Open Source Frameworks for Rapid Application Development

Marek Krętowski
Krzysztof Bandurski, Tomasz Łukaszuk, Tomasz Rybak

Software Departament
Faculty of Computer Science
Bialystok University of Technology

m.kretowski@pb.edu.pl
k.bandurski@pb.edu.pl, t.lukaszuk@pb.edu.pl, t.rybak@pb.edu.pl

Lecture topic

Python

Python: Table of content

1. What is Python?
   - Programming philosophy
   - Properties
   - Uses
2. Basics
   - Coding style
   - Native datatypes
   - The power of introspection
3. Functions and classes
   - Functions
   - Classes
4. Control flow
5. Advanced features
   - Decorators
   - Iterators and Generators

What is Python?

History

- Conceived in the late 1980 by Guido Van Rossum at Centrum Wiskunde & Informatica in Netherlands. Named after 'Monty Python's Flying Circus'.
- 16 October 2000: the release of Python 2.0
- 3 December 2008: the release of Python 3.0. Backward incompatible with Python 2.0, but many features have been backported to Python 2.6
- 2.x and 3.x releases are now in simultaneous develpment
- 27 November 2010: the release of Python 2.7.1 and 3.1.3

The Zen of Python I

. . . a.k.a. PEP 20 (PEP = Python Enhancement Proposal)

1. Beautiful is better than ugly.
2. Explicit is better than implicit.
3. Simple is better than complex.
4. Complex is better than complicated.
5. Flat is better than nested.
6. Sparse is better than dense.
7. Readability counts.
8. Special cases aren't special enough to break the rules.
9. Although practicality beats purity.
10. Errors should never pass silently.
11. Unless explicitly silenced.
12. In the face of ambiguity, refuse the temptation to guess.

The Zen of Python II

13. There should be one– and preferably only one –obvious way to do it.
14. Although that way may not be obvious at first unless you're Dutch.
15. Now is better than never.
16. Although never is often better than *right* now.
17. If the implementation is hard to explain, it's a bad idea.
18. If the implementation is easy to explain, it may be a good idea.
19. Namespaces are one honking great idea – let's do more of those!

General I

- Interpreted
- Uses byte-code (*.pyc and *.pyo files)
- EVERYTHING IS AN OBJECT
- Very clear, readable syntax
- Modules, classes, functions
- Full modularity, supporting hierarchical packages
- Multi-paradigm: object-oriented and structured programming + a number of other language features: fuctional programming, aspect-oriented programming
- Dynamic and strong typing, polymorphism, garbage collector, late binding

General II

- duck typing: 'when I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck.' - James Whitcomb Riley
- Operator overloading
- Indentation for block structure
- Strong introspection capabilities
- Extensions and modules easily written in C, C++ (or Java for Jython, or .NET languages for IronPython)

High-level data types

- Numbers: int, long, float, complex
- Strings: immutable
- Basic containers: lists, dictionaries, sets (mutable), tuples (immutable)
- Other types for e.g. binary data, regular expressions, introspection
- Extension modules can define new 'built in' datatypes

Why use it? I

- Reduced development time
- Very clear, readable syntax (program maintenance)
- Very easy to learn
- Quite fast! (Fractal benchmark, http://www.timestretch.com/FractalBenchmark.html)

Language	exec. time (s)	slowdown
C gcc-4.0.1	0.05	1.00x
Java 1.4.2	0.40	8.00x
Python 2.5.1	9.99	199.80x
Perl 5.8.6 optimized	12.37	247.34x
PHP 5.1.4	23.13	462.40x
Javascript Spider Monkey v1.6	31.06	621.27x
Ruby 1.8.4	34.31	686.18x

Why use it? II

- Extensive standard libraries and third party modules for virtually every task
- Runs everywhere (Windows, Linux/Unix, Mac, Amiga)
- Open and free!

What is it used for?

- rapid prototyping
- scripting (including web scripting)
- throw-away, ad hoc programming
- steering scientific applications
- XML processing
- database applications
- GUI applications

Who uses it?

- YouTube.com
- Industrial Light & Magic
- Google
- RedHat (installation tools)
- EveOnline (a massive multiplayer game)
- EZTrip.com
- Firaxis Games (Sid Meier's Civilization)

Basics

First Python program

```
1  def buildConnectionString(params):
2      """Build a connection string from a dictionary of parameters.
3      Returns string."""
4
5      return ";".join(["%s=%s" % (k, v) for k, v in params.items()])
6
7  if __name__ == "__main__":
8      myParams = {"server":"mpilgrim",
9                  "database":"master",
10                 "uid":"sa",
11                 "pwd":"secret",
12                 }
13     print buildConnectionString(myParams)
```

Output:

```
server=mpilgrim;uid=sa;database=master;pwd=secret
```

- This example shows basic uses of functions, strings, tuples, lists and dictionaries
- Note the code indentation!

Everything is an object

```
1  def buildConnectionString(params):
2      """Build a connection string from a dictionary of parameters.
3      Returns string."""
4
5      return ";".join(["%s=%s" % (k, v) for k, v in params.items()])
```

```
>>> import odbchelper
>>> params = {"server":"mpilgrim", "database":"master",
... "uid":"sa", "pwd":"secret"}
>>> print odbchelper.buildConnectionString(params)
server=mpilgrim;uid=sa;database=master;pwd=secret
>>> print odbchelper.buildConnectionString.__doc__
Build a connection string from a dictionary
Returns string.
```

- Note how we access the `__doc__` string - a function is also an object!

Dictionaries

```
>>> d = {"server":"mpilgrim", "database":"master"}
>>> d
{'server': 'mpilgrim', 'database': 'master'}
>>> d["server"]
'mpilgrim'
>>> d["mpilgrim"]
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
KeyError: mpilgrim
>>> d[24] = 6666
>>> d
{'server': 'mpilgrim', 'database': 'master', 24 : 666}
>>> del d['server']
>>> d
{'database': 'master', 24 : 666}
```

- Dictionary keys and values can be virtually anything (nesting!)
- Dictionaries ARE NOT ORDERED

Lists

```
>>> li = ["a", "b", "mpilgrim", "z", "example"]
>>> li
['a', 'b', 'mpilgrim', 'z', 'example']
>>> li[4]
'example'
>>> li[-1]
'example'
>>> li[1:3]
['b', 'mpilgrim']
>>> li.append("new")
>>> li
['a', 'b', 'mpilgrim', 'z', 'example', 'new']
>>> li = ['a', 'b', 'mpilgrim']
>>> li = li + ['example', 'new']
>>> li
['a', 'b', 'mpilgrim', 'example', 'new']
```

- List elements can also be anything (nesting!)
- Lists ARE ORDERED
- More list methods: `remove`, `push`, `pop`, `index` ...

List comprehensions

```
1  def buildConnectionString(params):
2      """Build a connection string from a dictionary of parameters.
3      Returns string."""
4
5      return ";".join(["%s=%s" % (k, v) for k, v in params.items()])
```

- Provide a concise way to create lists.
- Each list comprehension consists of an expression followed by a `for` clause, then zero or more `for` or `if` clauses.

```
>>> vec1 = [2, 4, 6]
>>> vec2 = [4, 3, -9]
>>> [(x, x**2) for x in vec1]
[(2, 4), (4, 16), (6, 36)]
>>> [x*y for x in vec1 for y in vec2]
[8, 6, -18, 16, 12, -36, 24, 18, -54]
>>> [x+y for x in vec1 for y in vec2]
[6, 5, -7, 8, 7, -5, 10, 9, -3]
>>> [vec1[i]*vec2[i] for i in range(len(vec1))]
[8, 12, -54]
```

Tuples

```
>>> t = ("a", "b", "mpilgrim", "z", "example")
>>> t
('a', 'b', 'mpilgrim', 'z', 'example')
>>> t[0]
'a'
>>> t[-1]
'example'
>>> t[1:3]
('b', 'mpilgrim')
>>> t.append("new")
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
AttributeError: 'tuple' object has no attribute 'append'
>>> "z" in t
True
```

- Tuple elements can also be anything (nesting!)
- Tuples ARE ORDERED
- Tuples HAVE NO METHODS and are immutable...
- ...but tuple elements can be!

Sets

```
>>> s1 = set(['one', 'two', 'three'])
>>> s2 = set(['two', 'three', 4])
>>> s1 | s2
set([4, 'two', 'three', 'one'])
>>> s1 ^ s2
set([4, 'one'])
>>> s1 & s2
set(['two', 'three'])
>>> s1 - s2
set(['one'])
```

- A set object is an unordered collection of immutable values
- Useful for membership testing, removing duplicates from a sequence, and computing mathematical operations such as intersection, union, difference, and symmetric difference.

Declaring variables

```
>>> x
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
NameError: There is no variable named 'x'
>>> x = 1
>>> x
1
>>> range(7)
[0, 1, 2, 3, 4, 5, 6]
>>> (MO, TUE, WED, THU, FRI, SAT, SUN) = range(7)
>>> MO
0
>>> TUE
1
>>> SUN
6
```

- You can't declare a variable without assigning it a value
- You can assign multiple values at once

Formatting strings I

```
>>> k = "uid"
>>> v = "sa"
>>> "%s=%s" % (k, v)
'uid=sa'
>>> uid = "sa"
>>> pwd = "secret"
>>> print pwd + " is not a good password for " + uid
secret is not a good password for sa
>>> print "%s is not a good password for %s" % (pwd, uid)
secret is not a good password for sa
>>> userCount = 6
>>> print "Users connected: %d" % (userCount, )
Users connected: 6
>>> print "Users connected: " + userCount
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
TypeError: cannot concatenate 'str' and 'int' objects
```

- Strong typing: you can't add an integer to a string!

Formatting strings II

```
>>> d = { 'pwd' : 'secret', 'uid' : 'sa' }
>>> "%(uid)s=%(pwd)s # user id: %(uid)s, password: %(pwd)s" % d
'sa=secret # user id: sa, password: secret'
```

- You can also format strings using dictionaries (keyword arguments)
- This allows you to use each argument many times (or not use it at all)

```
>>> params = {"server":"mpilgrim", "database":"master",
... "uid":"sa", "pwd":"secret"}
>>> ["%s=%s" % (k, v) for k, v in params.items()]
['server=mpilgrim', 'uid=sa', 'database=master', 'pwd=secret']
>>> ";".join(["%s=%s" % (k, v) for k, v in params.items()])
'server=mpilgrim;uid=sa;database=master;pwd=secret'
```

- Strings are also objects - like everything else!

Everything is an object

```
>>> s = "abc"
>>> dir(s)
['__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']
```

- Did I mention that everything is an object. . . ?
- NOTE: Double underscores are only a convention

`type`, `str` and other built-in functions

```
>>> type(1)
<type 'int'>
>>> li = []
>>> type(li)
<type 'list'>
>>> import odbchelper
>>> type(odbchelper)
<type 'module'>
>>> import types
>>> type(odbchelper) == types.ModuleType
True
>>> str(1)
'1'
>>> horsemen = ['war', 'pestilence', 'famine']
>>> horsemen
['war', 'pestilence', 'famine']
>>> str(horsemen)
"['war', 'pestilence', 'famine']"
>>> str(odbchelper)
"<module 'odbchelper' from 'c:\\docbook\\dip\\py\\odbchelper.py'>"
```

- Other introspecting built-ins: `callable`, `isinstance`, `getattr`

Functions and classes

Optional and named arguments

```
def info(object, spacing=10, collapse=1):
    #function body...
```

Valid calls:

```
info(odbchelper)                        #1
info(odbchelper, 12)                    #2
info(odbchelper, collapse=0)            #3
info(spacing=15, object=odbchelper)     #4
```

1. With only one argument, `spacing` gets its default value of 10 and `collapse` gets its default value of 1.
2. With two arguments, `collapse` gets its default value of 1.
3. Here you are naming the `collapse` argument explicitly and specifying its value. `spacing` still gets its default value of 10.
4. Even required arguments (like `object`, which has no default value) can be named, and named arguments can appear in any order.

*args and **kwargs

```
1  def foo(hello, *args, **kwargs):
2      print hello
3      print 'arguments:'
4      for each in args:
5          print 'arg: %s' % each
6      print 'keyword arguments:'
7      for each in kwargs:
8          print 'kwargs: %s=%s' % (each, kwargs[each])
9
10 if __name__ == '__main__':
11     foo('LOVE', 'one', 'two', kwarg1='three', kwarg2='four')
```

Result:

```
LOVE
arguments:
arg: one
arg: two
keyword arguments:
kwargs: kwarg1=three
kwargs: kwarg2=four
```

Classic classes

```
>>> class ClassicClass:
...     pass
...
>>> dir(ClassicClass)
['__doc__', '__module__']
```

- A class can implement certain operations that are invoked by special syntax (such as arithmetic operations or subscripting and slicing) by defining methods with special names (e.g. `__getitem__`). This allows classes to define their own behavior with respect to language operators.
- Note the `__doc__` attribute.
- Classes are also objects!

New-style classes

```
>>> class NewStyleClass(object):
...     pass
...
>>> dir(NewStyleClass)
['__class__', '__delattr__', '__dict__', '__doc__', '__format__', '
    __getattribute__', '__hash__', '__init__', '__module__', '__new__', '
    __reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '
    __str__', '__subclasshook__', '__weakref__']
```

- Introduced in Python 2.2 to unify types and classes.
- Always inherit directly either from `object` or a built-in type
- In Python 3.x the explicit base class is not required (because everything will subclass `object`).
- Django uses mostly new-style classes.

Class vs. class instance

```
>>> class A(object):
...     name = 'class A'
...
>>> a = A()
>>> a.name
'class A'
>>> a.instance_name = 'instance a'
>>> a.instance_name
'instance a'
>>> a.name = 'instance a'
>>> A.name
'class A'
>>> a.name
'instance a'
```

- Class attributes serve as default values for class instance attributes
- Class instances can have more attributes than their respective classes!

Methods

```
class Advanced(object):
    def __init__(self, name):
        self.name = name
    def Description():
        return 'This is an advanced class.'
    Description = staticmethod(Description)
    def ClassDescription(cls):
        return 'This is advanced class: %s' % repr(cls)
    ClassDescription = classmethod(ClassDescription)
```

- Three types of methods:
  - Instance methods (`__init__`)
  - Static methods (`Description`)
  - Class methods (`ClassDescription`)
- Class methods and static methods require decorators (to be presented later on)

Methods

- Instance methods always receive the instance as the first argument (can acces all the attributes of both the instance and the class).
- Class methods always receive the class as the first argument (can access all the attribute of the class only)
- Static methods do not receive either the instance or the class (cannot access either)
- Class methods may be called using either an instance or a class

Control flow

Truth value testing

- Any object can be tested for truth value, for use in an `if` or `while` condition or as operand of the Boolean operations. The following values are considered false:
  - `None`
  - `False`
  - zero of any numeric type, e.g. `0, 0L, 0.0, 0j`
  - any empty sequence, e.g. `'', (), []`
  - any empty mapping, e.g. `{}`
  - instances of user-defined classes, if the class defines a `__nonzero__()` or `__len__()` method, when that method returns the integer zero or bool value `False`.
- All other values are considered true

Boolean operations

Operation	Result
x `or` y	if *x* is false, then *y*, else *x*
x `and` y	if *x* is false, then *x*, else *y*
`not` x	if *x* is false, then `True`, else `False`

- `or` is a short-circuit operator, evaluates the second argument only if the first one is `False`
- `and` is a short-circuit operator, evaluates the second argument only if the first one is `True`
- Note that `or`/`and` operations do not have to return only `True` or `False`

`if` statements

```
>>> x = int(raw_input("Please enter an integer: "))
Please enter an integer: 42
>>> if x < 0:
...      x = 0
...      print 'Negative changed to zero'
... elif x == 0:
...      print 'Zero'
... elif x == 1:
...      print 'Single'
... else:
...      print 'More'
...
More
```

- There can be zero or more `elif` parts
- The `else` part is optional

`while` statements

```
>>> a, b = 0, 1
>>> while b < 1000:
...     print b,
...     a, b = b, a+b
...
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
```

- Nothing new here...
- A trailing comma avoids the newline after the output.

`for` statements

```
>>> # Measure some strings:
... a = ['cat', 'window', 'defenestrate']
>>> for x in a:
...     print x, len(x)
...
cat 3
window 6
defenestrate 12
```

- Rather than always iterating over an arithmetic progression of numbers (like in Pascal), or giving the user the ability to define both the iteration step and halting condition (as C), PythonâTMs `for` statement iterates over the items of any sequence (a list or a string), in the order that they appear in the sequence.
- It is not safe to modify the sequence being iterated over in the loop (this can only happen for mutable sequence types, such as lists).

Exceptions and exception handling

```
for arg in sys.argv[1:]:
    try:
        f = open(arg, 'r')
    except IOError:
        print 'cannot open', arg
    else:
        print arg, 'has', len(f.readlines()), 'lines'
        f.close()
    finally:
        print "Finished with the file"
```

- A `try` statement may have more than one except clause
- If an exception occurs during execution of the `try` clause, the rest of the clause is skipped, the respective `except` clause is executed, and then execution continues after the `try` statement.
- The `try ... except` statement has an optional `else` clause, which, when present, must follow all except clauses. It is useful for code that must be executed if the try clause does not raise an exception.
- The `finally` clause is executed no matter what happens (even

Advanced features

Decorators: the concept

A decorator is usually a function that transforms another function:

```
1  def decorator_function(target):
2      # Do something with the target function
3      target.attribute = 1
4      return f
5
6  def target(a,b):
7      return a + b
8
9  #This is how we apply the decorator
10 target = decorator_function(target)
```

In this simple case, the decorator just adds an attribute to the function being decorated:

```
>>> target(1,2)
3
>>> target.attribute
1
```

Why use decorators?

- We need to change functions and methods: to add synchronisation, to add logging, etc
- This was possible before, but changes had to be made in places other than the declaration and could be hard to find later on
- It is reasonable to group them with the declaration of a function
- Since Python 2.2 two decorators were added: classmethod and staticmethod
- Adding syntax has been difficult, as this should be rather simple, not scarying for newcommers, give clear intent, and be clearly visible
- Java style decorators (decorator) were added in Python 2.4a2

Java-like syntax

Since Python 2.4a2, the code shown earlier can be rewritten as follows:

```
1  def decorator_function(target):
2      # Do something with the target function
3      target.attribute = 1
4      return target
5
6  # Here is the decorator, with the syntax '@function_name'
7  @decorator_function
8  def target(a,b):
9      return a + b
```

- Note that decorator functions are called when they are applied, not when the decorated function is called
- Decrators can do many things (conditional function calling, transforming arguments), but they are not really different from what you've seen above

Wrapper functions

```
1  def decorator(target):
2
3      def wrapper():
4          print 'Calling function "%s"' % target.__name__
5          return target()
6
7      # Now we need to assign the atttribute to the *wrapper function*.
8      wrapper.attribute = 1
9      return wrapper
10
11 @decorator
12 def target():
13     print 'I am the target function'
```

```
>>> target()
Calling function "target"
I am the target function

>>> target.attribute
1
```

- The wrapper function can do whatever it wants to the target function

Decorators for functions that accept arguments

```
1  def decorator(target):
2
3      def wrapper(*args, **kwargs):
4          kwargs.update({'debug': True}) # Edit the keyword arguments
5          print 'Calling "%s" in debug mode' % target.__name__
6          return target(*args, **kwargs)
7
8      wrapper.attribute = 1
9      return wrapper
10
11 @decorator
12 def target(a, b, debug=False):
13     if debug: print '[Debug] I am the target function'
14     return a+b
```

```
>>> target(1,2)
Calling "target" in debug mode
[Debug] I am the target function
3

>>> target.attribute
1
```

Decorators for instance methods

A decorator for a class method should assume that the first argument is always `self`:

```
1   def wrapper(self, *args, **kwargs):
2       # Do something with 'self'
3       print self
4       return target(self, *args, **kwargs)
```

Needless to say, an instance method can also be used as a decorator.

Decorators that accept arguments

```
1  def options(value):
2      def decorator(target):
3          # Do something with the target function
4          target.attribute = value
5          return target
6      return decorator
7
8  @options('value')
9  def target(a,b):
10     return a + b
```

```
>>> target(1,2)
>>> target(1,2)
3

>>> target.attribute
'value'
```

Since the `decorator` function is defined inside the `options` function, it has access to any of the arguments passed to `options`.

Problem with lists

```
1  def pow2(upto):
2      """ Returns a list of powers of two """
3      powers = []
4      startat = 1
5      startpower = 1
6
7      while startpower <= upto:
8          powers.append(startat)
9          startat *= 2
10         startpower += 1
11
12     return powers
13
14 for vv in pow2(10):
15     print vv
```

- The list is stored entirely in the memory
- Whant if we wanted to iterate over 10000000 (i.e., A LOT OF) powers of two? (Python's integer has infinite precision, btw.)
- What if `pow2` was much more complex and we wanted to use it in many different places of code?

Understanding the `for` loop

- The problem can be solved with `while` loops, but Pythonists do not consider them elegant :)
- Python supports a concept of iteration over containers, implemented using two distinct mehods:
  - `container.__iter__()`: returns an `iterator` object
  - `iterator.__iter__()`: returns the `iterator` object itself
  - `iterator.next()`: return the next item from the container.
- If a user-defined class has these two methods, it can be used in `for ...in`. loops

Generators

- Python's generators provide a convenient way to implement the iterator protocol.
- Using a `yield` expression in a function definition is sufficient to cause that definition to create a generator function instead of a normal function
- `yield` returns the value (like a `return`) and suspends the execution of the function
- When a generator function is called, it returns an iterator known as a generator
- `generator.next()`: Starts the execution of a generator function or resumes it at the last executed yield statement

Generators

- Let's rewrite the `pow2` function:

```
1  def pow2(upto):
2      """ Returns a generator that will generate
3      powers of two """
4      startat = 1
5      startpower = 1
6      while startpower <= upto:
7          yield startat
8          startat *= 2
9          startpower += 1
10
11 for vv in pow2(10):
12     print vv
```

- NOTE: `return` can be used within a generator function, but only without a value. Signals the end of the sequence.

Generator expressions

Simple generators coded like list comprehensions, with parentheses instead of brackets.

```
>>> sum(i*i for i in range(10))                 # sum of squares
285

>>> xvec = [10, 20, 30]
>>> yvec = [7, 5, 3]
>>> sum(x*y for x,y in zip(xvec, yvec))         # dot product
260
```

More compact but less versatile than full generator definitions. Tend to be more memory friendly than equivalent list comprehensions.

Sources

- http://diveintopython.org/ by Mark Pilgrim
- http://www.siafoo.net/article/68 by David (?) - a nice article on decorators
- http://www.python.org
- ...