Programming Paradigms

Lecture 18:

Dynamic Languages

Prof. Dr. Michael Pradel

Software Lab, University of Stuttgart

Winter 2019/2020

Wake-up Exercise

What does this JavaScript code print?

```
function Foo() {
    this.n = 3;
}
function bar() {
    return this.n;
}
Foo.prototype.bar = function() {
    return 7;
}
x = new Foo();
x.bar = bar;
console.log(x.bar());
```

https://ilias3.uni-stuttgart.de/vote/0ZT9

Wake-up Exercise

What does this JavaScript code print?

```
function Foo() {
    this.n = 3;
}
function bar() {
    return this.n;
}
Foo.prototype.bar = function() {
    return 7;
}
x = new Foo();
x.bar = bar;
console.log(x.bar());
```

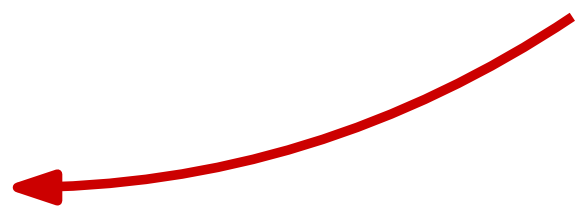
Result: 3

https://ilias3.uni-stuttgart.de/vote/0ZT9

Wake-up Exercise

What does this JavaScript code print?

```
function Foo() {
    this.n = 3;
}
function bar() {
    return this.n;
}
Foo.prototype.bar = function() {
    return 7;
}
x = new Foo();
x.bar = bar;
console.log(x.bar());
```



Object whose prototype provides a `bar` method

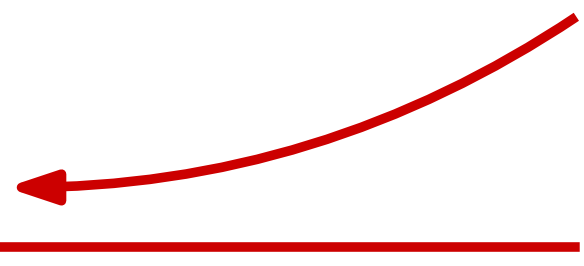
Result: 3

https://ilias3.uni-stuttgart.de/vote/0ZT9

Wake-up Exercise

What does this JavaScript code print?

```
function Foo() {
    this.n = 3;
}
function bar() {
    return this.n;
}
Foo.prototype.bar = function() {
    return 7;
}
x = new Foo();
x.bar = bar;
console.log(x.bar());
```



Object whose prototype provides a `bar` method

Overwriting the method

Result: 3

https://ilias3.uni-stuttgart.de/vote/0ZT9

Wake-up Exercise

What does this JavaScript code print?

```
function Foo() {
    this.n = 3;
}
function bar() {
    return this.n;
}
Foo.prototype.bar = function() {
    return 7;
}
x = new Foo();
x.bar = bar;
console.log(x.bar());
```

Return the `n` property of the current object

Object whose prototype provides a `bar` method

Overwriting the method

Result: 3

https://ilias3.uni-stuttgart.de/vote/0ZT9

Dynamic Languages

- **A.k.a. "scripting languages"**
  - Nowadays used for much more than "scripts"
- **Special-purpose languages**
  - Bash, sed, awk
- **General-purpose languages**
  - JavaScript, Python, Ruby, PHP

Characteristics of Dynamic PLs

- Batch and interactive use
- Economy of expression
- Lack of declaration and simple scoping rules
- Flexible dynamic typing
- Easy access to other programs
- Sophisticated pattern matching and string manipulation
- High-level data types

Batch and Interactive Use

- **Batch use: Pass entire file/program to the compiler, interpreter, or runtime engine**
- **Interactive use: Pass one line or command after another**
  - Interactive shell that evaluates statements and expressions as they come
  - REPL: Read-eval-print loop

Demo

- **Python REPL**

Economy of Expression

- **Little "boilerplate" code**
- **Example: Hello world**
  - Java:

```
class Hello {
  public static void main(String[] args) {
    System.out.println("Hello, world!");
  }
}
```

  - Perl, Python, Ruby:

```
print "Hello, world!\n"
```

Declarations and Scoping

- Do variables have to be declared?
- What's the scope of a variable?
- Can subroutines be nested, and if yes, what does it mean?
- Any namespaces for information hiding?

Declarations and Scoping

- Do variables have to be declared?
- What's the scope of a variable?
- Can subroutines be nested, and if yes, what does it mean?
- Any namespaces for information hiding?

Different PLs: Different answers

Variable Declarations

Three popular options:

- Variables are not declared
  - E.g., Python, Ruby
- Declarations are optional
  - E.g., Perl, JavaScript,
- Must declare variables
  - E.g., Scheme (and Perl and JavaScript in strict mode)

Demo

strictMode.js

Scope of Variables: Ruby

- **If a variable is never declared, what's its scope?**
- **Ruby's answer: Naming conventions**
  - `foo`: local variable
  - `$foo`: global variable
  - `@foo`: instance variable of current object
  - `@@foo`: instance variable of current object's class

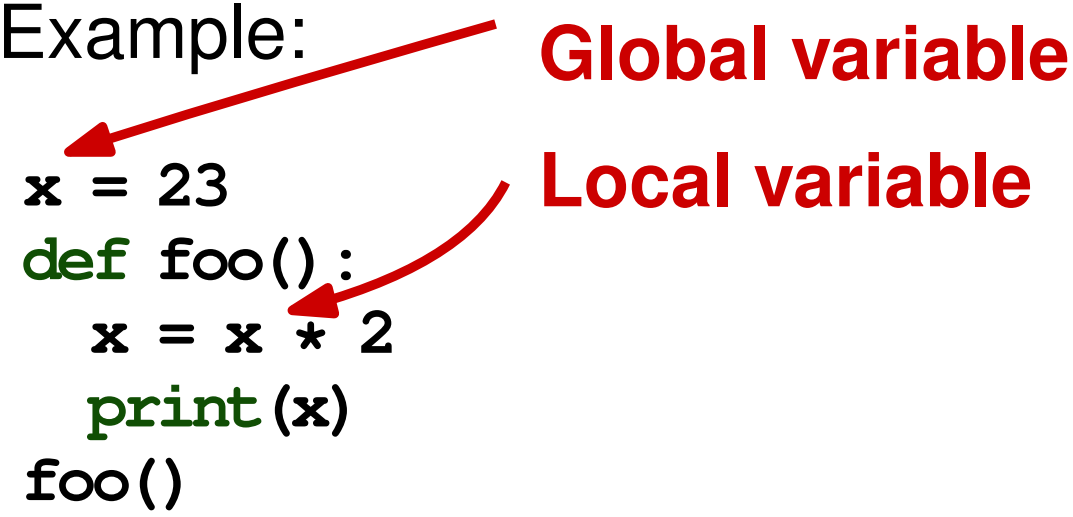
Scope of Variables Python

- **Python: Scope depends on location of variable definition**
  - Defined outside of function: Global
  - Defined inside a function: Local
  - Example:

```
x = 23
def foo():
  x = x * 2
  print(x)
foo()
```

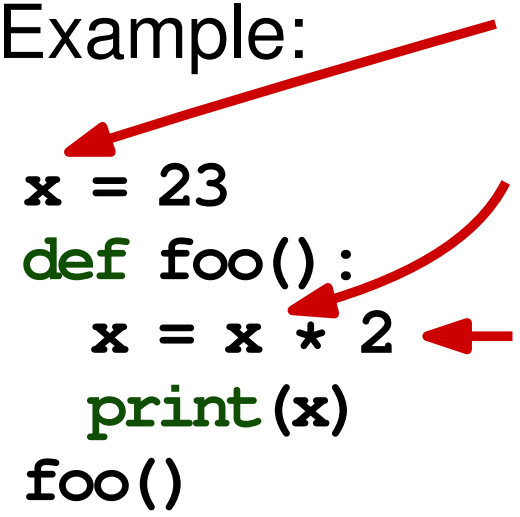
Scope of Variables Python

- **Python: Scope depends on location of variable definition**
  - Defined outside of function: Global
  - Defined inside a function: Local
  - Example:



Scope of Variables Python

- **Python: Scope depends on location of variable definition**
  - Defined outside of function: Global
  - Defined inside a function: Local
  - Example:



Global variable

Local variable

UnboundLocalError: local variable 'x' referenced before assignment

Scope of Variables: Python (2)

- **By default, global variables are readable in functions**
- **Make global variable writable: `global` keyword**
  - Example:

```
x = 23
def foo():
    global x
    x = x * 2
    print(x)
foo()
```

Scope of Variables: Python (2)

- By default, global variables are readable in functions
- Make global variable writable: `global` keyword
  - Example:

```
x = 23
def foo():
    global x
    x = x * 2
    print(x)
foo()
```

No local variable `x`; global variable `x` is writable

Prints 46

Scope of Variables: Python (3)

- **Nested functions: Scope of variable is closest enclosing scope that contains a write to the variable**
  - Example:

```
def foo():
    def bar():
        y = x + 2
        print(y)
    x = 3
    bar()
    print(x)
foo()
```

Scope of Variables: Python (3)

- **Nested functions: Scope of variable is closest enclosing scope that contains a write to the variable**
  - Example:

```
def foo():
    def bar():
        y = x + 2
        print(y)
    x = 3
    bar()
    print(x)
foo()
```

`y` is local to `bar`

`x` is local to `foo`, and hence, also visible in `bar`

Scope of Variables: Python (3)

- **Nested functions: Scope of variable is closest enclosing scope that contains a write to the variable**
  - Example:

```
def foo():
    def bar():
        y = x + 2
        print(y)
    x = 3
    bar()
    print(x)
foo()
```

y is local to bar

x is local to foo, and hence, also visible in bar

Prints 5 and 3

Quiz: Scopes in Python

What does this Python code print?

```
i = 1; j = 3

def outer():
  def middle(k):
    def inner():
      global i
      i = 4
    inner()
    return i, j, k
  i = 2
  return middle(j)

print(outer())
print(i, j)
```

https://ilias3.uni-stuttgart.de/vote/0ZT9

Quiz: Scopes in Python

What does this Python code print?

```
i = 1; j = 3

def outer():
  def middle(k):
    def inner():
      global i
      i = 4
    inner()
    return i, j, k
  i = 2
  return middle(j)

print(outer())
print(i, j)
```

Output: (2, 3, 3) and 4 3

https://ilias3.uni-stuttgart.de/vote/0ZT9

Quiz: Scopes in Python

What does this Python code print?

```
i = 1; j = 3
```

Global variables

```
def outer():
  def middle(k):
    def inner():
      global i
      i = 4
    inner()
    return i, j, k
  i = 2
  return middle(j)

print(outer())
print(i, j)
```

Output: (2, 3, 3) and 4 3

https://ilias3.uni-stuttgart.de/vote/0ZT9

Quiz: Scopes in Python

What does this Python code print?

```
i = 1; j = 3

def outer():
  def middle(k):
    def inner():
      global i
      i = 4
    inner()
    return i, j, k
  i = 2
  return middle(j)

print(outer())
print(i, j)
```

Global variables

Global variable made available in `inner`

Output: (2, 3, 3) and 4 3

https://ilias3.uni-stuttgart.de/vote/0ZT9

Quiz: Scopes in Python

What does this Python code print?

```
i = 1; j = 3

def outer():
  def middle(k):
    def inner():
      global i
      i = 4
    inner()
    return i, j, k
  i = 2
  return middle(j)

print(outer())
print(i, j)
```

Global variables

Global variable made available in `inner`

This `i` is local to `outer`, and hence, also visible in `middle`

Output: (2, 3, 3) and 4 3

https://ilias3.uni-stuttgart.de/vote/0ZT9

Flexible Dynamic Typing

- **Dynamically typed, i.e., no type annotations**
- **Type coercions: Same variable interpreted differently depending on context**
  - Example (Perl):

```
$a = "4";
print $a . 3 . "\n";   # prints 43
print $a + 3 . "\n";   # prints 7
```

Flexible Dynamic Typing

- **Dynamically typed, i.e., no type annotations**
- **Type coercions: Same variable interpreted differently depending on context**
  - Example (Perl):

```
$a = "4";
print $a . 3 . "\n";   # prints 43
print $a + 3 . "\n";   # prints 7
```

$a interpreted as a string

$a interpreted as an integer

Access to Other Programs

- **Syntax and lightweight APIs to interact with underlying operating system, e.g.,**
  - Launch other commands
  - File manipulation
  - Process management

Access to Other Programs

- **Syntax and lightweight APIs to interact with underlying operating system, e.g.,**
  - Launch other commands
  - File manipulation
  - Process management

```
# Bash:
for fig in *.eps
do
  ps2pdf $fig
done
```

```
# Python:
proc = ...
os.kill(proc, 9)
```

Pattern Matching & String Manip.

- **Built-in support for sophisticated**
  - ... pattern matching
  - ... search within text
  - ... string manipulation
- **Useful, e.g., for**
  - ... text processing and report generation
  - ... manipulating textual input and output of external programs

Extended Regular Expressions

- **Formal language theory: Regular expressions (regexs) describe regular languages**
- **Practical PLs: Various non-regular extensions of regular expressions**
  - E.g., backreference
    - `/x(a|b)x\1/` means x, following by a or b, following by x, and then followed by the same character as before (a or b)

POSIX Regexs

- **Concatenation, alternation (`|`), Kleene closure (`*`)**
- **Parentheses for grouping**
- **Repetitions**
  - `?` for zero or one
  - `+` for one or more
  - `{n}` for n
  - `{n,}` for n or more
  - `{n,m}` for n to m

POSIX Regexs

- **Concatenation, alternation (`|`), Kleene closure (`*`)**
- **Parentheses for grouping**
- **Repetitions**
  - `?` for zero or one
  - `+` for one or more
  - `{n}` for n
  - `{n,}` for n or more
  - `{n,m}` for n to m

Examples:

- `/ab(cd|ef)g*/` matches abcdg, abefg, abcdgggg, etc.
- `/a(bc){2,}/` matches abcbc, abbcbcbc, etc.

POSIX Regexs (2)

- `^` and `$`: beginning and end of string
- `[ ... ]` describes character classes, i.e., set of possible characters
  - Negation: Prepend with `^`
- Dot (`.`): Any character except newline

POSIX Regexs (2)

- **`^` and `$`: beginning and end of string**
- **`[ ... ]` describes character classes, i.e., set of possible characters**
  - Negation: Prepend with `^`
- **Dot (`.`): Any character except newline**

Examples:

- `/[^aq]/` means neither a nor q
- `/^b.d/` matches bxda and bqd, but neither abxd nor bd

Implementation of Regexs

- **Before matching a regex against a string: Compile it into**
  - ... deterministic finite automaton
  - ... non-deterministic finite automaton with backtracking
- **Reusing compiled regex: Efficient when matching against it multiple times**

Implementation of Regexs

- **Before matching a regex against a string: Compile it into**
  - ... deterministic finite automaton
  - ... non-deterministic finite automaton with backtracking
- **Reusing compiled regex: Efficient when matching against it multiple times**

For regexs that describe a regular language

For extended regular expressions, e.g., using backreferences

Example: ReDoS Vulnerabilities

- **Understanding regexs: Important, e.g., to prevent vulnerabilities**
- **ReDoS: Regular expression denial of service**
  - Matching against potentially malicious input takes super-linear time
  - E.g., regex `/(a+)+b/` in JavaScript
    - 30x 'a': 15 seconds; 35x 'a': 8 minutes

More details: *Freezing the Web: A Study of ReDoS Vulnerabilities in JavaScript-based Web Servers*. Staicu & Pradel. USENIX Security '18

Demo

ReDoS.js

Quiz: POSIX Regexs

- **Which of the following is correct?**
  - `/a?b{1,2}c$/` matches bbcc
  - `/(ab?)+c/` matches ababcc
  - `/[xyz]+.[^xyz]+/` matches yxx
  - `/[xyz]+.[^xyz]+/` matches xxxa
  - `/(x.*|yz){2}/` matches xxyzyz

https://ilias3.uni-stuttgart.de/vote/0ZT9

Quiz: POSIX Regexs

- **Which of the following is correct?**
  - `/a?b{1,2}c$/` matches bbcc ✘
  - `/(ab?)+c/` matches ababcc ✔
  - `/[xyz]+.[^xyz]+/` matches yxx ✘
  - `/[xyz]+.[^xyz]+/` matches xxxa ✔
  - `/(x.*|yz){2}/` matches xxyzyz ✔

https://ilias3.uni-stuttgart.de/vote/0ZT9

High-level Data Types

- **Many built-in data types**
  - Sets, lists, tuples, dictionaries, etc.
- **Not in a library, but part of syntax and semantics of the PL**
  - Example: JavaScript

```
var john = {
  "age": 23,
  "address": "1024 Oxford St"
}
var people =[]
people.push(john)
```

High-level Data Types

- **Many built-in data types**
  - Sets, lists, tuples, dictionaries, etc.
- **Not in a library, but part of syntax and semantics of the PL**
  - Example: JavaScript

```
var john = {
  "age": 23,
  "address": "1024 Oxford St"
}
var people =[]
people.push(john)
```

Objects are dictionaries

Array or list

Prototypes

- **Prototype-based object-orientation**
  - Alternative to class-based object-orientation
  - E.g., in Self and JavaScript
- **Constructor functions create objects**
- **Prototype: Object that stores properties available in all objects created with a constructor**
  - E.g., methods of the created objects

Example: JavaScript

```
function Cat(name) {
    this.name = name;
}

Cat.prototype = {
    sayHello: function() {
        console.log("Miau – I'm " + this.name);
    }
};

var myCat = new Cat("Catty");
myCat.sayHello();
```

Example: JavaScript

Function that serves as a constructor

```
function Cat(name) {
    this.name = name;
}

Cat.prototype = {
    sayHello: function() {
        console.log("Miau – I'm " + this.name);
    }
};

var myCat = new Cat("Catty");
myCat.sayHello();
```

Example: JavaScript

Function that serves as a constructor

Prototype associated with the constructor

```
function Cat(name) {
    this.name = name;
}

Cat.prototype = {
    sayHello: function() {
        console.log("Miau – I'm " + this.name);
    }
};

var myCat = new Cat("Catty");
myCat.sayHello();
```

Example: JavaScript

```
function Cat(name) {
    this.name = name;
}

Cat.prototype = {
    sayHello: function() {
        console.log("Miau – I'm " + this.name);
    }
};

var myCat = new Cat("Catty");
myCat.sayHello();
```

Function that serves as a constructor

Prototype associated with the constructor

Create object and call a method

Prototype Chains

- **Chaining multiple prototypes to implement inheritance-like code reuse**
- **Property access**
  - At first, lookup in object itself
  - If not found: Go up prototype chain of object until found

Example: JavaScript

```
var animalPrototype = {
    eat: function() {
        console.log("Eating...");
    }
};

function Cat(name) {
    this.name = name;
}
Cat.prototype = Object.create(animalPrototype);
Cat.prototype.sayHello = function() {
    console.log("Miau – I'm " + this.name);
};

var myCat = new Cat("Catty");
myCat.sayHello();
myCat.eat();
```

Example: JavaScript

```
var animalPrototype = {
    eat: function() {
        console.log("Eating...");
    }
};

function Cat(name) {
    this.name = name;
}
Cat.prototype = Object.create(animalPrototype);
Cat.prototype.sayHello = function() {
    console.log("Miau – I'm " + this.name);
};

var myCat = new Cat("Catty");
myCat.sayHello();
myCat.eat();
```

Prototype for animals

Example: JavaScript

```
var animalPrototype = {
    eat: function() {
        console.log("Eating...");
    }
};

function Cat(name) {
    this.name = name;
}
Cat.prototype = Object.create(animalPrototype);
Cat.prototype.sayHello = function() {
    console.log("Miau – I'm " + this.name);
};

var myCat = new Cat("Catty");
myCat.sayHello();
myCat.eat();
```

Prototype for animals

Creates object with `animalPrototype` as its prototype

Example: JavaScript

```
var animalPrototype = {
    eat: function() {
        console.log("Eating...");
    }
};

function Cat(name) {
    this.name = name;
}
Cat.prototype = Object.create(animalPrototype);
Cat.prototype.sayHello = function() {
    console.log("Miau – I'm " + this.name);
};

var myCat = new Cat("Catty");
myCat.sayHello();
myCat.eat();
```

Prototype for animals

Creates object with `animalPrototype` as its prototype

Prints "Eating ..."

Characteristics of Dynamic PLs

- Batch and interactive use
- Economy of expression
- Lack of declaration and simple scoping rules
- Flexible dynamic typing
- Easy access to other programs
- Sophisticated pattern matching and string manipulation
- High-level data types ✔