The Evolution of Programming Languages

A Personal Perspective

Lutz Hamel

What's Happening to PLs Today?

- There is a qualitative shift if you look at programming languages such as Python or Ruby and compare them to languages such as C and Java:
  - Type systems have become much more flexible – dynamic typing
  - Data structures have become much more abstract; similar to functional programming languages
  - Full support for higher-order programming
  - Clean, succinct syntax

PL Comparison

- In order to compare PLs we use two benchmark programs
- Simple things should be easy
  - seems kind of obvious but in Java for example that is not true
- The Polymorphic List
  - one thing that programmers do a lot is keeping track of things
    - arrays
    - vectors
    - lists
    - tuples

‘Hello’

- Here is a very simple program that allows us to assess how easy it is to implement something simple in a programming language
- The pseudo code is,

```
Begin
    Ask user for name.
    Print “Hello ” + name
End
```

The Polymorphic List

- Polymorphic means “multiple shapes” – in terms of lists that means that we can have a list with items that are not necessarily related (via types)
- This not something only OO programmers do but John McCarthy who designed Lisp recognized early on that keeping lists of things is vital to programming in general
  - hence LISt Processor

https://en.wikipedia.org/wiki/John_McCarthy_(computer_scientist)

The Polymorphic List

- Pseudo code:

```
Begin
    Let orange [be of type Orange]
    Let apple [be of type Apple]
    Let pear [be of type Pear]
    Let list <- list-of(orange,apple,pear)
    Print list
End
```

PLs in 1950s/1960s

- 1951 – Regional Assembly Language
- 1952 – Autocode
- 1954 – IPL (forerunner to LISP)
- 1955 – FLOW-MATIC (led to COBOL)
- 1957 – FORTRAN (First compiler)
- 1957 – COMTRAN (precursor to COBOL)
- 1958 – LISP
- 1958 – ALGOL 58
- 1959 – FACT (forerunner to COBOL)
- 1959 – COBOL
- 1959 – RPG
- 1962 – APL
- 1962 – Simula
- 1962 – SNOBOL
- 1963 – CPL (forerunner to C)
- 1964 – Speakeasy (computational environment)
- 1964 – BASIC
- 1964 – PL/I
- 1966 – JOSS
- 1967 – BCPL (forerunner to C)

- Lisp, FORTRAN, and Basic only survivors
- Fortran and Basic not really general purpose languages
  - only compound data structure is the array
  - no recursion

Source: https://en.wikipedia.org/wiki/History_of_programming_languages

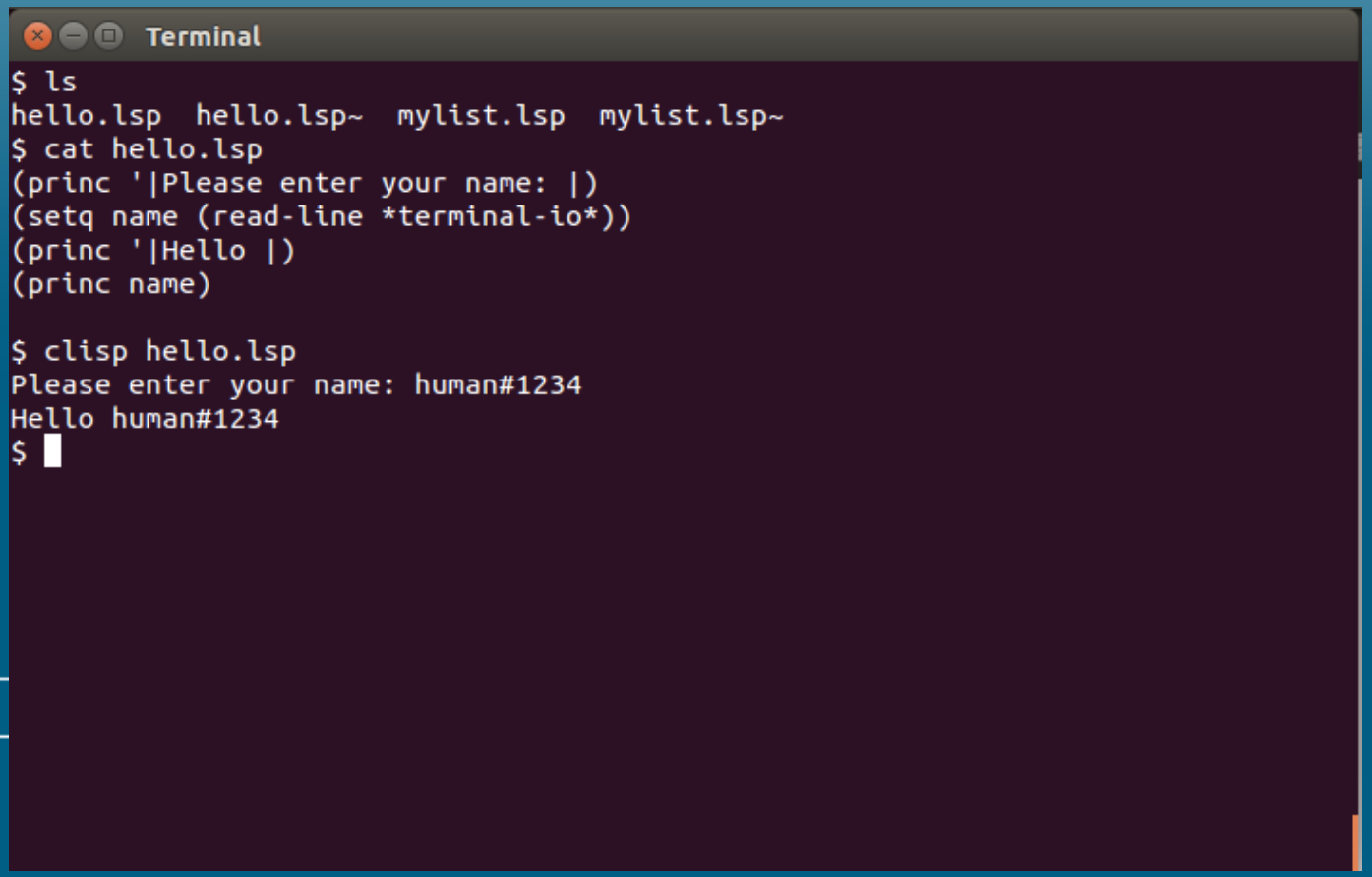
Lisp

- Hugely influential
  - recursion
  - garbage collection
  - higher-order programming
  - “programs are data – data are programs”
  - fundamental data structure: the list
  - dynamically typed (barely…)

Lisp - Easy Things are Easy

```
(princ '|Please enter your name: |)
(setq name (read-line *terminal-io*))
(princ '|Hello |)
(princ name)
```

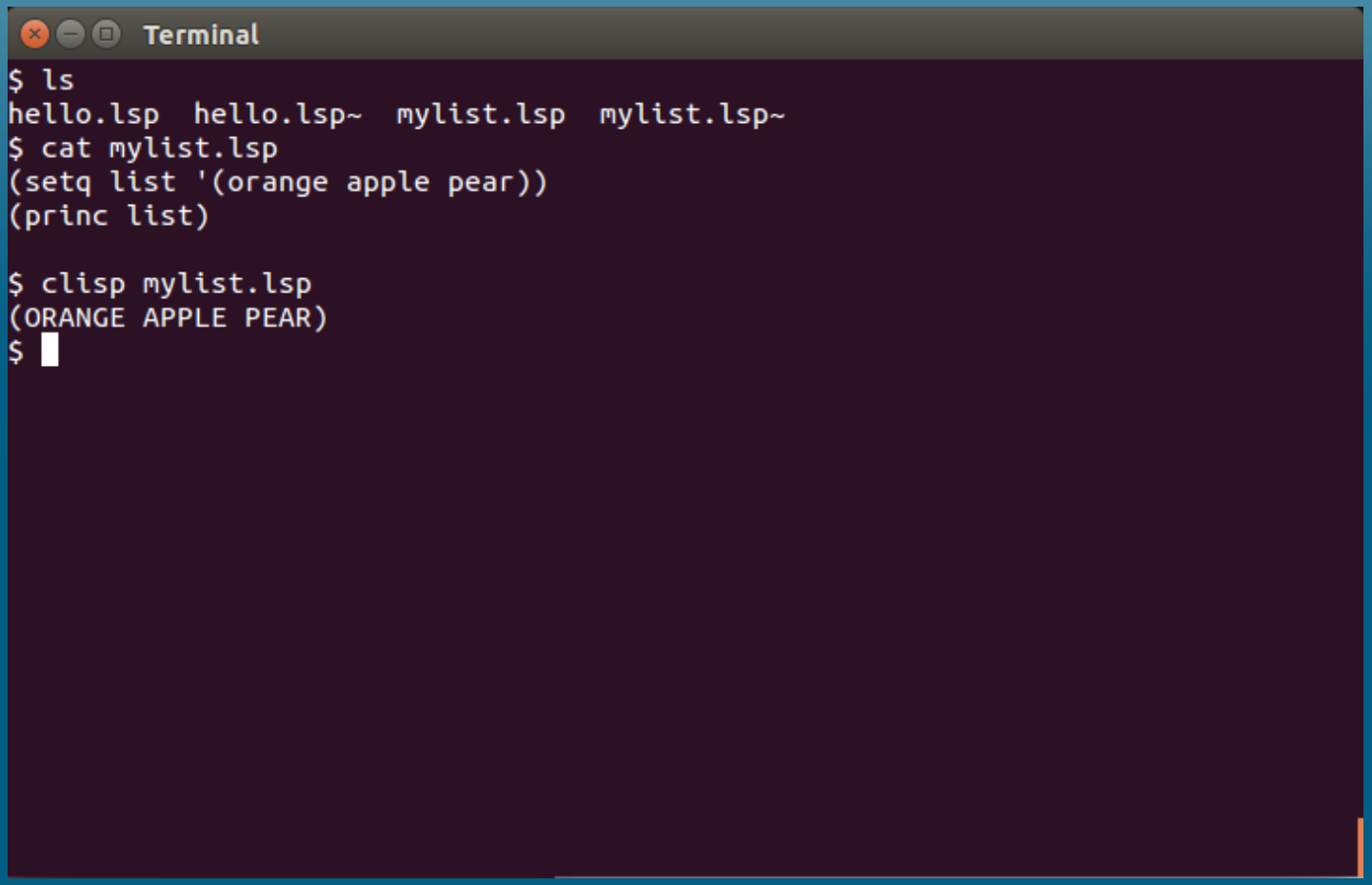
Lisp - Easy Things are Easy



Lisp - Polymorphic List

```
(setq list '(orange apple pear))
(princ list)
```

Lisp - Polymorphic List



PLs in the 1960s/1970s

- 1968 – Logo
- 1969 – B (forerunner to C)
- 1970 – Pascal
- 1970 – Forth
- 1972 – C
- 1972 – Smalltalk
- 1972 – Prolog
- 1973 – ML
- 1975 – Scheme
- 1978 – SQL (a query language, later extended)

- By far the most popular language from that era is C
- Even today, 40+ years later, it is one of the most used programming languages

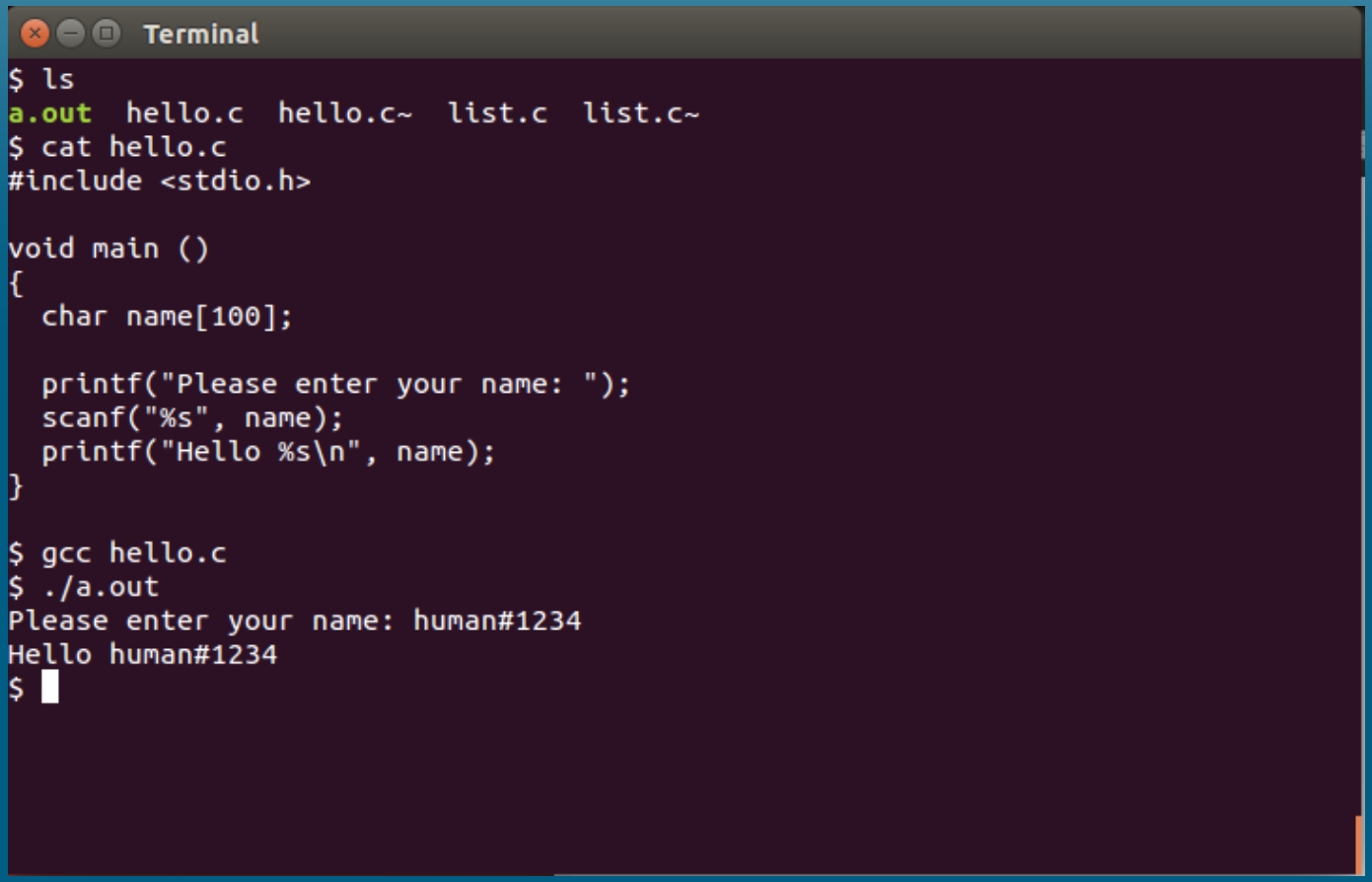
C

- A hugely successful language designed for developing real time systems/OSs
- hall marks
  - very tight syntax
  - pointers and pointer arithmetic including function pointers
  - explicit memory management

C - Simple Things are Easy

```
#include <stdio.h>

void main ()
{
  char name[100];

  printf("Please enter your name: ");
  scanf("%s", name);
  printf("Hello %s\n", name);
}
```

C - Simple Things are Easy



C – Polymorphic List

- VERY difficult!
- Lists/arrays can only be of the same data type, the only way to get different data types represented in a list/array is to do something creative with union/struct.

C - Polymorphic List

- A simple polymorphic list that allows you to store ints and floats in the same structure
- It feels like a kludge - and it is
- C does not support polymorphic lists

```
void main ()
{
  struct
  {
    enum {INT, FLOAT} tag;
    union
    {
      int i;
      float f;
    } u;
  } a[2];

  a[0].tag = INT;
  a[0].u.i = 1;

  a[1].tag = FLOAT;
  a[1].u.f = 1.0;
}
```

Static Type Systems

- Pros: great at catching programming errors early
- Cons: over-complicates code

Question: are static type systems great at catching bugs that get introduced because of the over-complication of code?

PLs in the 1980s/1990s

- 1980 – C++ (as C with classes, renamed in 1983)
- 1983 – Ada
- 1984 – Common Lisp
- 1984 – MATLAB
- 1985 – Eiffel
- 1986 – Objective-C
- 1986 – Erlang
- 1987 – Perl
- 1988 – Tcl
- 1988 – Mathematica
- 1989 – FL (Backus)

- 1990 – Haskell
- 1991 – Python
- 1991 – Visual Basic
- 1993 – Ruby
- 1993 – Lua
- 1994 – CLOS (part of ANSI Common Lisp)
- 1995 – Ada 95
- 1995 – Java
- 1995 – Delphi (Object Pascal)
- 1995 – JavaScript
- 1995 – PHP
- 1996 – WebDNA
- 1997 – Rebol
- 1999 – D

Java

- OO programming language modeled after C++
- Design objective – be as OO as possible, removing some of the design choices C++ made:
  - no global objects/functions
  - no multiple inheritance
  - a class structure that is rooted in Object
  - OO wrappers around I/O
  - “Everything is an object”
    - except for primitives like ints and floats

Java - Simple Things are Easy

```
import java.io.*;

public class Hello
{
   public static void main(String[] args) throws IOException
   {
       InputStreamReader sr = new InputStreamReader(System.in);
       BufferedReader in = new BufferedReader(sr);

       System.out.print("Please enter your name: ");
       String name = in.readLine();
       System.out.println("Hello " + name);
   }
}
```

Dogmatic OO?!?

Java - Simple Things are Easy

Terminal

```
$ cat Hello.java
import java.io.*;

public class Hello
{

   public static void main(String[] args) throws IOException
   {
        InputStreamReader sr = new InputStreamReader(System.in);
        BufferedReader in = new BufferedReader(sr);

        System.out.print("Please enter your name: ");
        String name = in.readLine();
        System.out.println("Hello " + name);
   }
}
$ javac Hello.java
$ java Hello
Please enter your name: human#1234
Hello human#1234
$
```

Java - Polymorphic List

```
abstract class Fruit
{
    abstract void print();
}

class Apple extends Fruit
{
    void print() { System.out.println("Apple"); }
}

class Orange extends Fruit
{
    void print() { System.out.println("Orange"); }
}

class Pear extends Fruit
{
    void print() { System.out.println(”Pear"); }
}
```

Java - Polymorphic List

```
class Basket
{
    public static void main(String[] args)
    {
        List<Fruit> list = new ArrayList<Fruit>();
        list.add(new Apple());
        list.add(new Orange());

        for(Fruit fruit : list){
            fruit.print();
        }
    }
}
```

Java - Polymorphic List

Terminal

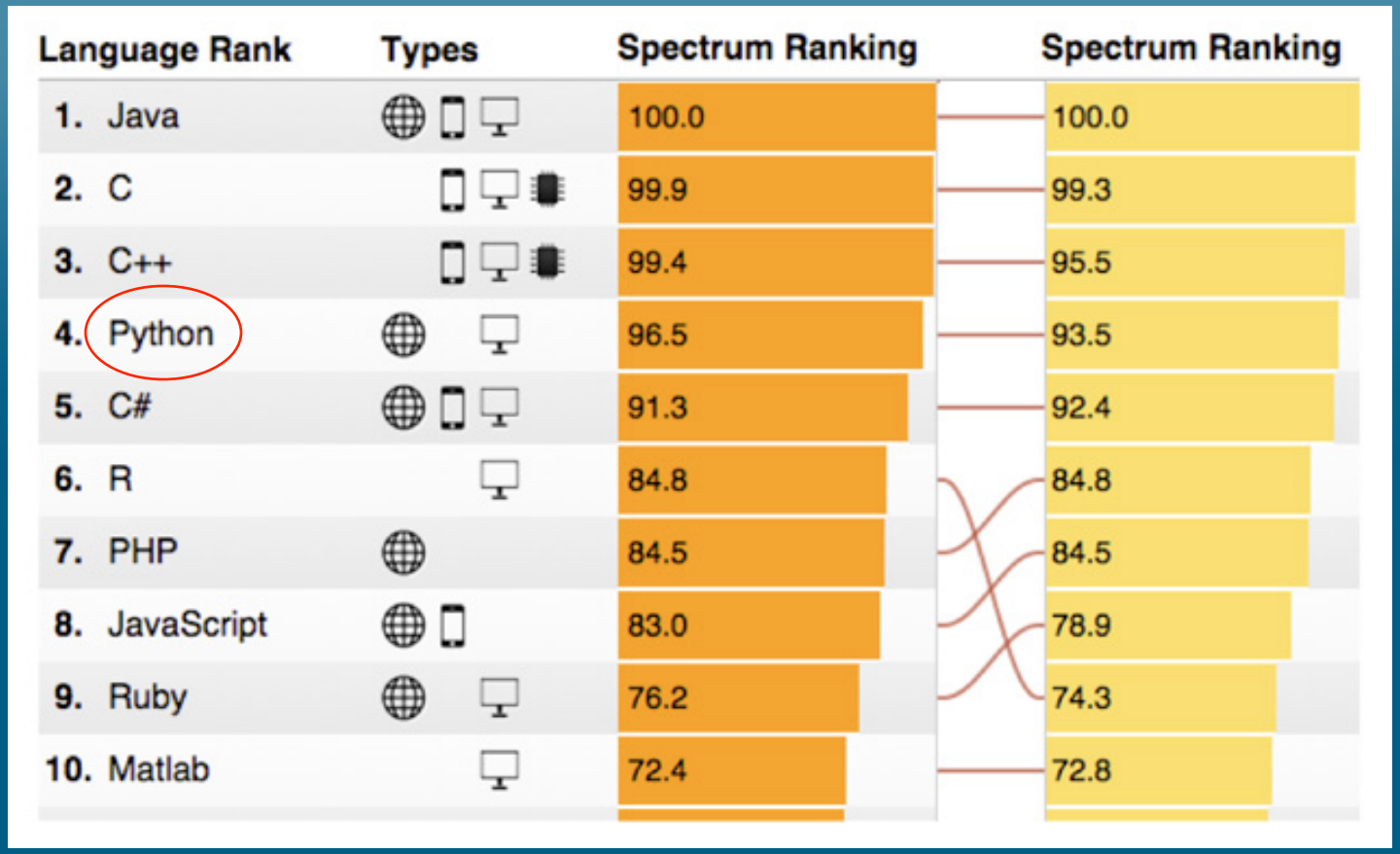
```
{
    void print() { System.out.println("Pear"); }
}

class Basket
{
    public static void main(String[] args)
    {
        List<Fruit> list = new ArrayList<Fruit>();
        list.add(new Apple());
        list.add(new Orange());
        list.add(new Pear());

        for(Fruit fruit : list){
            fruit.print();
        }
    }
}
$ javac Basket.java
$ java Basket
Apple
Orange
Pear
$
```

So much code that it does not even fit into a single terminal window!

Java – Polymorphic List

- Needs class hierarchy
- Needs generics as container
- Lots of scaffolding, lots of code – lots of possibility for error

PLs in 20XX



Source: http://spectrum.ieee.org/computing/software/the-2015-top-ten-programming-languages

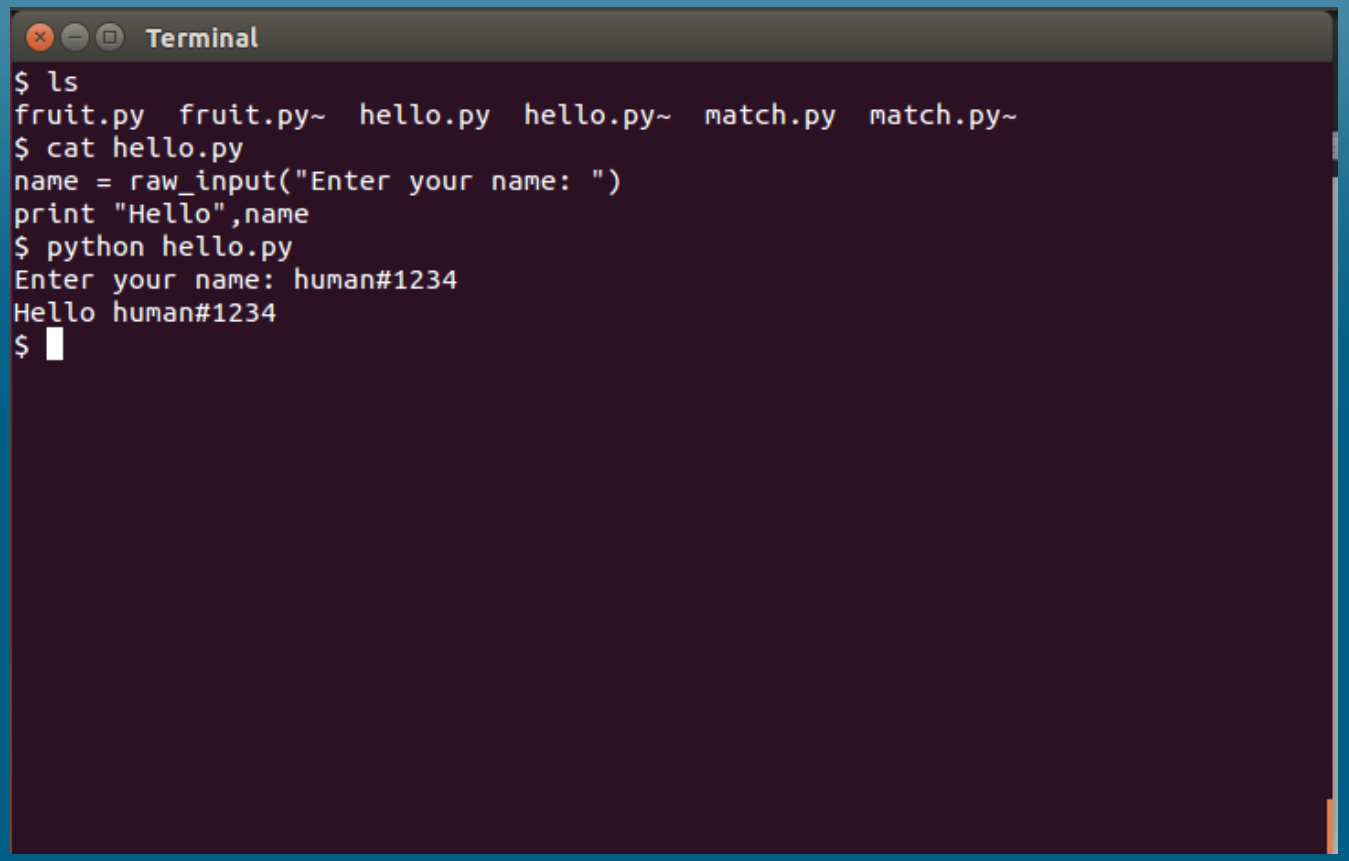
Python

- Python supports multiple programming paradigms, including object-oriented, imperative and functional programming or procedural styles.

Python - Simple Things are Easy

```
name = raw_input("Enter your name: ")
print "Hello",name
```

Python - Simple Things are Easy

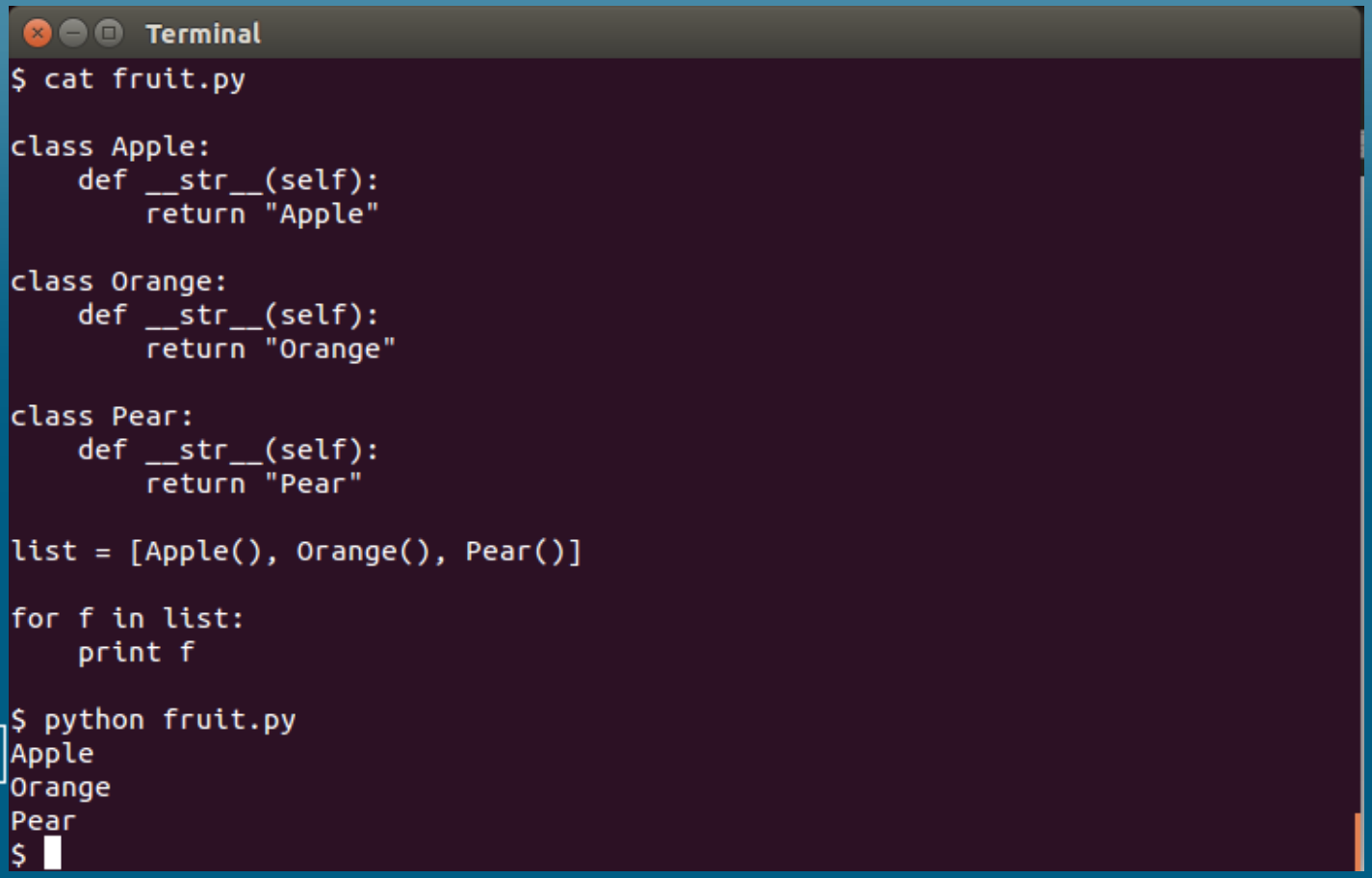


Python - Polymorphic List

- Dynamic typing
- “Duck typing” (no base class necessary)
- Clean syntax

```
class Apple:
    def __str__(self):
        return "Apple"

class Orange:
    def __str__(self):
        return "Orange"

class Pear:
    def __str__(self):
        return ”Pear"

list = [Apple(), Orange(), Pear()]

for f in list:
    print f
```

Python - Polymorphic List



“Duck Typing”

- The name of the concept refers to the duck test, attributed to James Whitcomb Riley, which may be paraphrased as follows:
  - *An object that walks like a duck, swims like a duck, and quacks like a duck is a duck.*
- In duck typing, a programmer is only concerned with ensuring that objects behave as demanded of them in a given context, rather than ensuring that they are of a specific class.

Source: https://en.wikipedia.org/wiki/Duck_typing

Lightweight OO

- “Duck Typing” is a corner stone to make OO more usable
- In large projects class hierarchies evolve
  - VERY difficult to accomplish in OO systems such as C++ and Java
  - much easier to handle in OO systems such as Python and Ruby - class hierarchies consist of multiple smaller ones not necessarily related via a single base class
  - but polymorphic programming still available because of “duck typing” and dynamic typing

Full Circle?

Lisp – 1950s

```
(princ '|Please enter your name: |)
(setq name (read-line *terminal-io*))
(princ '|Hello |)
(princ name)
```

Python - 2016

```
name = raw_input("Enter your name: ")
print "Hello",name
```

Full Circle?

Lisp – 1950s

```
(setq list '(orange apple pear))
(princ list)
```

Python - 2016

```
class Apple:
    def __str__(self):
        return "Apple"

class Orange:
    def __str__(self):
        return "Orange"

class Pear:
    def __str__(self):
        return ”Pear"

list = [Apple(), Orange(), Pear()]

for f in list:
    print f
```

Conclusions

- New languages like Python, Ruby, R etc
  - dynamic typing
  - lightweight OO (“duck typing”)
  - clean, concise syntax
  - higher order
  - sacrifice strong typing for much more abstract program structures (i.e. lists)

Question: Less code, more abstract syntax and data structures = better code?

Thank You!

- Presentation available on my homepage
  - http://homepage.cs.uri.edu/faculty/hamel/pubs/