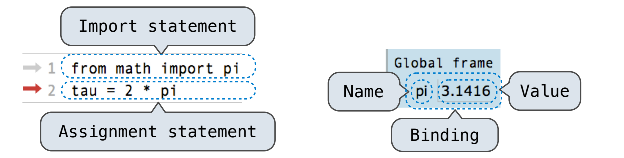


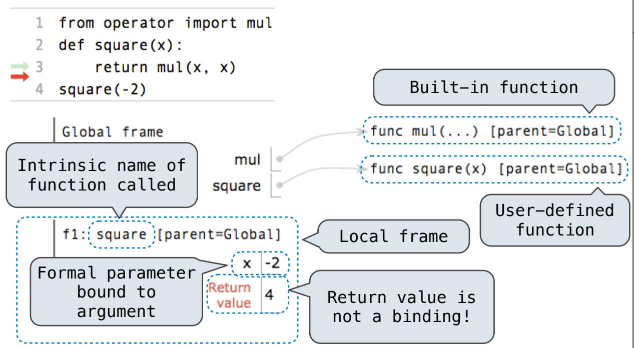
Code (left):

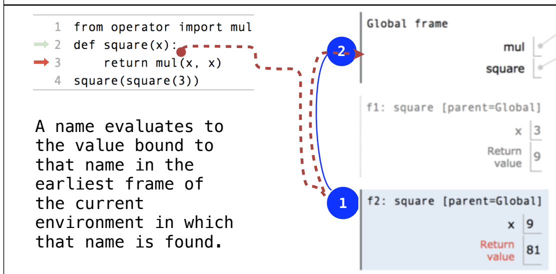
Statements and expressions
Red arrow points to next line.
Gray arrow points to the line just executed

Frames (right):

A name is bound to a value

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





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:

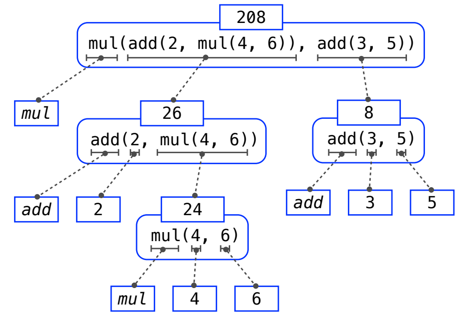
1. Evaluate the subexpression <left>.
2. If the result is a false value v, then the expression evaluates to v.
3. Otherwise, the expression evaluates to the value of the subexpression <right>.

Evaluation rule for not expressions:

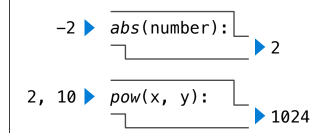
1. Evaluate <exp>; The value is True if the result is a false value, and False otherwise.

Execution rule for while statements:

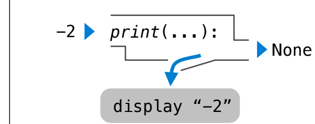
1. Evaluate the header's expression.
2. If it is a true value, execute the (*whole*) suite, then return to step 1.

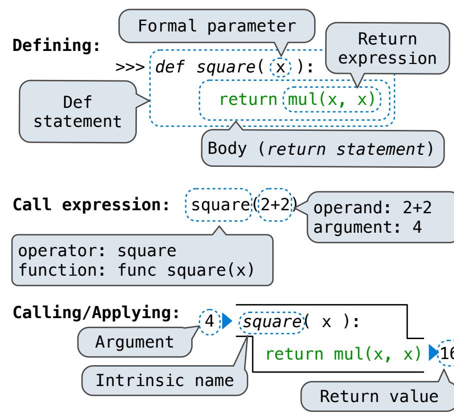


Pure Functions



Non-Pure Functions





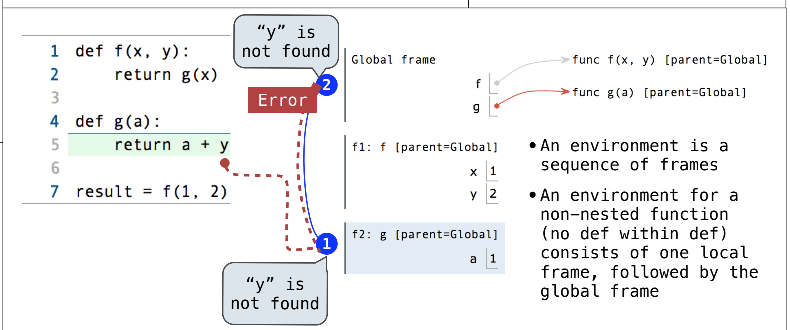

```
def strconcat( a, b ):
    print( a + " " + b )

strconcat( "hello", "world" )
```

hello world

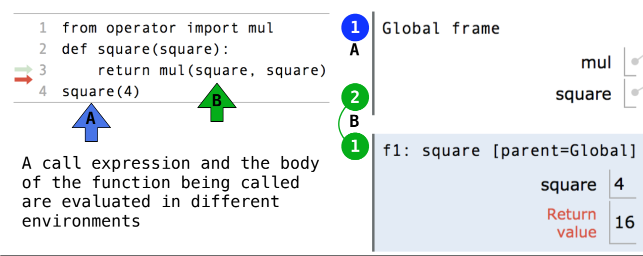
A and B:
  True if A is True and B is True
A or B:
  True if A is True or B is True
not A:
  True if A is False
  False if A is True

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

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

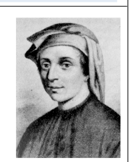


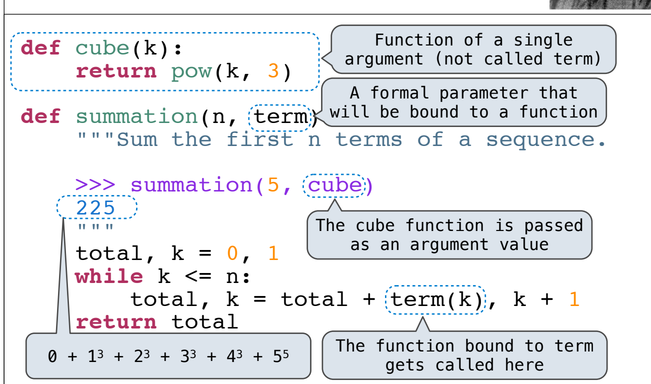
- An environment is a sequence of frames
- An environment for a non-nested function (no def within def) consists of one local frame, followed by the global frame



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

```
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
```





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

```
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

```
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

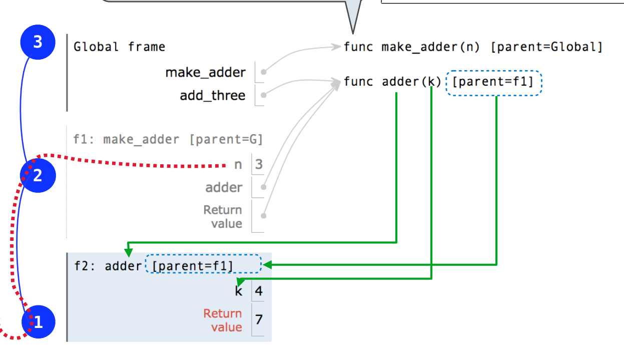
Can refer to names in the enclosing function

- 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




```
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.

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:

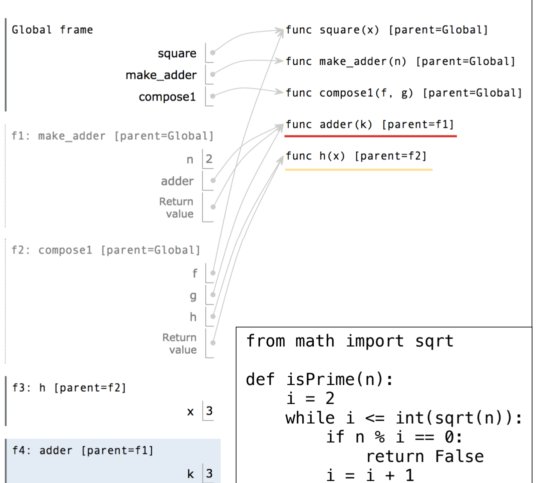
1. Add a **local frame**, titled with the *<name>* of the function being called.
2. Copy the parent of the function to the **local frame**: [parent=*<label>*]
3. Bind the *<formal parameters>* to the arguments in the **local frame**.
4. Execute the body of the function in the environment that starts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

```
>>> min(2, 1, 4, 3)
1
>>> max(2, 1, 4, 3)
4
>>> abs(-2)
2
>>> pow(2, 3)
8
>>> len('word')
4
>>> round(1.75)
2
>>> print(1, 2)
1 2
>>> float(5)
5.0
```

```
>>> 2 + 3
5
>>> 2 * 3
6
>>> 2 ** 3
8
>>> 5 / 3
1.6666666666666667
>>> 5 // 3
1
>>> 5 % 3
2
>>> str(5)
'5'
>>> int('5')
5
```

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

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

def compose1(f, g):
    def h(x):
        return f(g(x))
    return h

compose1(square, make_adder(2))(3)
```

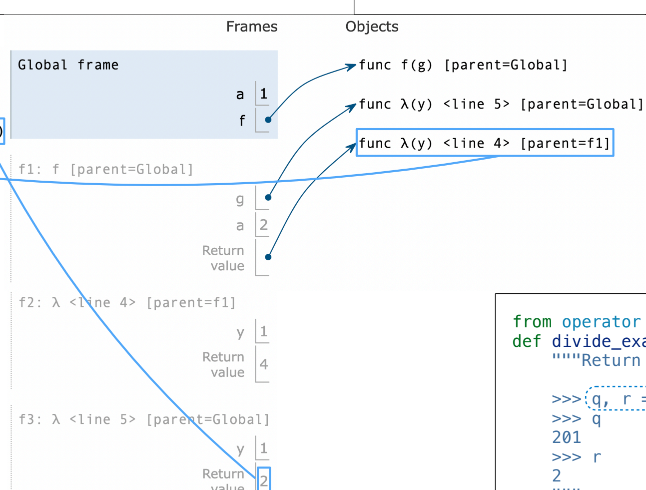
Return value of make_adder is an argument to compose1




```
from math import sqrt

def isPrime(n):
    i = 2
    while i <= int(sqrt(n)):
        if n % i == 0:
            return False
        i = i + 1
    return True
```

```
def search(f):
    """Return the smallest non-negative
    integer x for which f(x) is a true value.
    """
    x = 0
    while True:
        if f(x):
            return x
        x += 1

def is_three(x):
    """Return whether x is three.

    >>> search(is_three)
    3
    """
    return x == 3

def inverse(f):
    """Return a function g(y) that returns
    x such that f(x) == y.

    >>> sqrt = inverse(lambda x: x * x)
    >>> sqrt(16)
    4
    """
    return lambda y: search(lambda x: f(x)==y)
```

```
a = 1
def f(g):
    a = 2
    return lambda y: a * g(y)
f(lambda y: a + y)(a)
```



False values so far: **0, False, '', None**

Anything value that's not false is true.

```
>>> if 0:
...       print('*')
>>> if 1:
...       print('*')
*
>>> if abs:
...       print('*')
*
```

```
>>> if 1 and 0:
...       print('*')
>>> if 1 or 0:
...       print('*')
*
>>> if 1 or 1/0:
...       print('*')
*
```

```
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

Two return values, separated by commas

CS 61A Midterm 2 Study Guide – Page 1

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

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

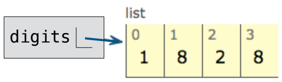
```
>>> today = datetime.date(2019, 10, 13)
>>> repr(today) # or today.__repr__()
'datetime.date(2019, 10, 13)'
>>> str(today) # or today.__str__()
'2019-10-13'
```

The result of evaluating an f-string literal contains the str string of the value of each sub-expression.

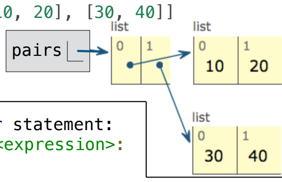
```
>>> f'pi starts with {pi}...'
'pi starts with 3.141592653589793...'
>>> print(f'pi starts with {pi}...')
pi starts with 3.141592653589793...
```

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 list, tuple, iterator, etc.)
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

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

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

Dictionaries:

```
words = {
      "más": "more",
      "otro": "other",
      "agua": "water"
}
```

```
>>> len(words)
3
>>> "agua" in words
True
>>> words["otro"]
'other'
>>> words["pavo"]
KeyError
>>> words.get("pavo", "🤔")
'🤔'
```

Dictionary comprehensions:

{key: value for <name> in <iter exp>}

```
>>> {x: x*x for x in range(3,6)}
{3: 9, 4: 16, 5: 25}
```

```
>>> [word for word in words]
['más', 'otro', 'agua']
>>> [words[word] for word in words]
['more', 'other', 'water']
>>> words["oruguita"] = 'caterpillar'
>>> words["oruguita"]
'caterpillar'
>>> words["oruguita"] += '🐛'
>>> words["oruguita"]
'caterpillar🐛'
```

Functions that aggregate iterable arguments

- **sum**(iterable[, start]) -> value — *sum of all values*
- **max**(iterable[, key=func]) -> value — *largest value*
  max(a, b, c, ...[, key=func]) -> value
  min(iterable[, key=func]) -> value — *smallest value*
  min(a, b, c, ...[, key=func]) -> value
- **all**(iterable) -> bool — *whether all are true*
  any(iterable) -> bool — *whether any is true*

Many built-in Python sequence operations return iterators that compute results lazily

map(func, iterable):
  Iterate over func(x) for x in iterable
filter(func, iterable):
  Iterate over x in iterable if func(x)
zip(first_iter, second_iter):
  Iterate over co-indexed (x, y) pairs
reversed(sequence):
  Iterate over x in a sequence in reverse order

To view the contents of an iterator, place the resulting elements into a container

list(iterable):
  Create a list containing all x in iterable
tuple(iterable):
  Create a tuple containing all x in iterable
sorted(iterable):
  Create a sorted list containing x in iterable

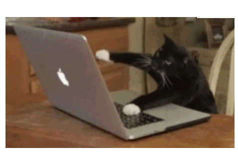
```
def cascade(n):
    if n < 10:
        print(n)
    else:
        print(n)
        cascade(n//10)
        print(n)
```

```
>>> cascade(123)
123
12
1
12
123
```

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

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

λ λ λ



List 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
```

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

```
>>> a = [10, 20, 30]
>>> list(a)
[10, 20, 30]
>>> a[:]
[10, 20, 30]
```

Tuples:

```
>>> empty = ()
>>> len(empty)
0
>>> conditions = ('rain', 'shine')
>>> conditions[0]
'rain'
>>> conditions[0] = 'fog'
Error
```

```
>>> all([False, True])
False
>>> all([])
True
>>> sum([1, 2])
3
>>> sum([1, 2], 3)
6
>>> sum([])
0
>>> sum([[1], [2]], [])
[1, 2]
```

```
>>> any([False, True])
True
>>> any([])
False
>>> max(1, 2)
2
>>> max([1, 2])
2
>>> max([1, -2], key=abs)
-2
```

List methods:

```
>>> 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

Removes first matching value

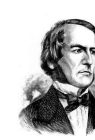
Add all values

Replace a slice with values

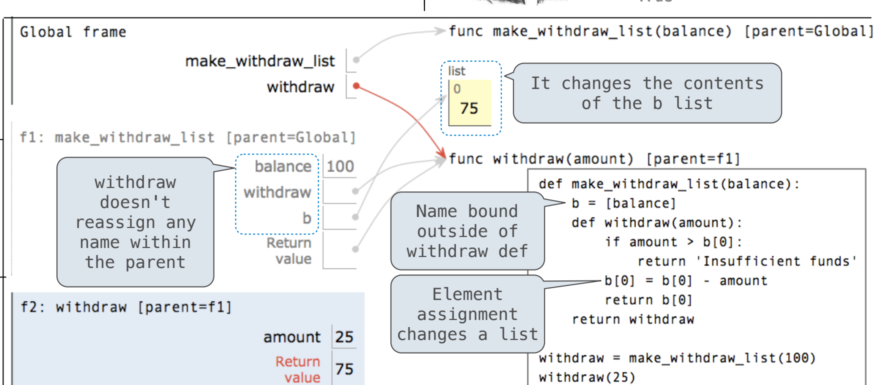
Add an element at an index

False values:
- Zero
- False
- None
- An empty string, list, dict, tuple

All other values are true values.



```
>>> bool(0)
False
>>> bool(1)
True
>>> bool('')
False
>>> bool('0')
True
>>> bool([])
False
>>> bool([[]])
True
>>> bool({})
False
>>> bool(())
False
>>> bool(lambda x: 0)
True
```



iter(iterable):
  Return an iterator over the elements of an iterable value
next(iterator):
  Return the next element of an iterator

```
>>> s = [3, 4, 5]
>>> t = iter(s)
>>> next(t)
3
>>> next(t)
4
```

```
>>> d = {'one': 1, 'two': 2}
>>> k = iter(d)
>>> next(k)
'one'
>>> next(k)
'two'
```

```
>>> v = iter(d.values())
>>> next(v)
1
>>> next(v)
2
```

A generator function is a function that yields values instead of returning.

```
>>> def plus_minus(x):
...     yield x
...     yield -x
```

```
>>> t = plus_minus(3)
>>> next(t)
3
>>> next(t)
-3
```

```
def a_then_b(a, b):
    yield from a
    yield from b
```

```
>>> list(a_then_b([3, 4], [5, 6]))
[3, 4, 5, 6]
```

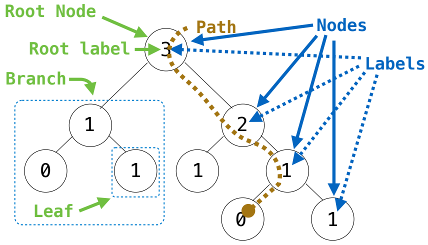
CS 61A Midterm 2 Study Guide — Page 2

Recursive description:

- A **tree** has a root **label** and a list of **branches**
- Each branch is a **tree**
- A tree with zero branches is called a **leaf**

Relative description:

- Each location is a **node**
- Each **node** has a **label**
- One node can be the **parent/child** of another

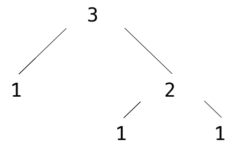



```
def tree(label, branches=[]):
    for branch in branches:
        assert is_tree(branch)
    return [label] + list(branches)
def label(tree):
    return tree[0]
def branches(tree):
    return tree[1:]
def is_tree(tree):
    if type(tree) != list or len(tree) < 1:
        return False
    for branch in branches(tree):
        if not is_tree(branch):
            return False
    return True
def is_leaf(tree):
    return not branches(tree)
```

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 leaves(t):
    """The leaf values in t.
    >>> leaves(fib_tree(5))
    [1, 0, 1, 0, 1, 1, 0, 1]
    """
    if is_leaf(t):
        return [label(t)]
    else:
        return sum([leaves(b) for b in branches(t)], [])
```

```
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 = label(left) + label(right)
        return tree(fib_n, [left, right])
```

Anatomy of a recursive function:

- The **def statement header** is like any function
- Conditional statements check for **base cases**
- Base cases are evaluated **without recursive calls**
- Recursive cases are evaluated **with recursive calls**

```
def sum_digits(n):
  "Sum 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
```

- **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
```

SCRATCH PAPER