# CS 345 Project: *polite*
# (Python Objective Lite)

*BY*
HANLIN CHIANG AND ALEX KUO

# Features

- No declaration needed to use variables!
- Indentations as delimiters
- First class functions
- Lambda expression
- Recursive data structure
  - **Unrestricted level of members**

    ```
    a.level1_member.level2_member.level3_member. …
    ```
- Variables are dynamically bounded, with exceptions of a few built-in functions and constants.

# Data types

- Basic types:
  - *object*, *integer*, *float*, *bool*, *list*, *tuple*, and *string*
  - Constants: `None`, `True`, `False`

- Functions as objects

```
def somefunction(v1, v2):
      …
a= somefunction
a(1,2)
```

```
def new_method(self, input):
      …
obj.process = new_method
obj.process(data)
```

# Code Structure

```
def offset_value(offset):
   def new_function(input):
         return offset + input
   return new_function

def map(l, f):
   c = 0
   newl = list()
   while c < len(l):
         newl.append(f(l.get(c)))
         c = c + 1
   return newl

ls = [ 1, 2, 3, 4,]
funobj = offset_value(1)
print ls
print map(ls, funobj)
```

- Code structure based on python:
  - Indentation defines block
  - Allows operator overloading
  - Reflection
  - Features mostly subset of python ( after all, there were only 4 weeks)
- Key additions:
  - Everything is object!
  - Dynamic method parameter passing
  - Calling non-existing member will not result in error *

✓ **Some thing that you can do in polite**

```
a = 1.__add__(2)
def new_method(self, that):
   …
a.my_method= new_method
a.my_method(1)
a=object().this.and_that
```

** Due to a different view on how object and member interact with each other, details later.*

# Typing and Data Binding

- Dynamic data binding, variables are not restricted by type even after assignment.

```
a= [] ; a= 'string value'

# member can be assigned
#   at any time
a.b = 13

# output: string value  13
print a, a.b
```

- Function parameters are checked at runtime. This only be done at runtime, since variables may be assigned a function object that'd accept the call.

```
def function(u, v):
   …
# not happening.
function(3, 4, 5)
```

- Interpreter determines at runtime for functions that demands correct variable type

```
a = 1
b = “”
b.__add__ = a.__add__

# statement will fail runtime type
#   checking, cannot add string and
#   integer together without type
#   conversion.
c = b + 3
```

- Only static checking performed is to ensure that a variable being used must have already been assigned somewhere

```
c = 3 + b  # b is undefined yet.
b = 10
```

# Scope: Static Scope/ Global variables

```python
x = 1000
def  testglobal():
   # uses ‘x’ defined above, but cannot
   #   assign value to ‘x’
   print 'outer1: ', x,
   def  inner():
      # tells function to refer to ‘x’
      #   from top level instead
      global x
      print ' inner1: ', x, ' adding 1'
      x = x + 1  # now allows assignment
   return  inner

def testglobal2():
   global x
   print 'outer2: ', x,
   def  inner():
      # one way street, can refer to
      #   variables in testglobal2, but may
      #   not modify them.  global goes
      #   directly to top level
      print ' inner2: ', x
   x = x + 1  # add one to check inner.
   return  inner
```

```python
a = testglobal(); a()
b = testglobal2(); b()
x = 1002
testglobal()()
testglobal2()()
a()
b()
```

**Result:**

global *x*

```
outer1:   1000  inner1:  1000   adding 1
outer2:  1001  inner2:  1001
outer1:   1000  inner1:  1002   adding 1
outer2:  1003  inner2:  1003
 inner1:  1004   adding 1
 inner2:   1001
```

# Parameters Passing

- Similar to how Java handles value passing, parameters are passed by value, which are references to instances.

```
def function(a):
  a.s = 1  # adding member
  a = 3    # reassign value

a = 2
print a, a.s # Output: 2 None

# this adds member to ‘a’
function(a)
print a, a.s # Output: 2 1
```

- Method call resolves parameter input dynamically. It will use instance as first argument if function is still missing one argument.

```
def function(v):
  …
def function2():
  …
a.f1 = function
a.f2 = function2
a.f1()    # use ‘a’ as input

a.f1(1)   # may also pass in
          # directly

a.f2()    # normal call
```

# First Class Function / Lambda Expression

```
def add_offset(offset):
   # state at time of function
   #   object creation is saved
   def new_adder(input):
      # in this case, ‘offset’ is now
      #    a saved state for new_adder
      return offset + input

   return new_adder

def map(l, f):
   # Passing function as parameter
   c = 0
   newl = list()
   while c < len(l):
      # perform value mapping with
      #   function given
      newl.append(f(l.get(c)))
      c=c+1
   return newl
```

```
ls = [ 1, 2, 3, 4, 5]
funobj1 = add_offset(1)
funobj2 = add_offset(10)

print 'input=',ls
print map(ls, funobj1)
print map(ls, funobj2)

# lastly, a lambda expression.
print map(ls, @ x: x * 2 > 6)
```

**Result:**

```
input= [ 1, 2, 3, 4, 5 ]
[ 2, 3, 4, 5, 6 ]
[ 11, 12, 13, 14, 15 ]
[ False, False, False, True, True ]
```

# Abstract Syntax Tree

Based on the concept of message sending and selector.

3 +4 actually performs the following:

```
send(3,'__add__')(3,4)
```

This is the cause why member access `a.b.c` would not cause error. `a.b` evaulates to:

```
send(a,'b')
```

and would return **None**.

More interesting application (not quite implemented):

```
a_database.(a_query)
```

would send a query to database. Message could be anything!

```
+Statements:
 | PRINT:  (\n)
 | PRINT:  (\n)
 |     LITERAL: string(>>> Syntax tree)
 | ASSIGN: a
 |     LITERAL: integer(10)
 | ASSIGN: b
 |     LITERAL: integer(12)
 | ASSIGN: c
 |     LITERAL: integer(13)
 | ASSIGN: d
 |     CALL:
 |      +functionObj:
 |       | CALL:
 |       |  +functionObj:
 |       |   | IDENTIFIER: send
 |       |  +paramList:
 |       |   | CALL:
 |       |   |  +functionObj:
 |       |   |   | CALL:
 |       |   |   |  +functionObj:
 |       |   |   |   | IDENTIFIER: send
 |       |   |   |  +paramList:
 |       |   |   |   | IDENTIFIER: a
 |       |   |   |   | LITERAL: string(__add__)
 |       |   |  +paramList:
 |       |   |   | IDENTIFIER: b
 |       |   | LITERAL: string(__add__)
 |      +paramList:
 |       | IDENTIFIER: c
```

```
# code that the tree
# represents!

print
print '>>> Syntax tree'
a = 10
b = 12
c = 13
d = a + b + c
```

# Operator Overloading, Reflection, and Runtime Type Conversion

```
a = object()
# checking type of 'a' and its members
print type(a), dir(a)
```

**Result:**

```
type:object [ __and__, __eq__, __getattr__, __ne__, __nonzero__, __not__, __or__, __setattr__, __str__ ]
```

```
# adding a new member
a.new_member=10
print dir(a)
```

**Result:**

new member is added and reflected in runtime.

```
[ __and__, __eq__, __getattr__, __ne__, __nonzero__, __not__, __or__, __setattr__, __str__, new_member ]
```

```
def myInt(a):
    newInt= int(a)
    def new_add(v1, v2):
        # add is now a string concatenation
        result = str(v1) + ' + ' + str(v2)
        result.__add__ = new_add
        return result
    newInt.__add__=new_add
    return newInt

a = myInt(1); b = 2; c = 3; d = 4; e =5.5
print a + b + c * d + e
# BIT which retrieves default member
a.__add__=super(a, '__add__')
# result is a float due to 'e'
print a + b + c * d + e
```

**Result:**

```
1 + 2 + 12 + 5.5
20.5
```

# Continuation, Simulating Generator with object

```
def fibonacci(upper_bound):
    ar = list()                        # Using list to calcluate fibonacci series
    ar.counter=0
    def next(self):                    # simulating generator call
        if upper_bound == self.counter :
            return False
        if self.counter < 2:
            self.append(1)
        else:
            self.append(self.get(-1) + self.get(-2))
        self.yield = self.get(-1)      # yield for the value.
        self.counter = self.counter + 1
        return True
                                       # While not as elegant as yield statement, Python generator under the
    ar.has_next = next                 #  cover behaves similar to an iteration, using exception to signal
    return ar                          #  for termination.

fibgen = fibonacci(5)
while fibgen.has_next():
    print fibgen.yield,
print
print fibgen
```

Each call to has_next yields another fibonacci number!

```
1 1 2 3 5 8
[ 1, 1, 2, 3, 5, 8 ]
```

# Simple Parser Class!

```
def SimpleParser():  # fundef as class def
  obj = object()
  # how about an inner class?
  def Node(value):
    # basically, method definitions
    def print_value(self): print self,
    def in_order(self): …
    def pre_order(self): …
    def post_order(self): …

    # and saving them as actual members
    value.print_value = print_value
    value.in_order = in_order
    value.pre_order = pre_order
    value.post_order = post_order
    return value

  def process_input(self, input):
       …
  obj.process_input = process_input
  return obj
```

```
input = ‘3 + 4 + 5 * 6 + 7’
parser = SimpleParser()
print ‘input = “’ + input + ‘”’
parser.process_input(input)
parser.head.in_order(); print
parser.head.pre_order(); print
parser.head.post_order(); print
```

- A simple parser that parses arithmetic expression with only numbers, ‘+’, and ‘*’ operators into a binary tree.
- Output

```
input = "3 + 4 + 5 * 6 + 7"
3 + 4 + 5 * 6 + 7
+ + + 3 4 * 5 6 7
3 4 + 5 6 * + 7 +
```

- Perhaps inheritance?

```
def ComplexParser():
   obj = SimpleParser()
   …
```