Organization

- Discussion of the meaning of ‘dynamic language’
- Case studies
- Highlighting dynamic language features and design patterns
- Brief overview of internals of Python
What's all this about dynamic languages?

- Very high-level
- Typically interpreted
- Commonly used in web development, UI design, and scripting
- Often make tradeoff between language expressiveness and execution speed
Dynamic Languages:

- Javascript
- PHP
- Ruby
- Objective C
- Python
- Actionscript
“Non-Dynamic” Languages

- Obvious examples include:
  - C
  - C++
  - C#
  - Java
  - Objective C
Wait, hold on...

- Why is Objective C listed twice?
What is the Definition of ‘Dynamic Language’?

- There doesn’t seem to be any widely accepted definition
- Categorization by features
  - A dynamic language is a language that exhibits dynamic features
  - A somewhat circular definition
- Languages can belong to both
Case Study: C++

- Why pick on C++?
  - Mature language
  - Large set of language features
  - Object Oriented
  - Popular
Compile-time Information

- A lot of information is available at compile time:
  - Type and number of functions
  - Class definitions
  - Variable Declarations
  - Exception types
    - These might not be obvious at write-time...
C++ is Still Flexible

- C++ is not particularly hampered by the requirement for compile-time certainty
- Good software engineering practices allow for scalable codebases
  - Generic code
  - Reusable code
- Compile-time checking is very valuable as an error-detection tool
Example: Template Functions

template <class T>
T getMax(T op1, T op2) {
    if (op1 > op2) {
        return op1;
    } else {
        return op2;
    }
}

• What is the type of T?

• When do we find out?

• What if the comparison operator isn’t supported for T?
Binding

- In the previous example, the type of T is *bound* at compile time:
  - `int max = getMax<int>(a, b);`
- By execution time, the type of T is set.
- In the event unsupported operations, the compiler will notify us.
- Why does C++ choose to bind so soon?
Another Example: Functions

- Breadth-First Search allows for a ‘visitor’ function
- How to pass this function into an algorithm method?
  - At runtime: Pass a function pointer
  - At compile time: Use a template
Functions as First-Class Objects

- Function pointers can be passed around, but the number of possible functions to point to is statically known.

- Why not create a function on the fly?

- This is strictly speaking not a ‘dynamic’ feature, but it is much more common in ‘dynamic’ languages than in ‘static’ languages.
Yet Another Example: Eval

- Build a string containing a valid C++ snippet, compile it at runtime, and execute it
Nope
Case Study: Python

- Why pick on Python?
  - Mature language
  - Large set of language features
  - Object Oriented
  - Popular
  - It can get weird
Python Factsheet

• Interpreted
• Almost all decisions are deferred until runtime
• Massive standard library
• Object-oriented, imperative, somewhat functional
• Design emphasizes simplicity and readability
• Practically a metalanguage
Comparator Example

def get_max(a, b):
    return a if a > b else b

• What is the type of a and b?
• What if a or b are of different types?
• Where do we declare the return type of this function?
• When do we discover failure?
Dynamic Features

- Binding is performed at the last possible instant
- Duck type system / Dynamic type system
- Everything is a first class object: functions, classes, packages, etc.
- Very introspective
  - Python programs know a great deal about themselves
Late Binding

- Variables are names that point to objects in memory
- A name can be set to point to something else entirely
- Sort of like void pointers in C/C++
Duck Typing

• “If it walks like a duck and quacks like a duck...”

• The current set of methods and properties determines valid semantics, not its inheritance or implementation of a specific interface (thanks, Wikipedia)

• Invalid operations are caught and reported by the runtime

• “It's easier to beg forgiveness than ask permission”

• In other words, try to perform an operation and recover from failure
Duck Typing Example

```python
>>> def negate(a):
...     return -a
... >>> negate([a, b, c])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in negate
TypeError: bad operand type for unary -: 'list'
```
Why not the Comparator?

```python
def get_max(a, b):
    ... return a if a > b else b
...
>>> get_max('a', 100)
'a'
>>> get_max('a', None)
'a'
>>> get_max('a', [])
'a'
>>> get_max([], {})
[]
```
def print_something():
    print_function()

def function1():
    print 'Who’s on first?’
def function2():
    print ‘No, Hoo’s on second’

print_function = function1
print_something()
# Who’s on first
print_function = function2
print_something()
# No, Hoo’s on second
Late Binding Design Patterns

- Very useful for testing
- Can replace the behaviors of entire classes with a single assignment
First-Class Objects: Functions

- Functions can be created on the fly and returned as return values.
- Function ‘declarations’ are actually statements that create function objects in memory.
- e.g. Decorator pattern
- Can selectively augment and construct functions.
Decorator Pattern Example

def authenticated(function):
    def dummy(*args, **kwargs):
        if not kwargs['session'].authenticated():
            raise NotAuthenticatedError
        else:
            function(*args, **kwargs)
    return dummy

def post_comment(comment, session):
    # perform comment posting
    post_comment = authenticated(post_comment)
First Class Objects: Classes

- Just like functions, class ‘declarations’ are statements
- Classes can be created on the fly
- Classes can be modified freely after their creation
- Methods and members can be added and removed at will
First Class Objects: Modules

- The `import` statement executes a file and imports all top-level objects into the namespace (which is a first-class object, by the way).
- Module importing can fail: no such file, error in file execution, etc.
- Imports can be attempted and retried.
- For instance, operating system-specific packages are imported one by one until one finally doesn’t fail.
In CPython, the reference implementation, Python execution proceeds as follows:

- The module is parsed by the interpreter
- An abstract syntax tree is generated
- AST is converted to bytecodes
- Interpreter runs over those bytecodes one by one
Various Python Implementations

- Python has one reference implementation, written in C: CPython.
- There are various alternate implementations, targeting various platforms and virtual machines.
  - Examples include Jython for the JVM, and IronPython for the CLR
  - Generally community-developed open source projects
  - Black sheep of the family: PyPy
The Python Virtual Machine: CPython

- The virtual machine is the 'system' on which the python 'executable' runs.
- Stack-based architecture
- Designed for simplicity of implementation
CPython Opcodes

- Part standard instruction set architecture:
  - Arithmetic operations
  - Stack operations
  - Control Flow
  - Loading and storing
  - Function Calls
CPython Opcodes

- Part high level virtual machine language:
  - Container operations
  - Module importing
  - Namespace manipulations
  - Exception handling
  - Function and class creation
Denouement

• Hopefully this served as a very brief introduction to dynamic languages in general and Python in particular.

• The sequel will expand on Python internals, and introduce PyPy, arguably the oddest of the alternative implementations.
Thank you!