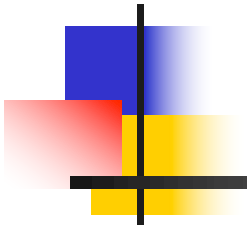


Dynamic Languages

Read: Scott, Chapter 14 (optional)

Lecture Outline

- Scripting programming languages
- Dynamic programming languages
- Taking stock: PL design choices
- Python

Scripting Languages

- E.g., Tcl, awk
- Originate in the 1970's from UNIX (shell scripts)

- Purpose
  - To process text files with ease
  - To launch components and combine (or "glue") these components into an application
- Characteristics
  - Ease of use, flexibility, rapid prototyping: hence, scripting languages are dynamically typed
  - Extensive support for text processing

References

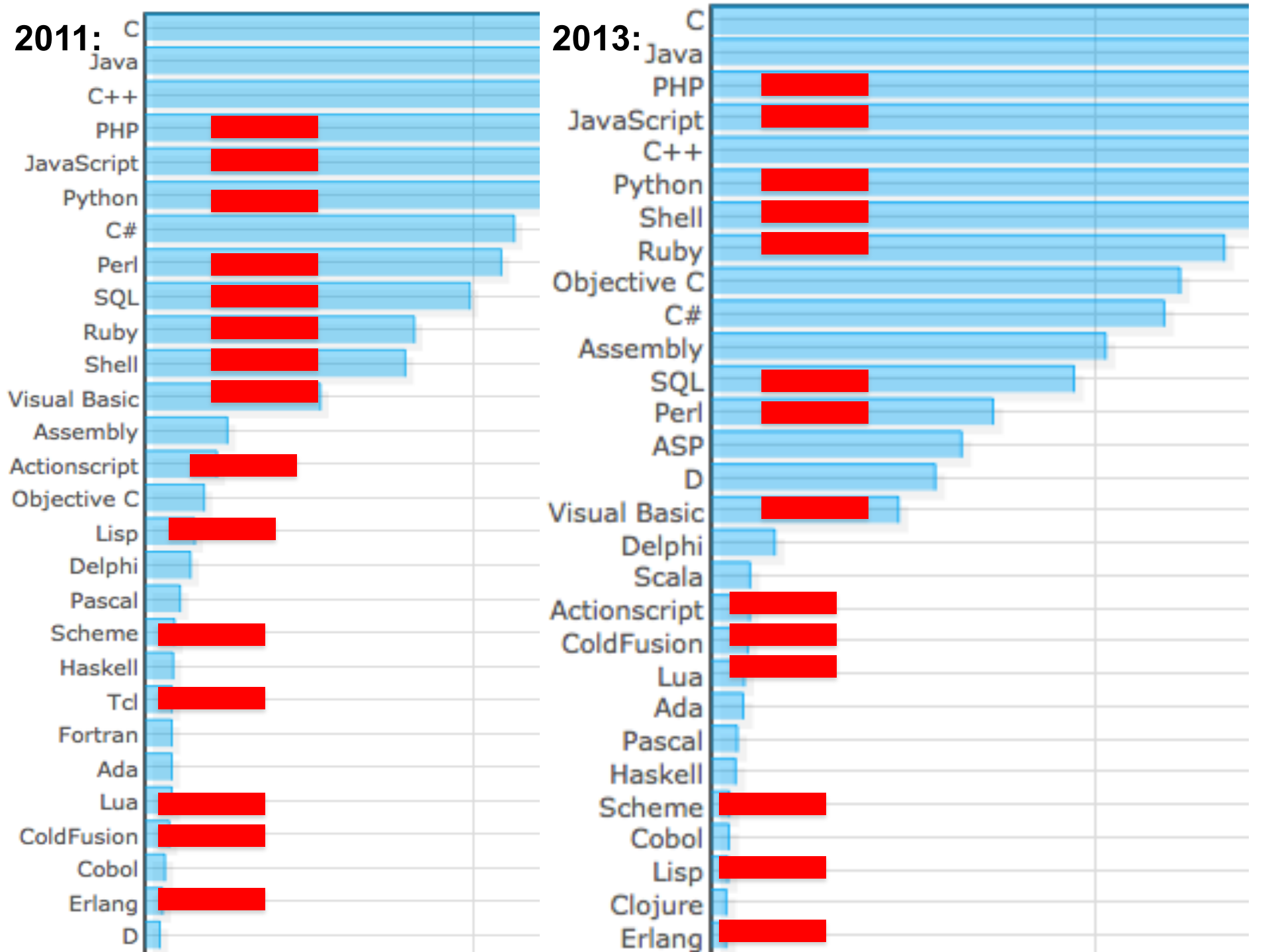
- Following slides are based on
  - “The Rise of Dynamic Languages” talk by Jan Vitek
  - “The Essence of JavaScript”, ECOOP’10 paper by Shriram Krishnamurthi et al.
  - “The Eval that Men Do”, ECOOP’11 paper by Gregor Richards et al.

2 Decades of Dynamic Languages

Visual Basic	dyn	1991
Python	dyn	1991
Lua	dyn	1993
R	dyn	1993
Java	stat+dyn	1995
JavaScript	dyn	1995
Ruby	dyn	1995
PHP	dyn	1995
C#	stat+dyn	2001
Scala	stat	2002
F#	stat	2003
Clojure	dyn	2007
Julia	dyn	2009

Last decade or so:

Go	stat	~2009
Rust	stat	2010
Swift	stat	2014
TypeScript	stat+dyn	


2011:
C
Java
C++
PHP
JavaScript
Python
C#
Perl
SQL
Ruby
Shell
Visual Basic
Assembly
Actionscript
Objective C
Lisp
Delphi
Pascal
Scheme
Haskell
Tcl
Fortran
Ada
Lua
ColdFusion
Cobol
Erlang
D
2013:
C
Java
PHP
JavaScript
C++
Python
Shell
Ruby
Objective C
C#
Assembly
SQL
Perl
ASP
D
Visual Basic
Delphi
Scala
Actionscript
ColdFusion
Lua
Ada
Pascal
Haskell
Scheme
Cobol
Lisp
Clojure
Erlang

Characteristics

- Dynamic typing, also known as Duck typing
  - Type checking amounts to running the “Duck test” at runtime:

“If it walks like a duck and swims like a duck and quacks like a duck, then it is a duck.” --- paraphrased from J. W. Riley

```
fun F( x ) {
   x.quack();
}
```

Other Characteristics

- Reference model, garbage collected
- Reflective! (i.e., eval is a prominent feature)
  - Use of eval ranges "from sensible to stupid" (ref. "The Eval that Men Do" by Richards et al.)

eval *is evil. Avoid it.*

eval *has aliases. Don't use them.*

--- Douglas Crockford

Other Characteristics

- Use of eval ranges "from sensible to stupid" (ref. "The Eval that Men Do" by Richards et al.)

```
var flashVersion = parse();

flash2Installed = flashVersion == 2;
flash3Installed = flashVersion == 3;
… // same for versions 4 to 11
for (var i = 2; i <= maxVersion; i++)
  if (eval("flash"+i+"Installed")==true)
    actualVersion = i;
```

The Eval that Men Do

The Eval That Men Do

The evil that men do lives after them; the good is oft interred with their bones.

William Shakespeare

Other Characteristics

- High-level data structures, libraries
- Lightweight syntax
- Delay error reporting, error-oblivious
  - Very difficult to trace and fix errors

- Performance challenged
  - C interpreters ~2-5 times slower than Java
  - Java interpreters ~16-43 times slower than Java

A bit on JavaScript

- 95% of all web sites use JavaScript

- Single-threaded
- Reference model, garbage collected
- Reflective! (i.e., eval is a prominent feature)
- High-level data structures, libraries
- Lightweight syntax
- Error-oblivious

Objects and Fields in JavaScript

- Objects have fields (field are also known as properties)
- Field lookup
  - `x["f"], x.f`
  - `{ "f" : 7 }["g"]`
    JavaScript is error-oblivious, in the above example, it returns undefined, and continues!
- Field update
  - `x["f"] = 2, x.f = 2`
  - `{ "f" : 0 }["g"] = 10` results in

`{ "f" : 0, "g" : 10 }`

Objects and Fields in JavaScript

- Field delete
  - `delete x.f`
  - `delete { "f" : 7, "g" : 0 }["g"]`

Arrays Are Objects

```
function sum(arr){
  var r = 0;
  for (var i=0; i<arr["length"]; i=i+1){
    r = r + arr[i]
  };
  return r
};
```

`sum([1,2,3])` yields what?

```
var a = [1,2,3,4];
delete a["3"];
```

`sum(a)` yields what?

Functions Are Objects

```
f = function(x) { return x+1 }
f.y = 90
```

`f(f.y)` yields what?

Other unexpected behavior…

- **`with`** statement
- **`eval`**

Wat?

- https://www.destroyallsoftware.com/talks/wat

Lecture Outline

- Scripting programming languages
- Dynamic programming languages
- Taking stock: PL design choices
- Python

Taking Stock: Design Choices

	Datatypes	Control-flow	Semantics/ Basic Operation
Scheme	Booleans, Numbers, Symbols, Lists	Conditional flow, Recursion	Reduction/ Function application
Java	Primitive types, Classes (library, and user-defined)	Conditional flow, Iteration	State-transition/ Assignment statement
C++	Primitive types, struct types, pointer types, array types, Classes	Conditional flow, Iteration	State-transition/ Assignment statement

Taking Stock: Design Choices

	Datatypes	Control-flow	Semantics/ Basic Operation
Haskell	ADTs, Type Classes	Conditional flow, Recursion	Reduction/ Function application
Python			

Taking Stock: Design Choices

	Variable Model	Parameter Passing Mechanism	Scoping	Typing
Scheme	Reference model	By value	Static, nested function definitions	Dynamic, type-safe
Java	Value model for simple types, reference model for class types	By sharing	Static	Static and dynamic, type-safe
C++	Value model	By value and by reference	Static	Static, type-unsafe

Taking Stock: Design Choices

	Variable Model	Parameter Passing Mechanism	Scoping	Typing
Haskell	Reference model	By name (lazy evaluation)	Static, nested function definitions	Static, type-safe
Python				

Python

- Designed by Guido van Rossum at CWI Amsterdam in 1991
- Multi-paradigm programming language
  - All characteristics of dynamic languages
  - It has “functional” features
    - E.g. higher-order functions, map and reduce on lists, list comprehensions
  - It has “object-oriented” features
    - E.g., iterators, array slicing operations, reflection, exceptions, (multiple) inheritance, dynamic loading

Python: Syntax and Scoping

Scott

```
m=1; j=3
def outer():
    def middle(k):
        def inner():
            m=4
            print (m,j,k)
        inner()
        return m,j,k
    m=2;
    return middle(j)
print (outer())
print (m,j)
```

Variable belongs to block where it is written, unless explicitly imported. What is the output of this program?

```
# new local m

# 3 element tuple
# new local m
# old (global) j
```

Python: Syntax and Scoping

Scott

```
m=1; j=3
def outer():
    def middle(k):
        def inner():
            global m
            m=4
            print (m,j,k)
        inner()
        return m,j,k
    m=2;
    return middle(j)
print (outer())
print (m,j)
```

Variable belongs to block where it is written, unless explicitly imported. What is the output of this program?

```
# from main program, not outer
```

```
# 3 element tuple
# new local m
# old (global) j
```

What's printed if we removed
the red "global m" statement?
then remove the blue assignment?

Python: Syntax and Scoping

```
m=1;
def outer():
    def middle():
        def inner():
            return m;
        inner()
        return inner
    m=2
    return middle()
fun = outer()
print (fun())
```

What is the output of this program?

```
# middle returns a closure
# { inner() { return m; }, m->2 }

# fun is a reference to closure
```

Scoping: Static Scoping Rules

- Blocks (scopes) are defined by indentation
- There are no variable declarations
  - A variable belongs to block where it is written
- "Closest enclosing scope" rule applies
  - Lookup proceeds from inner to outer blocks
  - 'global' overrides rule for access to outermost scope
- Functions are first class values
  - Static scoping entails closures and unlimited extent

Datatypes

- Numbers (+, *, **, pow, etc) - immutable
- Collections
  - Sequences
    - Strings are immutable!
    - Lists
    - Tuples are immutable!
  - Mappings
    - Dictionaries
- Files

Datatypes

- So, what model for variables does Python use?
- Reference model for variables
- Equality
  - == equal in value (structural equality, value equality)
    (Same in Scala, == redirects to equals!)
  - is same object (reference equality)
  - None acts like null in C, a placeholder for an actual value

Control Flow

- Short-circuit evaluation of boolean expressions
- Constructs for conditional control flow and iteration: **`if, while, for`**

```
for x in [“spam”, “eggs”, “ham”]:
    print x #iterates over list elements
s = “lumberjack”
for y in s: print y #iterates over chars
```

- Use of iterators defines the “Python style”

Functions

- Two forms of function definition. First class values

```
def incr(x): return x+1 #function incr
#list of 2 functions
incrs = [lambda x: x+1, lambda x: x+2]
```

- Polymorphism

```
#what datatypes can this function be used on?
def intersect(seq1, seq2):
  res = [ ]
  for x in seq1:   #iterates over elements in seq1
    if x in seq2:  #checks if element is in seq2
      res.append(x) #if so, adds element to list
  return res

print intersect([1,2,3],[2,4,6])                    #[2]
print intersect( (1,2,3,4), (4,3,5))                #[3,4]
print intersect( (1,2,3), [1,2,3])                  #[1, 2, 3]
print intersect({1:'a',2:'b',3:'c'}, {1:'a',4:'d'}) #[1]
print intersect({1:'a',2:'b',3:'c'}, {4:'a',5:'b'}) #[]
#clearly the intersection is on the keys, not the values!
```

Functions

- What is the parameter passing mechanism in Python?

- Call by value
  - But each value is a reference!
  - So we say that parameter passing is call by sharing
  - Be careful: if we pass a reference to a mutable object, callee may change argument object

Taking Stock: Python

- Datatypes?
- Control flow?
- Basic operation?
- Variable model?
- Parameter passing mechanism?
- Scoping?
  - Are functions first-class values?
- Typing?