CS61A Lecture 23
Py

Jom Magrotker
UC Berkeley EECS
July 26, 2012

Computer Science in the News

http://www.theverge.com/2012/7/26/3188043/robot-baby-osaka-university-asada-laboratory

2

Today

- Interpretation: Review and More!
- Py, Python written in Python.

3

How do we structure our programs?
What functions do we need?
What classes should we design?

Structure

And

Interpretation

How can we make the computer understand our code?
How does the computer currently understand our code?

of

Computer Programs

Review: Interpretation

`>>>` What happens here?

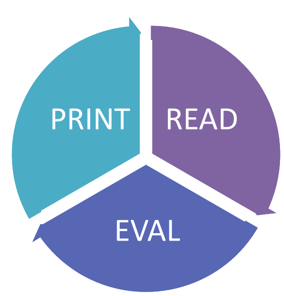
The main question for this module is:

What exactly happens at the prompt for the interpreter?

More importantly, what *is* an interpreter?

5

Review: Interpretation



6

Review: Evaluation

Evaluation finds the value of an expression, using its corresponding expression tree.

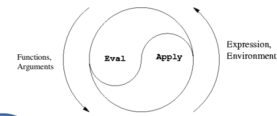
It discovers the form of an expression and then executes the corresponding evaluation rule.

Review: Eval and Apply

The **`eval-apply` cycle** is essential to the evaluation of an expression, and thus to the interpretation of many computer languages. It is not specific to calc.

`eval` receives the expression (and in some interpreters, the environment) and returns a function and arguments;

`apply` applies the function on its arguments and returns another expression, which can be a value.

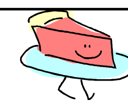


http://mitpress.mit.edu/sicp/full-text/sicp/book/img283.gif

Announcements

- Project 3 is due **Today**.
- Homework 11 is due **Saturday, July 28**.
  - Coming out today. Tom has been delayed in formatting it and what not because the computer he keeps that material on was having issues and was being repaired (took longer than expected).
  - We are **very** sorry about this and hope you'll understand.
- Starting ***next week*** we will be holding discussions in 320 instead of 310 Soda.

Py

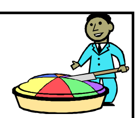


Tom developed a Python written in Python.

... He calls it Py. (He's actually quite proud of it ☺)

Py has:
- Numbers
- True/False
- None
- Primitive Functions
- Variables
- Def statements
  - Including return and nonlocal
- If statements
- Lambdas
- Comments

Py



If you're logged onto an instructional computer, you can find the code in ~cs61a/lib/python_modules/py/

Or, when logged in, you can run it using:

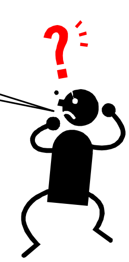
python3 –m py.interpreter

There might be a *small* number of updates before you see it in homework and lab, as it's still in beta.

Py: Why?

Many of you are wondering...

"Why would you write a Python using Python?!?!!?"



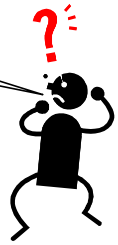
We admit that it sounds *sort of* redundant.

HOWEVER...

Py: Why?

Many of you are wondering…

"Why would you write a Python using Python?!?!!?"



It is instructive!

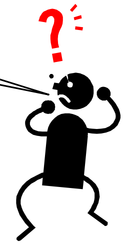
- We haven't seen a "full" programming language yet.
- We already know how this language works, so we can focus on the interpretation.

13

Py: Why?

Many of you are wondering…

"Why would you write a Python using Python?!?!!?"



Turns out it has been done before (and is sometimes better than other versions)!

pypy

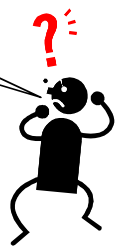
http://www.pypy.org

14

Py: Why so Small?

The other question you might be asking is…

"Why not all of Python?"



- We wanted to keep the code small.
- What we did implement is, more or less, enough to do the majority of "interesting" things a modern programming language does.

15

Py: Demo

If you were in lecture, you saw a small demonstration of the language here.

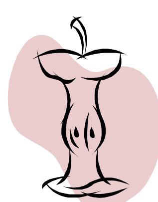
16

Looking at ~~Pie~~ Py

Unlike calc, it is organized into a variety of files.

Core parts:

```
interpreter.py
parsing.py
environments.py
statements.py
values.py
```



17

Looking at ~~Pie~~ Py

In principle, works the same as Calc.

What's different?

- Uses the environment model of computation.
  - We have variables and they are made available in various (possibly shared) environments.
  - Functions have parameters, bodies, and know what environment to extend when being applied.
- Statements (Expression Trees) are a bit smarter.
  - Handles evaluating themselves.
  - Uses attributes instead of general lists.

18

Looking at ~~Pie~~ Py: REPL

```
def read_eval_print_loop(interactive=True):
    while True:
        try:
            statement = py_read_and_parse()
            value = statement.evaluate(the_global_environment)
            if value != None and interactive:
                print(repr(value))
        except ...:
            ...
```

Repeatedly...

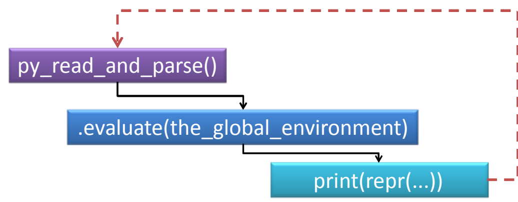
Read in a statement and produce a `Stmt` object.

Evaluate the statement we read. Use the global environment.

Print the (representation of) the resulting value, if there is one.

19

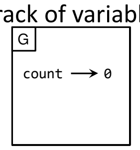
Looking at ~~Pie~~ Py: REPL



20

Looking at ~~Pie~~ Py: Environments

In this language, we use the environment model to keep track of variable names and their values.



In Py, this is implemented in a class `Environment`, which behaves much like a dictionary.

21

Looking at ~~Pie~~ Py: Environments

```
class Environment:
    ...
    def __init__(self, enclosing_environment=None):
        self.enclosing = enclosing_environment
        self.bindings = {}
        self.nonlocals = []
    ...
```

The environment that this one extends.

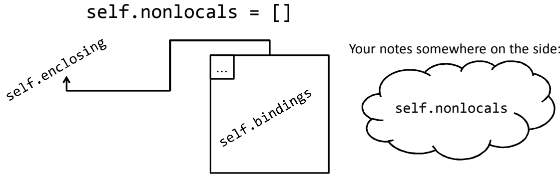
The bindings (variable names and their values) found in this frame.

Names that should not be bound in this frame.

22

Looking at ~~Pie~~ Py: Environments

```
class Environment:
    def __init__(self, enclosing_environment=None):
        self.enclosing = enclosing_environment
        self.bindings = {}
        self.nonlocals = []
```



23

Looking at ~~Pie~~ Py: Environments

```
class Environment:
    ...
    def is_global(self):
        return self.enclosing is None

    def note_nonlocal(self, var):
        self.nonlocals.append(var)
    ...
```

"Am I the global environment?"

Make a note of a variable name that is to be treated as nonlocal for this frame.

24

Looking at ~~Pie~~ Py: Environments

```
class Environment:
    ...
    def __getitem__(self, var):
        if var in self.bindings:
            return self.bindings[var]
        elif not self.is_global():
            return self.enclosing[var]
        else:
            raise ...
    ...
```

If I have the variable in my frame...

... give back the value.

If there are more frames to look through...

... look at the next frame.

Otherwise, we've got a problem!

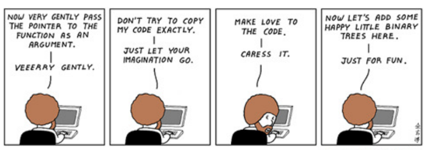
***Note:** In Py, nonlocal fills the role of **both** global and nonlocal in regular Python.

Looking at ~~Pie~~ Py: Environments

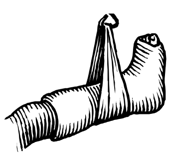
```
class Environment:
    ...
    def set_variable(self, var, val, is_nonlocal=False):
        if is_nonlocal and var not in self.bindings:
            self.enclosing.set_variable(var,
                                        val,
                                        is_nonlocal)
        elif not is_nonlocal or var in self.bindings:
            self.bindings[var] = val

    def __setitem__(self, var, val):
        self.set_variable(var,
                          val,
                          var in self.nonlocals)
    ...
```

If the variable is nonlocal and doesn't already exist, it belongs in a different frame.

Otherwise, add (or update) the binding in this frame.



The Joy of Programming with Bob Ross

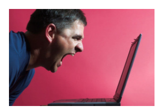


Break

http://invisiblebread.com/2012/06/new-workspace/
http://wondermark.com/wp-content/uploads/2012/07/sheep.gif

Looking at ~~Pie~~ Py: Statements

Yesterday, we saw the class `Exp`, which `Calc` used to represent expressions it read in and evaluated.



Py has something similar, which is the `Stmt` class. Unlike `Exp`:

- `Stmt` isn't limited to function calls.
- `Stmt` will handle evaluating itself, rather than having a separate function operate on them.

Looking at ~~Pie~~ Py: Statements

```
class AssignStmt(Stmt):
    def __init__(self, target, expr):
        self.target = target
        self.expr = expr

    def evaluate(self, env):
        env[self.target] = self.expr.evaluate(env)
```

Create the statement with all of the information that the statement is composed of.

Evaluate this statement in a given environment.

Looking at ~~Pie~~ Py: Call Expressions

```
class CallExpr(Expr):
    def __init__(self, op_expr, opnd_exprs):
        self.op_expr = op_expr
        self.opnd_exprs = opnd_exprs

    def evaluate(self, env):
        func_value = self.op_expr.evaluate(env)
        opnd_values = [opnd.evaluate(env)
                           for opnd in
                       self.opnd_exprs]
        return func_value.apply(opnd_values)
```

A call expression has an operator and a series of operands.

Evaluate the operator.

Evaluate the operands.

Apply the value of the operator onto the value of the operands.

Looking at ~~Pie~~ Py: Representing Functions

So we've seen the "eval" of the Py interpreter and how it keeps track of state.

$$f(x) = ax^2 + bx + c$$

How do we represent functions?

– Not that different from double bubbles!

Looking at ~~Pie~~ Py: Primitive Functions

```
class Function:
    def __init__(self, *args):
        raise NotImplementedError()

    def apply(self, operands):
        raise NotImplementedError()

class PrimitiveFunction(Function):
    def __init__(self, procedure):
        self.body = procedure

    def apply(self, operands):
        return self.body(*operands)
```

Primitives allow you access to a function "under the hood."

They simply call the "under the hood" procedure on the values when applied.

Looking at ~~Pie~~ Py: Primitive Functions

```
primitive_functions = [
    ("or", PrimitiveFunction(lambda x, y: x or y)),
    ("and", PrimitiveFunction(lambda x, y: x and y)),
    ("not", PrimitiveFunction(lambda x: not x)),
    ("eq", PrimitiveFunction(lambda x, y: x == y)),
    ("ne", PrimitiveFunction(lambda x, y: x != y)),
    ...
]

def setup_global():
    for name, primitive in primitive_functions:
        the_global_environment[name] = primitive
```

In the interpreter, there's a function which loads up the global environment with primitive functions, setup_global.

Looking at ~~Pie~~ Py: User-Defined Functions

```
class CompoundFunction(Function):
    def __init__(self, args, body, env):
        self.args = args
        self.body = body
        self.env = env

    def apply(self, operands):
        call_env = Environment(self.env)
        if len(self.args) != len(operands):
            raise TypeError("Wrong number of arguments passed to function!")
        for name, value in zip(self.args, operands):
            call_env[name] = value
        for statement in self.body:
            try:
                statement.evaluate(call_env)
            except StopFunction as sf:
                return sf.return_value
        return None
```

Make a new frame.

Bind the arguments.

Step into the environment and evaluate each statement of the body....

... until something is returned...

... or we reach the end (and return None).

Looking at ~~Pie~~ Py: User-Defined Functions

```
class StopFunction(BaseException):
    def __init__(self, return_value):
        self.return_value = return_value

class ReturnStmt(Stmt):
    def __init__(self, expr=None):
        self.expr = expr

    def evaluate(self, env):
        if self.expr is None:
            raise StopFunction(None)
        raise StopFunction(self.expr.evaluate(env))
```

Returning values from functions turns out to be harder than simply identifying one of the body statements as a return (what if the return is inside an if statement?). We implemented it as an exception. ReturnStmt raises one when evaluated. CompoundFunction catches it in apply.

Py: That's Pretty Much It

There's some other little bits that we might not have focused on today, but the code outside of parsing.py and testing.py ***should*** be relatively easy to follow once you get a sense of where things live.

We should note that this code is not necessarily representative of how all interpreters work (in fact that's just plain false). It is, however, *similar* to (but ***not*** the same as) the way things will work on the project.

CONCLUSION

- An interpreter is a function that, when applied to an expression, performs the actions required to evaluate that expression.
- We saw an interpreter for a subset of the Python language, Py, written in Python.
- The *read-eval-print loop* reads user input, evaluates the statement in the input, and prints the resulting value.
- To implement the environment model of computation, we use... Environments! Environments can be implemented like dictionaries.
- Using an object oriented approach, we can have our expression trees be responsible for evaluating themselves.
- Apply is now the job of the Function class, which represents function values.

37