Import statement

```
from math import pi
tau = 2 * pi
```

Assignment statement

Global frame

Name	Value
pi	3.1416

Binding

Code (left):

Statements and expressions
Red arrow points to next line.
Gray arrow points to the line just executed

Frames (right):

A name is bound to a value

In a frame, there is at most one binding per name

```
from operator import mul
def square(x):
    return mul(x, x)
square(-2)
```

Built-in function

Global frame
mul → func mul(...) [parent=Global]
square → func square(x) [parent=Global]

Intrinsic name of function called

User-defined function

f1: square [parent=Global]
x -2
Return value 4

Local frame

Formal parameter bound to argument

Return value is not a binding!

```
from operator import mul
def square(x):
    return mul(x, x)
square(square(3))
```

Global frame: mul, square
f1: square [parent=Global] x 3, Return value 9
f2: square [parent=Global] x 9, Return value 81

A name evaluates to the value bound to that name in the earliest frame of the current environment in which that name is found.

Evaluation rule for call expressions:

1. Evaluate the operator and operand subexpressions.
2. Apply the function that is the value of the operator subexpression to the arguments that are the values of the operand subexpressions.

Applying user-defined functions:

1. Create a new local frame with the same parent as the function that was applied.
2. Bind the arguments to the function's formal parameter names in that frame.
3. Execute the body of the function in the environment beginning at that frame.

Execution rule for def statements:

1. Create a new function value with the specified name, formal parameters, and function body.
2. Its parent is the first frame of the current environment.
3. Bind the name of the function to the function value in the first frame of the current environment.

Execution rule for assignment statements:

1. Evaluate the expression(s) on the right of the equal sign.
2. Simultaneously bind the names on the left to those values, in the first frame of the current environment.

Execution rule for conditional statements:

Each clause is considered in order.
1. Evaluate the header's expression.
2. If it is a true value, execute the suite, then skip the remaining clauses in the statement.

Evaluation rule for or expressions:

1. Evaluate the subexpression <left>.
2. If the result is a true value v, then the expression evaluates to v.
3. Otherwise, the expression evaluates to the value of the subexpression <right>.

Evaluation rule for and expressions:

1. Evaluate the subexpression <left>.
2. If the result is a false value v, then the expression evaluates to v.
3. Otherwise, the expression evaluates to the value of the subexpression <right>.

Evaluation rule for not expressions:

1. Evaluate <exp>; The value is True if the result is a false value, and False otherwise.

Execution rule for while statements:

1. Evaluate the header's expression.
2. If it is a true value, execute the (*whole*) suite, then return to step 1.

208
mul(add(2, mul(4, 6)), add(3, 5))
mul | 26: add(2, mul(4, 6)) | 8: add(3, 5)
add | 2 | 24: mul(4, 6) | add | 3 | 5
mul | 4 | 6

Pure Functions

-2 → *abs*(number): → 2

2, 10 → *pow*(x, y): → 1024

Non-Pure Functions

-2 → *print*(...): → None

display "-2"

Defining:

```
>>> def square( x ):
        return mul(x, x)
```

Formal parameter

Return expression

Def statement

Body (*return statement*)

Call expression: square(2+2)

operand: 2+2
argument: 4

operator: square
function: func square(x)

Calling/Applying: 4 → *square*(x): → 16

return mul(x, x)

Argument

Intrinsic name

Return value

Compound statement

Clause

<header>:
    <statement>
    <statement>
    ...
<separating header>:
    <statement>
    <statement>
    ...
...

Suite

1 statement,
3 clauses,
3 headers,
3 suites,
2 boolean contexts

```
def abs_value(x):
    if x > 0:
        return x
    elif x == 0:
        return 0
    else:
        return -x
```

```
def f(x, y):
    return g(x)

def g(a):
    return a + y

result = f(1, 2)
```

"y" is not found

Error

"y" is not found

Global frame
f → func f(x, y) [parent=Global]
g → func g(a) [parent=Global]

f1: f [parent=Global] x 1, y 2

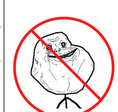
f2: g [parent=Global] a 1

- An environment is a sequence of frames
- An environment for a non-nested function (no def within def) consists of one local frame, followed by the global frame

```
from operator import mul
def square(square):
    return mul(square, square)
square(4)
```

A call expression and the body of the function being called are evaluated in different environments

1 A Global frame: mul, square
2 B
1 f1: square [parent=Global] square 4, Return value 16



Higher-order function: A function that takes a function as an argument value or returns a function as a return value

Nested def statements: Functions defined within other function bodies are bound to names in the local frame

```
def fib(n):
    """Compute the nth Fibonacci number, for N >= 1."""
    pred, curr = 0, 1    # Zeroth and first Fibonacci numbers
    k = 1                # curr is the kth Fibonacci number
    while k < n:
        pred, curr = curr, pred + curr
        k = k + 1
    return curr
```

```
def cube(k):
    return pow(k, 3)

def summation(n, term):
    """Sum the first n terms of a sequence.

    >>> summation(5, cube)
    225
    """
    total, k = 0, 1
    while k <= n:
        total, k = total + term(k), k + 1
    return total
```

Function of a single argument (not called term)

A formal parameter that will be bound to a function

The cube function is passed as an argument value

$0 + 1^3 + 2^3 + 3^3 + 4^3 + 5^3$

The function bound to term gets called here

```
square = lambda x,y: x * y
```

Evaluates to a function. No "return" keyword!

A function
with formal parameters x and y
that returns the value of "x * y"

Must be a single expression

```
square = lambda x: x * x
```

VS

```
def square(x):
    return x * x
```

- Both create a function with the same domain, range, and behavior.
- Both functions have as their parent the environment in which they were defined.
- Both bind that function to the name square.
- Only the def statement gives the function an intrinsic name.

```
def make_adder(n):
    """Return a function that takes one argument k and returns k + n.

    >>> add_three = make_adder(3)
    >>> add_three(4)
    7
    """
    def adder(k):
        return k + n
    return adder
```

A function that returns a function

The name add_three is bound to a function

A local def statement

Can refer to names in the enclosing function

When a function is defined:

1. Create a **function value**: func <name>(<formal parameters>)
2. Its parent is the current frame.

   f1: make_adder   func adder(k) [parent=f1]

3. Bind <name> to the **function value** in the current frame (which is the first frame of the current environment).

When a function is called:

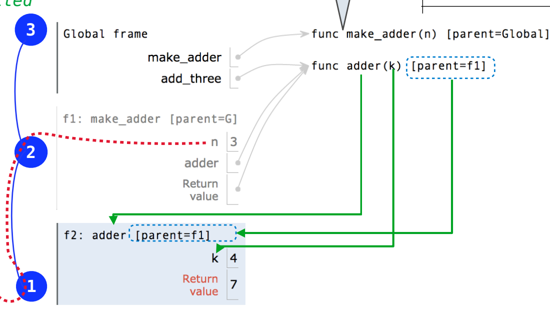
1. Add a **local frame**, titled with the <name> of the function being called.
2. Copy the parent of the function to the **local frame**: [parent=<label>]
3. Bind the <formal parameters> to the arguments in the **local frame**.
4. Execute the body of the function in the environment that starts with the **local frame**.

- Every user-defined **function** has a *parent frame* (often global)
- The parent of a **function** is the frame in which it was *defined*
- Every local **frame** has a *parent frame* (often global)
- The parent of a **frame** is the parent of the function *called*

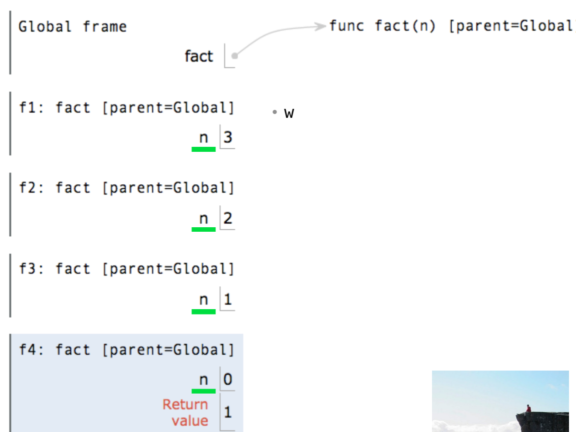
A function's signature has all the information to create a local frame

```
def make_adder(n):
    def adder(k):
        return k + n
    return adder

add_three = make_adder(3)
add_three(4)
```

Nested def




```
def fact(n):
    if n == 0:
        return 1
    else:
        return n * fact(n-1)

fact(3)
```

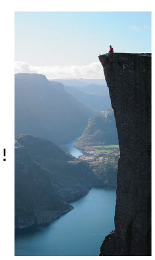



```
def compose1(f, g):
    """Return a function h that composes f and g.

    >>> compose1(square, make_adder(2))(3)
    25
    """
    def h(x):
        return f(g(x))
    return h
```

Return value of make_adder is an argument to compose1

Is fact implemented correctly?

1. Verify the base case.
2. Treat fact as a functional abstraction!
3. Assume that fact(n-1) is correct.
4. Verify that fact(n) is correct, assuming that fact(n-1) correct.

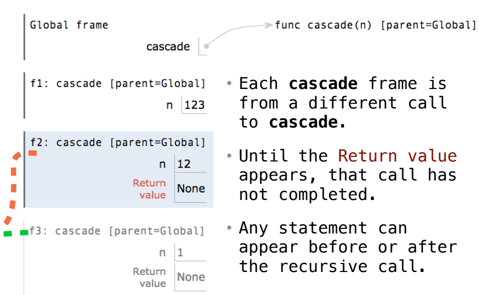


Anatomy of a recursive function:

- The **def statement header** is similar to other functions
- Conditional statements check for **base cases**
- Base cases are evaluated **without recursive calls**
- Recursive cases are evaluated **with recursive calls**

```
def sum_digits(n):
    """Return the sum of the digits of positive integer n."""
    if n < 10:
        return n
    else:
        all_but_last, last = n // 10, n % 10
        return sum_digits(all_but_last) + last
```

```
def cascade(n):
    if n < 10:
        print(n)
    else:
        print(n)
        cascade(n//10)
        print(n)

cascade(123)
```



Program output:

```
123
12
1
12
```

- Each **cascade** frame is from a different call to **cascade.**
- Until the Return value appears, that call has not completed.
- Any statement can appear before or after the recursive call.

- Recursive decomposition: finding simpler instances of a problem.
- E.g., **count_partitions(6, 4)**
- Explore two possibilities:
  - Use at least one 4
  - Don't use any 4
- Solve two simpler problems:
  - **count_partitions(2, 4)**
  - **count_partitions(6, 3)**
- Tree recursion often involves exploring different choices.

```
def count_partitions(n, m):
    if n == 0:
        return 1
    elif n < 0:
        return 0
    elif m == 0:
        return 0
    else:
        with_m = count_partitions(n-m, m)
        without_m = count_partitions(n, m-1)
        return with_m + without_m
```

```
1
12
123
1234
123
12
1
```

```
def inverse_cascade(n):
    grow(n)
    print(n)
    shrink(n)

def f_then_g(f, g, n):
    if n:
        f(n)
        g(n)

grow =   lambda n: f_then_g(grow,  print,  n//10)
shrink = lambda n: f_then_g(print, shrink, n//10)
```

n:	0,	1,	2,	3,	4,	5,	6,	7,	8,
fib(n):	0,	1,	1,	2,	3,	5,	8,	13,	21,

```
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-2) + fib(n-1)
```

```
from operator import floordiv, mod
def divide_exact(n, d):
    """Return the quotient and remainder of dividing N by D.

    >>> q, r = divide_exact(2012, 10)
    >>> q
    201
    >>> r
    2
    """
    return floordiv(n, d), mod(n, d)
```

Multiple assignment to two names

Multiple return values, separated by commas

Numeric types in Python:

```
>>> type(2)
<class 'int'>
```
Represents integers exactly

```
>>> type(1.5)
<class 'float'>

>>> type(1+1j)
<class 'complex'>
```
Represents real numbers approximately

Rational implementation using functions:

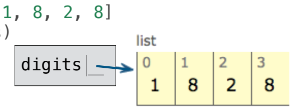
```
def rational(n, d):
    def select(name):
        if name == 'n':
            return n
        elif name == 'd':
            return d
    return select
```
This function represents a rational number

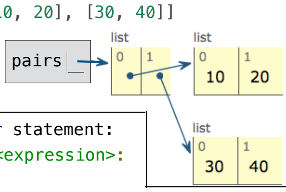
Constructor is a higher-order function

```
def numer(x):
    return x('n')

def denom(x):
    return x('d')
```
Selector calls x

Lists:

```
>>> digits = [1, 8, 2, 8]
>>> len(digits)
4
>>> digits[3]
8
>>> [2, 7] + digits * 2
[2, 7, 1, 8, 2, 8, 1, 8, 2, 8]
>>> pairs = [[10, 20], [30, 40]]
>>> pairs[1]
[30, 40]
>>> pairs[1][0]
30
```





Executing a for statement:

```
for <name> in <expression>:
    <suite>
```

1. Evaluate the header <expression>, which must yield an iterable value (a sequence)
2. For each element in that sequence, in order:
   A. Bind <name> to that element in the current frame
   B. Execute the <suite>

Unpacking in a for statement:

A sequence of fixed-length sequences

```
>>> pairs=[[1, 2], [2, 2], [3, 2], [4, 4]]
>>> same_count = 0
```

A name for each element in a fixed-length sequence

```
>>> for x, y in pairs:
...     if x == y:
...         same_count = same_count + 1

>>> same_count
2
```

..., -3, -2, -1, 0, 1, 2, 3, 4, ...

range(-2, 2)

Length: ending value - starting value
Element selection: starting value + index

```
>>> list(range(-2, 2))
[-2, -1, 0, 1]
```
List constructor

```
>>> list(range(4))
[0, 1, 2, 3]
```
Range with a 0 starting value

Membership:

```
>>> digits = [1, 8, 2, 8]
>>> 2 in digits
True
>>> 1828 not in digits
True
```

Slicing:

```
>>> digits[0:2]
[1, 8]
>>> digits[1:]
[8, 2, 8]
```
Slicing creates a new object

List comprehensions:

[<map exp> for <name> in <iter exp> if <filter exp>]

Short version: [<map exp> for <name> in <iter exp>]

A combined expression that evaluates to a list using this evaluation procedure:
1. Add a new frame with the current frame as its parent
2. Create an empty *result list* that is the value of the expression
3. For each element in the iterable value of <iter exp>:
   A. Bind <name> to that element in the new frame from step 1
   B. If <filter exp> evaluates to a true value, then add the value of <map exp> to the result list

The result of calling **repr** on a value is what Python prints in an interactive session

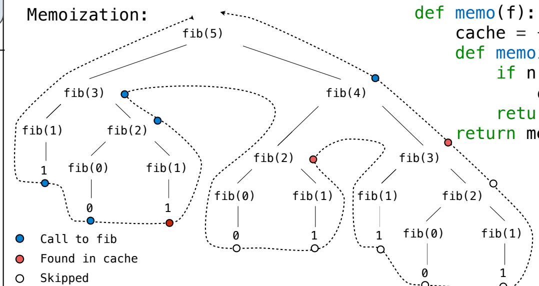
The result of calling **str** on a value is what Python prints using the **print** function

```
>>> 12e12
12000000000000.0
>>> print(repr(12e12))
12000000000000.0

>>> print(today)
2014-10-13
```

str and **repr** are both polymorphic; they apply to any object
repr invokes a zero-argument method __repr__ on its argument

```
>>> today.__repr__()
'datetime.date(2014, 10, 13)'

>>> today.__str__()
'2014-10-13'
```

Memoization:

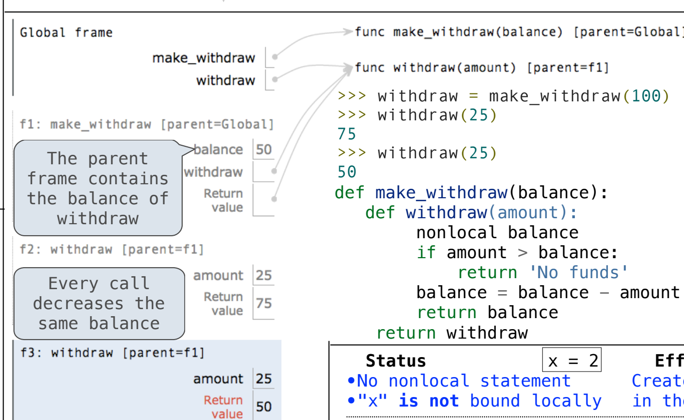



```
def memo(f):
    cache = {}
    def memoized(n):
        if n not in cache:
            cache[n] = f(n)
        return cache[n]
    return memoized
```

Type dispatching: Look up a cross-type implementation of an operation based on the types of its arguments
Type coercion: Look up a function for converting one type to another, then apply a type-specific implementation.

$R(n) = \Theta(f(n))$ means that there are positive constants k_1 and k_2 such that $k_1 \cdot f(n) \leq R(n) \leq k_2 \cdot f(n)$ for all **n** larger than some **m**

$\Theta(b^n)$	Exponential growth. Recursive fib takes $\Theta(\phi^n)$ steps, where $\phi = \frac{1+\sqrt{5}}{2} \approx 1.61828$. Incrementing the problem scales R(n) by a factor
$\Theta(n^2)$	Quadratic growth. E.g., overlap. Incrementing n increases R(n) by the problem size n
$\Theta(n)$	Linear growth. E.g., factors or exp
$\Theta(\log n)$	Logarithmic growth. E.g., exp_fast. Doubling the problem only increments R(n)
$\Theta(1)$	Constant. The problem size doesn't matter




```
>>> withdraw = make_withdraw(100)
>>> withdraw(25)
75
>>> withdraw(25)
50
def make_withdraw(balance):
    def withdraw(amount):
        nonlocal balance
        if amount > balance:
            return 'No funds'
        balance = balance - amount
        return balance
    return withdraw
```

Strings as sequences:

```
>>> city = 'Berkeley'
>>> len(city)
8
>>> city[3]
'k'
>>> 'here' in "Where's Waldo?"
True
>>> 234 in [1, 2, 3, 4, 5]
False
>>> [2, 3, 4] in [1, 2, 3, 4]
False
```

List & dictionary mutation:

```
>>> a = [10]
>>> b = a
>>> a == b
True
>>> a.append(20)
>>> a == b
True
>>> a
[10, 20]
>>> b
[10, 20]
```

```
>>> a = [10]
>>> b = [10]
>>> a == b
True
>>> b.append(20)
>>> a
[10]
>>> b
[10, 20]
>>> a == b
False
```

```
>>> nums = {'I': 1.0, 'V': 5, 'X': 10}
>>> nums['X']
10
>>> nums['I'] = 1
>>> nums['L'] = 50
>>> nums
{'X': 10, 'L': 50, 'V': 5, 'I': 1}
>>> sum(nums.values())
66
>>> dict([(3, 9), (4, 16), (5, 25)])
{3: 9, 4: 16, 5: 25}
>>> nums.get('A', 0)
0
>>> nums.get('V', 0)
5
>>> {x: x*x for x in range(3,6)}
{3: 9, 4: 16, 5: 25}
```

```
>>> suits = ['coin', 'string', 'myriad']
>>> suits.pop()
'myriad'
>>> suits.remove('string')
>>> suits.append('cup')
>>> suits.extend(['sword', 'club'])
>>> suits[2] = 'spade'
>>> suits
['coin', 'cup', 'spade', 'club']
>>> suits[0:2] = ['diamond']
>>> suits
['diamond', 'spade', 'club']
>>> suits.insert(0, 'heart')
>>> suits
['heart', 'diamond', 'spade', 'club']
```
Remove and return the last element
Remove a value
Add all values
Replace a slice with values
Add an element at an index

Identity:
<exp0> **is** <exp1>
evaluates to True if both <exp0> and <exp1> evaluate to the same object
Equality:
<exp0> == <exp1>
evaluates to True if both <exp0> and <exp1> evaluate to equal values
Identical objects are always equal values

You can **copy** a list by calling the list constructor or slicing the list from the beginning to the end.

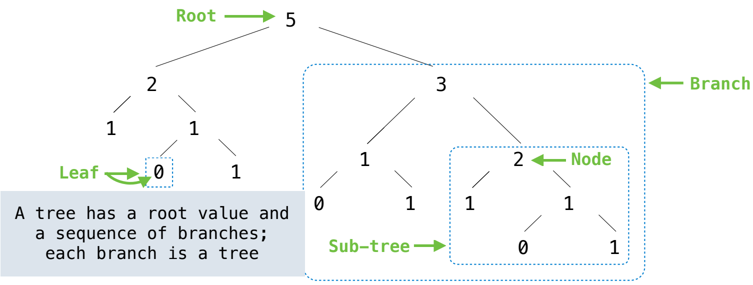
Constants: Constant terms do not affect the order of growth of a process
$\Theta(n)$ $\Theta(500 \cdot n)$ $\Theta(\frac{1}{500} \cdot n)$
Logarithms: The base of a logarithm does not affect the order of growth of a process
$\Theta(\log_2 n)$ $\Theta(\log_{10} n)$ $\Theta(\ln n)$
Nesting: When an inner process is repeated for each step in an outer process, multiply the steps in the outer and inner processes to find the total number of steps

```
def overlap(a, b):
    count = 0
    for item in a:
        for item in b:
            count += 1
    return count
```
Outer: length of a
Inner: length of b

If a and b are both length **n**, then overlap takes $\Theta(n^2)$ steps
Lower-order terms: The fastest-growing part of the computation dominates the total
$\Theta(n^2)$ $\Theta(n^2 + n)$ $\Theta(n^2 + 500 \cdot n + \log_2 n + 1000)$

Status	x = 2	Effect
•No nonlocal statement •"x" **is not** bound locally		Create a new binding from name "x" to number 2 in the first frame of the current environment
•No nonlocal statement •"x" **is** bound locally		Re-bind name "x" to object 2 in the first frame of the current environment
•nonlocal x •"x" **is** bound in a non-local frame		Re-bind "x" to 2 in the first non-local frame of the current environment in which "x" is bound
•nonlocal x •"x" **is not** bound in a non-local frame		SyntaxError: no binding for nonlocal 'x' found
•nonlocal x •"x" **is** bound in a non-local frame •"x" also bound locally		SyntaxError: name 'x' is parameter and nonlocal

Tree data abstraction:

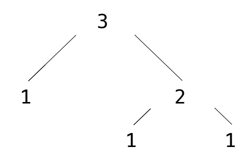



```
def tree(root, branches=[]):
    for branch in branches:
        assert is_tree(branch)
    return [root] + list(branches)
def root(tree):
    return tree[0]
def branches(tree):
    return tree[1:]
def is_tree(tree):
    if type(tree) != list or len(tree) < 1:
        return False
    for branch in branches(tree):
        if not is_tree(branch):
            return False
    return True
def is_leaf(tree):
    return not branches(tree)
def leaves(tree):
    """The leaf values in tree.

    >>> leaves(fib_tree(5))
    [1, 0, 1, 0, 1, 1, 0, 1]
    """
    if is_leaf(tree):
        return [root(tree)]
    else:
        return sum([leaves(b) for b in branches(tree)], [])
```

Verifies the tree definition

Creates a list from a sequence of branches

Verifies that tree is bound to a list




```
>>> tree(3, [tree(1),
...          tree(2, [tree(1),
...                   tree(1)])])
[3, [1], [2, [1], [1]]]
```

```
def fib_tree(n):
    if n == 0 or n == 1:
        return tree(n)
    else:
        left = fib_tree(n-2),
        right = fib_tree(n-1)
        fib_n = root(left) + root(right)
        return tree(fib_n, [left, right])
```

```
class Tree:
    def __init__(self, entry, branches=()):
        self.entry = entry
        for branch in branches:
            assert isinstance(branch, Tree)
        self.branches = list(branches)
    def is_leaf(self):
        return not self.branches
```

Built-in isinstance function: returns True if branch has a class that *is* **or** *inherits from* Tree

```
def fib_Tree(n):
    if n == 0 or n == 1:
        return Tree(n)
    else:
        left = fib_Tree(n-2)
        right = fib_Tree(n-1)
        fib_n = left.entry+right.entry
        return Tree(fib_n,[left, right])
```

```
def leaves(tree):
    if tree.is_leaf():
        return [tree.entry]
    else:
        return sum([leaves(b) for b in tree.branches], [])
```

```
class Link:
    empty = ()
    def __init__(self, first, rest=empty):
        self.first = first
        self.rest = rest
    def __getitem__(self, i):
        if i == 0:
            return self.first
        else:
            return self.rest[i-1]
    def __len__(self):
        return 1 + len(self.rest)
    def __repr__(self):
        if self.rest:
            rest_str = ', ' + repr(self.rest)
        else:
            rest_str = ''
        return 'Link({0}{1})'.format(self.first, rest_str)
```

Some zero length sequence

Yes, this call is recursive

Sequence abstraction special names:	
`__getitem__`	Element selection []
`__len__`	Built-in len function

Contents of the repr string of a Link instance

```
def extend_link(s, t):
    """Return a Link with the
    elements of s followed by
    those of t.
    """
    if s is Link.empty:
        return t
    else:
        return Link(s.first, extend_link(s.rest, t))

def map_link(f, s):
    if s is Link.empty:
        return s
    else:
        return Link(f(s.first), map_link(f, s.rest))
```

```
>>> s = Link(3, Link(4))
>>> extend_link(s, s)
Link(3, Link(4, Link(3, Link(4))))
>>> square = lambda x: x * x
>>> map_link(square, s)
Link(9, Link(16))
```

Python object system:

Idea: All bank accounts have a **balance** and an account **holder**; the **Account** class should add those attributes to each of its instances

A new instance is created by calling a class

```
>>> a = Account('Jim')
>>> a.holder
'Jim'
>>> a.balance
0
```

An account instance

balance: 0	holder: 'Jim'

When a class is called:
1. A new instance of that class is created:
2. The `__init__` method of the class is called with the new object as its first argument (named self), along with any additional arguments provided in the call expression.

`__init__` is called a constructor

self should always be bound to an instance of the Account class or a subclass of Account

```
class Account:
    def __init__(self, account_holder):
        self.balance = 0
        self.holder = account_holder
    def deposit(self, amount):
        self.balance = self.balance + amount
        return self.balance
    def withdraw(self, amount):
        if amount > self.balance:
            return 'Insufficient funds'
        self.balance = self.balance - amount
        return self.balance
```

Function call: all arguments within parentheses

```
>>> type(Account.deposit)
<class 'function'>
>>> type(a.deposit)
<class 'method'>
```

Method invocation: One object before the dot and other arguments within parentheses

```
>>> Account.deposit(a, 5)
10
>>> a.deposit(2)
12
```

Call expression

Dot expression

<expression> . <name>

The <expression> can be any valid Python expression.
The <name> must be a simple name.
Evaluates to the value of the attribute looked up by <name> in the object that is the value of the <expression>.

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
4. That value is returned unless it is a function, in which case a bound method is returned instead

Assignment statements with a dot expression on their left-hand side affect attributes for the object of that dot expression
- If the object is an instance, then assignment sets an instance attribute
- If the object is a class, then assignment sets a class attribute

Account class attributes

interest: ~~0.02~~ ~~0.04~~ 0.05
(withdraw, deposit, `__init__`)

Instance attributes of jim_account

balance: 0
holder: 'Jim'
interest: 0.08

Instance attributes of tom_account

balance: 0
holder: 'Tom'

```
>>> jim_account = Account('Jim')
>>> tom_account = Account('Tom')
>>> tom_account.interest
0.02
>>> jim_account.interest
0.02
>>> Account.interest = 0.04
>>> tom_account.interest
0.04
>>> jim_account.interest
0.04
```

```
>>> jim_account.interest = 0.08
>>> jim_account.interest
0.08
>>> tom_account.interest
0.04
>>> Account.interest = 0.05
>>> tom_account.interest
0.05
>>> jim_account.interest
0.08
```

```
class CheckingAccount(Account):
    """A bank account that charges for withdrawals."""
    withdraw_fee = 1
    interest = 0.01
    def withdraw(self, amount):
        return Account.withdraw(self, amount + self.withdraw_fee)
                          or
        return super().withdraw(      amount + self.withdraw_fee)
```

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.

```
>>> ch = CheckingAccount('Tom')  # Calls Account.__init__
>>> ch.interest     # Found in CheckingAccount
0.01
>>> ch.deposit(20)  # Found in Account
20
>>> ch.withdraw(5)  # Found in CheckingAccount
14
```

CS 61A Exam Scratch Paper

CS 61A Exam Scratch Paper

CS 61A Exam Scratch Paper

CS 61A Exam Scratch Paper