- Obejct-Oriented Programming

Terminology: Attributes, Functions, and Methods

All objects have **attributes**, which are name-value pairs

Classes are objects too, so they have attributes

Instance attribute: attribute of an instance

Class attribute: attribute of the class of an instance

- Binding an object to a new name using assignment does not create a new object

Methods and Functions

Methods are functions defined in the suite of a class statement.

However methods that are accessed through an instance will be bound methods. **Bound methods** couple together a function and the object on which that method will be invoked. This means that when we invoke bound methods, the instance is automatically passed in as the first argument.

```
>>> a = Account("Tiffany")
>>> Account.deposit
<function>
>>> a.deposit
<bound method>
```

Accessing Attributes

There are built-in functions that can help us access attributes.

Using getattr, we can look up an attribute using a string instead.

- getattr(<expression>, <attribute_name (string)>)
- getattr(a, 'balance') is the same as a.balance
- getattr(Account, 'balance') is the same as Account.balance

Using hasattr, we can check if an attribute exists.

- hasattr(<expression>, <attribute_name (string)>)
- hasattr(a, 'balance') returns True
- hasattr(Account, 'balance') returns False

Assigning Attributes

Python Tutor link

We saw this before, but let's formalize the rules for assigning/re-assigning instance and class attributes.

<expression>.<name> = <value>

Change attributes for the **object of that dot expression.**

If the expression evaluates to an instance: then assignment sets an instance attribute, even if it exists in the class.

If the expression evaluates to a class: then assignment sets a class attribute

- Inheritance

Looking Up Attributes by Name

<expression> . <name>

To evaluate a dot expression:

1. Evaluate the <expression> to the left of the dot, which yields the object of the dot expression
2. <name> is matched against the instance attributes of that object; if an attribute with that name exists, its value is returned
3. If not, <name> is looked up in the class, which yields a class attribute value (if no such class attribute exists, an AttributeError is reported)

(demo: tom_account.interest, tom_account.noSuchAttribute)

Inheritance

Inheritance is a technique for relating classes together

A common use: Two similar classes differ in their degree of specialization

The specialized class may have the same attributes as the general class, along with some special-case behavior

```
class <Name>(<Base Class>):
    <suite>
```

Conceptually, the new subclass inherits attributes of its base class

The subclass may override certain inherited attributes

To look up a name in a class:

1. If it names an attribute in the class, return the attribute value.

2. Otherwise, look up the name in the base class, if there is one.

- Special Methods

In Python, all objects produce two string representations:

- The **str** is legible to humans
- The **repr** is legible to the Python interpreter

print(obj) -->str(obj) -->无__ str __， 查有没有 __ repr __

obj --> print(repr(obj))

The result of calling **str** on the value of an expression is what Python prints using the **print** function:

Implementing repr and str

The behavior of **repr** is slightly more complicated than invoking __repr__ on its argument:

- An instance attribute called __repr__ is ignored! Only class attributes are found
- *Question*: How would we implement this behavior?

The behavior of **str** is also complicated:

- An instance attribute called __str__ is ignored
- If no __str__ attribute is found, uses **repr** string
- *Question*: How would we implement this behavior?

```
def repr(x):
    return x.__repr__(x)

def repr(x):
    return x.__repr__()

def repr(x):
    return type(x).__repr__(x)

def repr(x):
    return type(x).__repr__()

def repr(x):
    return super(x).__repr__()
```

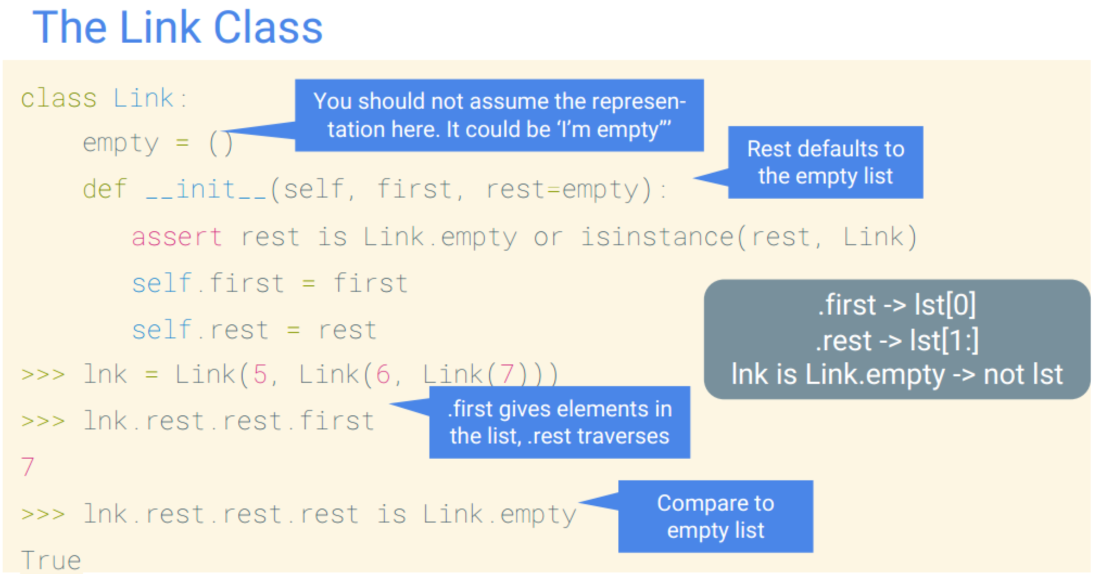
demo_1

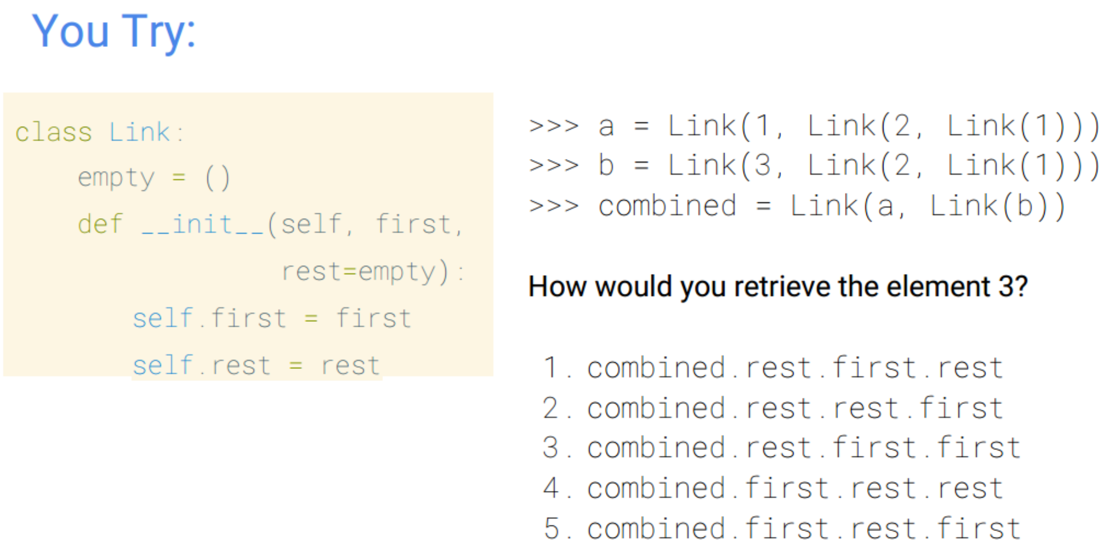
```python
# demo_1
class Dog:
    """ A dog """
    def __init__(self):
        self.__repr__ = lambda: 'teddy'
        self.__str__ = lambda: 'a teddy'

    def __repr__(self):
        return 'Dog()'

    def __str__(self):
        return 'a dog'

# (4) replace the built-in repr() and str()
def repr(x):
    return type(x).__repr__(x)

# Note this code (from cs61a) is not the real implementation for str()
def str(x):
    t = type(x)
    if hasattr(t, '__str__'):
        return t.__str__(x)
    else:
        return repr(x)

teddy = Dog()
print(teddy)
print(repr(teddy))
print(str(teddy))
print(teddy.__repr__())
print(teddy.__str__())

a dog
Dog()
a dog
teddy
a teddy
```

```python
# demo_2: __add__
class Ratio:
    def __init__(self, n, d):
        self.numer = n
        self.denom = d

    def __repr__(self):
        return 'Ratio({0}, {1})'.format(self.numer, self.denom)

    def __str__(self):
        return '{0}/{1}'.format(self.numer, self.denom)

    def __add__(self, other):
        if isinstance(other, int):
            n = self.numer + self.denom * other
            d = self.denom
            # (5) type coercion: convert one value to match the type of the
another
        elif isinstance(other, float):
```

```python
            return float(self) + other
        elif isinstance(other, Ratio):
             # (2)
            n = self.numer * other.denom + self.denom * other.numer
            d = self.denom * other.denom
        g = gcd(n, d)
        return Ratio(n//g, d//g)

    # (4) a+b: a.__add__(b), b.__radd__(a)
    __radd__ = __add__

    def __float__(self):
            return self.numer/self.denom

def gcd(n, d):
    while n != d:
        n, d = min(n, d), abs(n-d)
    return n
```

- Linked Lists & Trees





- Scheme

Assigning values to names

The define special form assigns a value to a name:

(define <name> <expr>)

How to evaluate:
Step 1. Evaluate the given expression.
Step 2. Bind the resulting value to the given name in the current frame.
Step 3. Return the name as a symbol.

```
scm> (define x (+ 3 4))
x
scm> x
7
scm> (define x (+ x 5))
x
scm> x
12
```

Control flow

The if special form allows us to evaluate an expression based on a condition:

(if <predicate> <if-true> <if-false>)

How to evaluate:
Step 1. Evaluate the <predicate>.
Step 2. If <predicate> evaluates to anything but #f, evaluate <if-true> and return the value. Otherwise, evaluate <if-false> if provided and return the value.

#f is the only Falsy value in Scheme

```
scm> (if #t 3 5)
3
scm> (if 0 (+ 1 0) (/ 1 0))
1
scm> (if (> 10 1) (* 5 6))
30
scm> (if (not 4) 1 (if #f 5 6))
6
```

Defining functions with names

The second version of define is a shorthand for creating a function with a name:

(define (<name> <param1> <param2> ...) <body>)

How to evaluate:
Step 1. Create a lambda procedure with the given parameters and body.
Step 2. Bind it to the given name in the current frame.
Step 3. Return the function name as a symbol.

```
scm> (define (square x) (* x x))
square
scm> square
(lambda (x) (* x x))
scm> (square 4)
16
scm> (square -10)
100
```

(Demo_2)

Lambda Expressions

The lambda special form returns a lambda procedure.

(lambda (<param1> <param2> ...) <body>)

How to evaluate:
Step 1. Create a lambda procedure with the given parameters and body.
Step 2. Return the lambda procedure.

The body expression is evaluated when the lambda procedure is applied.

```
scm> (lambda (x) (* x x))
(lambda (x) (* x x))
scm> ((lambda (x) (* x x)) 5)
25
scm> (define square (lambda (x) (* x x)))
square
scm> (square 4)
16
```

```
scm> (define x 5)
x
scm> (lambda (x y) (print 2))
(lambda (x y) (print 2))
scm> ((lambda (x) (print x)) 1)
1
scm> (define f (lambda () #f))
f
scm> (if f x (+ x 1))
5
scm> (if (f) (print 5) 6)
6
scm> (+ (if 1 2 3) (if 4 5 6))
7
```

Pairs

cons: construct
car and cdr: historical reason (Lisp on IBM 704)

- Pairs are created using the **cons** expression in Scheme
- **car** selects the first element in a pair
- **cdr** selects the second element in a pair
- The second element of a pair must be another pair, or **nil (empty)**

```
scm> (define x (cons 1 (cons 3 nil))
x
scm> x
(1 3)
scm> (car x)
1
scm> (cdr x)
(3)
```

Symbolic Programming

Symbols normally refer to values; how do we refer to symbols?

```
>(define a 1)
>(define b 2)
>(list a b)
(1 2)
```

No sign of "a" and "b" in the resulting value

Quotation is used to refer to symbols directly in Lisp.

```
>(list 'a 'b)
(a b)
>(list 'a b)
(a 2)
```

Short for (quote a), (quote b): Special form to indicate that the expression itself is the value.

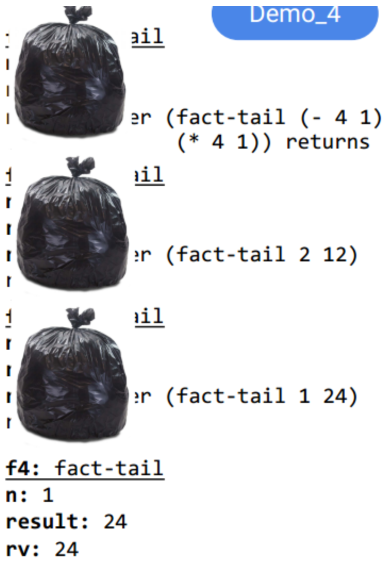
Quotation can also be applied to combinations to form lists.

```
>'(a b c)
(a b c)
>(car '(a b c))
a
>(cdr '(a b c))
(b c)
```

Tail calls

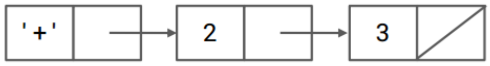
```
(define (fact n)
  (define (fact-tail n result)
    (if (<= n 1)
        result
        (fact-tail (- n 1) (* n result))))
  (fact-tail n 1))

fact(4)
```

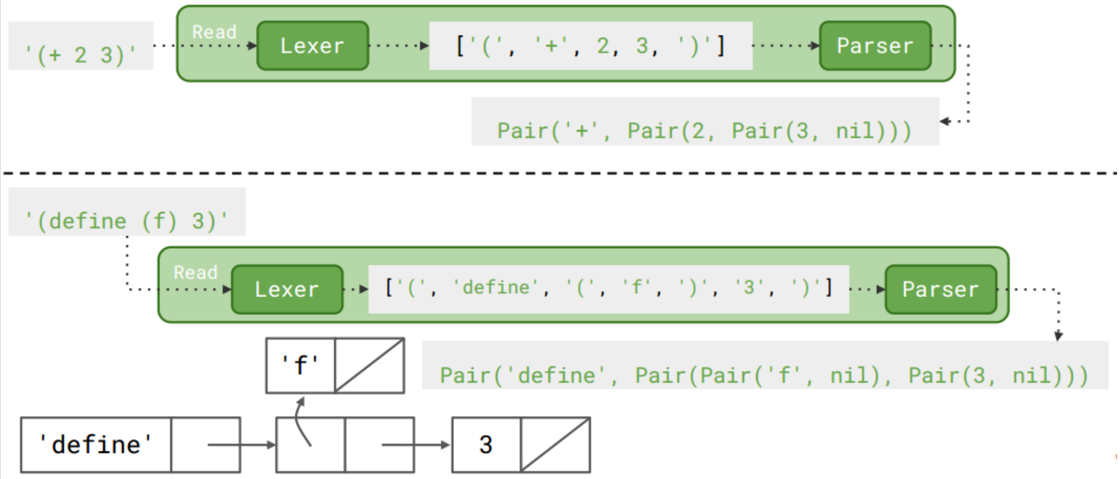
Number of frames the same regardless of input size!



- Interpreter

Reading Combinations





Special Case: quote

Recall that the quote special form can be invoked in two ways:

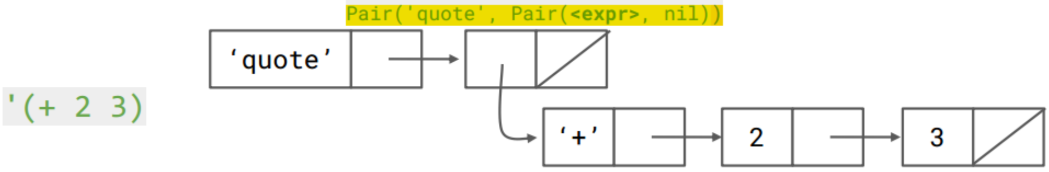
(quote <expr>) '<expr>

```
scm> 'hello
hello
scm> (quote hello)
hello
```

```
scm> '(1 2 3 4)
(1 2 3 4)
scm> (quote (1 2 3 4))
(1 2 3 4)
```

The special ' syntax gets converted by the reader into a quote expression, which is a list with 2 elements:

Pair('quote', Pair(<expr>, nil))



Pair('quote', Pair(Pair('+', Pair(2, Pair(3, nil))), nil))

Counting eval/apply calls: built-in procedures

How many calls to eval and apply are made in order to evaluate this expression?

(+ 2 (* 4 1) 5)

- eval(Pair('+',
  Pair(2,
  Pair(Pair('*', Pair(4, Pair(1, nil))),
  Pair(5, nil)))))
  - eval('+')
  - eval(2)
  - eval(Pair('*', Pair(4, Pair(1, nil))))
    - eval('*')
    - eval(4)
    - eval(1)
    - apply(BuiltinProc(scheme_mul), [4, 1])
  - eval(5)
  - apply(BuiltinProc(scheme_add), [2, 4, 5])

 # calls:
eval: 8
apply: 2

- Macros

Evaluating Macros

Recall evaluation procedure used for regular call expressions:

1. Evaluate the operator sub-expression, which evaluates to a regular procedure.
2. Evaluate the operand expressions in order.
3. Apply the procedure to the evaluated operands.

Macros, on the other hand, do the following:

1. Evaluate the operator sub-expression, which evaluates to a macro procedure.
2. Apply the macro procedure to the operand expressions *without evaluating them first*.
3. Evaluate the expression returned by the macro procedure in the frame the macro was called in

For Macro

Demo_6

Scheme doesn't have for loops, but thanks to macros, we can add them.

```
scm> (for x in '(1 2 3 4) do (* x x))
(1 4 9 16)
scm> (map (lambda (x) (* x x)) '(1 2 3 4))
(1 4 9 16)
```

```
(define-macro (for sym in vals do expr)
  (list 'map (list 'lambda (list sym) expr) vals))
```

Quasi-quoting

Quasiquotation allows you to have some parts of a list be read literally and some parts be evaluated.

It's especially useful for constructing code in macros.

```
(define-macro (for sym vals expr)
  (list 'map (list 'lambda (list sym) expr) vals))

(define-macro (for sym vals expr)
  `(map (lambda (,sym) ,expr) ,vals))
```

Short for (quasiquote …)

Short for (unquote …)

Much cleaner, right?

- Streams
- SQL