
```

There are no booleans in `Python`: `True` and `False` are just constants with value 1 and 0 respectively.

Python Type System

Objects in a nutshell

1. Objects are dictionaries, i.e., name-value pairs
2. Objects are (usually mutable) collections of names (also called Namespace Objects)
3. Every attribute access corresponds to a dynamic dictionary look-up
4. Access is performed via dotted notation

Variables and everything else

- Variables have no type annotation
  - A variable is just a reference to an object
  - It has no connection to underlying memory representation
- Functions and methods have no type annotations as well

Outline

Blocks and Control Structures

The Layout Rule

- A block is a textual element which has the following structure:

```
header:
  body # watch the indentation!
```

- The indentation is mandatory and must be consistent for the whole block body
- Python many common control structures
  - `if ... else`
  - `while`
  - `for`, but this is special and much more powerful than its C/Java counterpart
- no `switch/case` or `do/while`
- `break` and `continue` work as usual

Blocks and Control Structures

Example

```
count = 10
while count > 0:
  if count % 2 == 0:
    print count, "is even"
  elif: count % 3 == 0:
    print count, "is divisible by 3"
  else:
    print count
  count -= 1 # augmented assignment but no -- or ++
     operators
```

for Cycle

for statement and iterables

- for statements have this form:

```
for iterationVariables in iterable:
  body
```

- iterable is any object that supports *Iterable* protocol
  - Very similar to Java
  - Lists, tuples, dictionaries, sets
  - Must return an *Iterator* when passed to iter function
- body is executed once for every object returned by the iterator
- Iteration variables can “unpack” objects returned by the iterator
- Look over here for some idiomatic looping techniques
- Think of it as the Python equivalent of the Enhanced For Loop in Java or a for...in loop in JavaScript

for Cycle

Example

Iterations

```
for x in range(0, 7): # like for x in [1, 2, 3, 4, 5,
    6]
 print x * x
for x in "hello, world!":
 print x.upper() # prints "hello, world!" capitalized
     one character for each line
```

Example

Unpacking

```
for a, b in [(1, 2), (10, 20), (-1, 7)]:
 print a * b # prints 2 200 -7
for ch, i in zip("hello", range(0, 5)): # see also
   enumerate function
 print i, "=>", ch # prints a character and its index
```

The Peculiar Nature of Assignment

Scopes: an introduction

- A scope is a textual region in the code
- A scope determines the meaning of all unqualified names
- In `Python` only modules, functions and classes define scopes
  - Not control blocks
  - For example, the iteration variable in a `for` loop is available *after* the loop

Bindings

- A binding is a runtime association between a name and an object
- Assignment creates bindings
- Not only assignment operator creates bindings; also:
  - Class and function declarations
  - Module import
  - Anything that introduces a name (like iteration variables in `for` loops)

The Peculiar Nature of Assignment

Final remarks

1. In Python everything is executable code
   - Even what may you seem like a declarations
2. Variables come into existence when assigned

Example

```
for x in range(0, 10):
  pass # "do-nothing" statement
print x * x # prints 81
fun = "ciao"
print fun # prints ciao
def fun(): # function declaration
  pass
print fun # prints <function f at ...>
```

Functions and Function Declaration

Functions basics

- Functions are first class citizen, i.e. objects
- Function declaration:

```
def function_name(function_arguments):
  body
```

- Many possibilities for specifying function arguments
  - Default arguments
  - Named arguments
  - Packing/unpacking
- `return` keyword does what you expect
- But we will see also `yield`
- Since functions are objects they can be:
  - Passed around
  - “Renamed”
  - Queried (useful for metaprogramming geeks)

Functions and Function Declaration

Example

```
def integer_sum(fun, start=0, stop=10):
  accumulator = 0
  for x in range(start, stop):
    accumulator += fun(x)
  return accumulator

def twice(x):
  return 2 * x
twice_alias = twice # twice and twice_alias are
    simple variables!

integer_sum(twice, to=5) # start=0, returns 20
integer_sum(twice, 3, 12) # start=3, stop=12, returns
     126
integer_sum(twice_alias, 9) # start=9, stop=10,
    returns 18
```

Outline

Outline

OOP in Python

Class declaration

- OOP is no different than in Java
  - Objects, classes, instances, methods, ...
  - You also get multiple inheritance
- Some syntactic differences:

```
class AClass(object):
  def __init__(self, x, y):
    self.x = x
    self.other = y
  def some_method(self, x):
    return self.x + x

obj = AClass(10, "ciao")
```

- `self` is always explicit
- The “constructor” is the special methods `__init__`
- Invoke (I mean it) the class to build an instance

OOP Gotchas! (from a Java perspective)

- Objects are “open” (actually are dictionaries)
  - New attributes can be added to them

```
x = AClass(10, "ciao")
y = AClass(1, 2)
# x.attrib # this raises exception
y.attrib = [1, 2, 3] # now y has a new attribute
```

- Classes are object (therefore are open)
- Every assignment in class scope goes into class object

```
class C(object):
  x = 90
  def some_other_method(self): pass
print C.x # prints 90
```

- A class is like a big dictionary shared by all its instances

```
obj = C()
print obj.x # prints 90 again
```

Outline

Structural Subtyping

Subtyping *in theory*

- Subtyping is a form of type polymorphism
- Semantically, it is described by *Liskov substitution principle*:
- For every program P, if U is a subtype of T, I can replace every object of type T with objects of type U without altering program behaviour
  - Let's forget about the "*For every program*" stuff
  - It is undecidable anyway

Subtyping *for real*

- The best a compiler can do is to check names and type signatures
- It all boils down to object capabilities:
  - What can objects of a certain class do?
- It is all about object methods and attributes

Structural Subtyping

Subtyping *in theory*

- Subtyping is a form of type polymorphism
- Semantically, it is described by *Liskov substitution principle*:
- For every program P, if U is a subtype of T, I can replace every object of type T with objects of type U without altering program behaviour
  - Let's forget about the "*For every program*" stuff
  - It is undecidable anyway

Subtyping *for real*

- The best a compiler can do is to check names and type signatures
- It all boils down to object capabilities:
  - What can objects of a certain class do?
- It is all about object methods and attributes

Structural Subtyping

How many kinds of subtyping?

Nominal: subtype relations are explicitly stated by the programmer

Structural: subtype relation depends on object structure

What you already know

- In Java subtyping is nominal
- Programmer explicitly:
  - Extends a class
  - Implements an interface
- That way a hierarchy of types is defined
- Drawback: nominal subtyping is not “retroactive”

Structural Subtyping

How many kinds of subtyping?

Nominal: subtype relations are explicitly stated by the programmer

Structural: subtype relation depends on object structure

What you already know

- In Java subtyping is nominal
- Programmer explicitly:
  - Extends a class
  - Implements an interface
- That way a hierarchy of types is defined
- Drawback: nominal subtyping is not “retroactive”

Structural Subtyping

Example

- Suppose that I write an interface I_F to be used inside my framework
- Any new class that is supposed to be plugged into the framework will implement I_F
- What about all those classes that still respect I_F but were written before I_F was?
- The most favourite sport of OO programmers: wrapping things $\Rightarrow$ Adapter pattern

Structural Subtyping

Example

```
interface IStream {
  void write(String s);
  void close();
}

class Framework {
  void setup(IStream s){
    s.write("Ready");
    doStuff();
    s.close()
  }
}
```

```
class FrameworkLogger
    implements IStream {
  void write(String s) {
      ... }
  void close() { ... }
}
```

- `java.io.PrintWriter` class has exactly the same methods required by `IStream`
- But I cannot use it inside my framework
- I have to *adapt* it, even though it is useless!

Structural Subtyping

- In Structural Subtyping the interface of an object determines subtype relation
  - Only method signature counts
- In the preceding example both `FrameworkLogger` and `PrintWriter` classes would do
- Any object with `write(String)` and `close` methods could be provided to `setup`
- Structural subtyping is retroactive in this sense
- Available in many languages with different implementations
  - Scala
  - Haskell
  - OCaml
- Despite what you may think, it *can* be checked at compile time

Duck Typing

- Python type system is even more permissive
- Only the runtime interface of an object matters
- Whenever you access an object's attribute (method or member field) Python looks up in a dictionary
  - If the attribute is found it is returned
  - Otherwise an exception is raised
- Drawback: types are not checked at compile time!
  - You can catch type errors only at runtime
  - There is no compilation stage anyway

A Brief Introduction to Python by Examples

Stefano Benedettini

DEIS - Dipartimento di Elettronica Informatica e Sistemistica
University Bologna, Second Faculty of Engineering

Artificial Intelligence 2010