Lecture #13: Review

Computational Concepts Toolbox
- Data type: values, literals, operations,
- Expressions, Call expression
- Variables
- Assignment Statement, Tuple assignment
- Sequences: tuple, list
- Dictionaries
- Function Definition Statement
- Conditional Statement
- Iteration: list comp, for, while
- Lambda function expr.
- Higher Order Functions
  - Functions as Values
  - Functions with functions as argument
  - Assignment of function values
- Higher order function patterns
  - Map, Filter, Reduce
  - Function factories – create and return functions
  - Recursion
  - Abstract Data Types
  - Mutation
  - Class & Inheritance
  - Exceptions
  - Iterators & Generators

The Goal of CS88 ...
- Idea
- Understood by others
- Expressed in a clear computational form
- Interpreted by a “machine”
- Answers and Insights

SQL Review
- SELECT (ALL or DISTINCT) expressions over columns (map/reduce), optionally AS names
- FROM specification of table or join of tables
- WHERE conditional expression specifying rows in cols of tables
- GROUP BY aggregation expression defining collections of rows in filtered cols of tables
- ORDER BY expression on rows of filter cols defining order of result

How would you write a Python interpreter?
What do you give to the interpreter?

• An Expression
• A sequence of Statements
• optionally followed by an expression

Basic Process

• Parse the input into logical pieces
• Expression
  • Value or variable (leaves) – of a “type”
  • Tree of operators and operand expressions
    • .. * .., .. + .., …
  • Comprehensions
• Sequence of statements
  • assignment
  • def
  • conditional
  • iteration

Values

• Primitive Value
  • int, float, boolean
• Complex Values
  • string, tuple, list, dict,
  • function, class
  • object, method
• Variable
  • Reference to a value

At the bottom it’s a bunch of bits

• How many distinct things represented in \( N \) bits?
  • \( 2^N \) - Think recursively
    • 2 “things” in 1 bit – \( (0, 1) \)
    • Assume 2\(^{N-1}\) things in \( N-1 \) bits
    • \( 0 \cup \{0, \ldots, 2^{N-1} - 1\} \cup \{0, \ldots, 2^{N-1} - 1\} \)
  • “word” is now (typically) 64 bits
    • Can represent \( 2^{64} \) (over 18 quintillion or \( 1.8 \times 10^{19} \)) different values
• Addresses (unsigned ints): \( 0 \ldots 2^N - 1 \)
• Signed Integers: \( -2^{N-1} \ldots 2^{N-1} - 1 \)
• IEEE Float Point: \( -1^S \times 1.f \times 2^{e-1023} \)

Variable

• Starting with current frame
• Look up variable in frame
• If not present, try parent frame, repeatedly
• Until global frame is reached
• If not found there
• Raise an exception

Variable

• Starting with current frame
• Look up variable in frame
• If not present, try parent frame, repeatedly
• Until global frame is reached
• If not found there
• Raise an exception
Data Structures

Operators

• Evaluate the operand expressions (recursively)
• Check the types of the resulting values to determine the operator for symbol
• If no valid combination, raise exceptions
• Apply operator to resulting values to produce result

Call Expressions

• Evaluate the operand expressions (recursively)
• Evaluate "function" expression to get function to apply
• This may involve function return values or "," or ...
• Check that it is of function type
• If not, raise exception
• Apply function to resulting values to produce result

Built-in Data Structure Constructor

• Evaluate each of the index and value expressions
  - Or raise error
• Allocate storage to hold the data structure
• Fill in values at indices/Key
• Return a reference to the object

Comprehension Expression

• Evaluate iterable expression
• For each element in iteration
• Bind var tuple to value tuple
• Evaluate with each of those variable bindings
• Construct resulting object and return reference to it

Lambda Expression

• Construct a function object that evaluates expression in a frame with variables in <vars> bound to argument values and returning the result
• Return reference to the function object

Python 2.7

```
λ = lambda x, y : x*y
λ(a[2])
```

Frames

```
Global frame
```

Objects

```
λ
```

[ ]
Assignment Statement

\(<\text{var list}> = \)

- Evaluate RHS expression to get value
  - Or raise an exception
- Locate LHS variable(s) in frame path
- For each variable
  - If exists, set variable to expression value
  - If not, create variable of name(s) \(<\text{var list}>\) in current frame

\[
\begin{align*}
x &= 3 \\
y &= x + 4 \\
a, b &= 3, a+b
\end{align*}
\]

Set operation

\([\text{LHS} = \text{RHS}] = \)

- Evaluate RHS to get value
  - Or raise an exception
- Evaluation LHS expressions to get object and index/key
  - Or raise exception
- Set \(\text{obj}[\text{key}]\) to expression value

Define Statement

\(\text{def } <\text{fun name}>(<\text{var list}>) : <\text{suite of statements}>\)

- Construct a function object to evaluate \(<\text{suite of statements}>\) in a frame with \(<\text{var list}>\) as local variable bound to argument expressions
- \text{return} statements evaluate expression in current frame and return it as result of the call expression
- Introduce \(<\text{fun name}>\) into current frame, assigned a reference to the function object

Define

Python 2.7

```
def foo(x):
    return x+1

def bar(f, y):
    def cal(z):
        return f(z)
    bar(3, 4, x=x+1)

def <fun name>(<var list>):
    <suite of statements>
```

Sequence of Statements

- Evaluate each statement in sequence
- Introducing new variables up updating objects with each
### Conditional Statement

if:
   < true suite of statements >
else:
   <false suite>

- Evaluate if
- if it yields a truthy result, evaluate <true suite>
- Otherwise, if else: present, evaluate <false suite>

### Call Expressions

f( ... )

- Evaluate the operand expressions (recursively)
- Evaluate “function” expression to get function to apply
- This may involve function return values or “.” or ...
- Check that it is of function type
- If not, raise exception
- Apply function to resulting values to produce result

### Functions plus conditionals ...

- Recursion

### While Statement

while:
   < suite of statements >
else:
   < exit suite>

- Repeatedly evaluate
- If it yields a truthy result, evaluate <suite>
- Otherwise, if else: is present evaluate <exit suite>
- continue skips remain statements in suite
- break exits loop skipping <exit suite>

### For Statement

for <var list> in:
   < suite of statements >
else:
   < exit suite>

- Evaluate to get an iterable
- Repeatedly bind <var list> to next
- Evaluate <suite> with these bindings
- Until StopIteration is raised
- if else: is present evaluate <exit suite>

### Try statement

try:
   < suite of statements >
except as <var>:
   < except suite>

- Evaluate suite of statements
- If exception is raised which matches
- Evaluate except suite is var bound to exception object
Class statement

class <classname> ( <inheritance> ):
   < suite of statements >
   
   • If present, evaluate the inheritance list to obtain a class object or class type.
   • Create new namespace for classname
   • Evaluate <suite> in a new execution frame using a newly created namespace and global namespace
     Typically sequence of define statements
   • self in define for methods, self otherwise for object attributes
   • vars in class namespace for class attributes
   • Return resulting class object

operator

. <var>

   • References <var> in namespace of

with statement

with <var> as <var> [ , more ] :
   < suite of statements >

   • Evaluate suite of statements with vars bound to results of corresponding

Comprehension expressions

[ < for <var list> in ]
[ < for <var list> in <if> ]

   • Iteratively,
     – Evaluate next
     – If present, evaluate on it
     – Evaluate
     – until stop_iter exception
       – Collect all resulting values into result object

Software Design Patterns

• Higher Order Functions
• Recursion
• Data Parallel – Map-Reduce
• Abstract Data Types
  – Constructors, Selectors, Actions
• Object Oriented Programming
  – Encapsulation of behavior
• Iterators and Generators
  – Classes with __iter__ and __next__
  – yield statement

Uses of Computational Thinking

• Computational concepts model the world. Programming languages are mathematical formalisms just like any other: linear algebra, differential equations, statistics…
• Plus: Automatic verification of the model.
More CS:
• CS61b: More programming
• CS61c: Machine architecture (how the bits are moved)

So now …
Go model and change the world …