TODAY

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

STRUCTURE AND INTERPRETATION OF COMPUTER PROGRAMS
The main question for this module is:
What exactly happens at the prompt for the interpreter?
More importantly, what is an interpreter?

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.

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.
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: WHY?

Many of you are wondering...

We admit that it sounds **sort of** redundant.

HOWEVER...

“Why would you write a Python using Python?!?!?”
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.

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

http://www.pypy.org

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.

If you were in lecture, you saw a small demonstration of the language here.
Unlike calc, it is organized into a variety of files.

Core parts:
- interpreter.py
- parsing.py
- environments.py
- statements.py
- values.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.

```python
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 ...:
            ...
```

```python
py_read_and_parse()  .evaluate(the_global_environment)  print(repr(...))
```
L**O**OKING AT **P**IE **P**Y: **E**NVIRONMENTS

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.

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

Names that should not be bound in this frame.

Your notes somewhere on the side:

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

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

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

    ...

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

Looking at 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)

Looking at 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 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)

Looking at 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)

Looking at 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 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)
**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.