CS 88 Spring 2019 Computational Structures in Data Science

FINAL

INSTRUCTIONS

- You have 3 hours to complete the exam. Put your name and SID on every page.
- The exam is closed book; no resources are allowed except two 8.5" × 11" cheat sheets and the official CS 88 final reference sheet (attached to the back of the exam). Remove the reference sheet before turning in exam.
- Mark your answers **on the exam itself**. We will *not* grade answers written on scratch paper. Check that you have 8 double-sided pages (including cover page) for 7 problems.

Last name	
First name	
Student ID number	
Berkeley email (`_@berkeley.edu`)	
TA	
Name of the person to your left	
Name of the person to your right	
All the work on this exam is my own. **(please sign)**	

POLICIES & CLARIFICATIONS

- You may use built-in Python functions that do not require import, such as `min`, `max`, `pow`, and `abs`. You may not use functions defined on your study guide unless clearly specified in the question.
- For fill-in-the blank coding problems, we will only grade work written in the provided blanks. You may only write one Python statement per blank line, and it must be indented to the level that the blank is indented. Your solution must fit within the number of lines provided, but may not require all of the lines.
- Unless otherwise specified, you are allowed to reference functions defined in previous parts of the same question.

[This page is purposely left blank. Use it as scratch space.]

1. Evaluators Gonna Evaluate

For each of the expressions in the table below, write the output displayed by the interactive Python interpreter when the expression is evaluated. The output may have multiple lines. **If an error occurs, write "Error". If a function is outputted, write "function".** Your answers must fit within the boxes provided. Work outside the boxes will not be graded.

Hint: No answer requires more than 6 lines. The first two rows have been provided as examples. Recall: The interactive interpreter displays the value of a successfully evaluated expression, unless it is None. Assume that you have started `python3` and executed the following statements:

```
def anGenerator():
    x = 0
    while True:
        yield x
        x += 1

class GenIterator:
    def __init__(self):
        self.current = anGenerator()

    def __next__(self):
        return next(self.current)

    def __iter__(self):
        return self

class Flower:
    petals = True

    def __init__(self, colour):
        self.colour = colour

    def color(self):
        print("I'm colorful!")
```

```
class Tulip(Flower):
    season = "spring"

    def color(self):
        print(self.colour)


class Daffodil(Flower):

    def __init__(self, colour):
        self.colour = colour
        self.height = 0

    def color(self):
        print(self.colour)

    def grow(self, inches):
        self.height += inches

    def season(self):
        print("Season pushed back")
```

Expression	Interactive Output
`Flower.petals`	`True`
`Rose()`	`Error`
`tulip = Tulip("red")` `tulip.color()`	

`daffodil = Daffodil("yellow")` `daffodil.color()`	
`Flower.color(daffodil)`	
`daffodil.petals`	
`tulip.season = "early spring"` `print(Tulip.season, tulip.season)`	
`tule = Tulip("purple")` `tule.season`	

`tulip = Tulip("blue")`
`Tulip.color(daffodil)`
`tulip.color(daffodil)`	

`tulip.height = 100`
`Daffodil.grow(tulip, 200)`
`Tulip.height`	

`a = GenIterator()`
`for i in range(1, 6):`
`  print(next(a))`	

`for i in range(3):` `  print(next(a))`	
`next(GenIterator())`	

2. Some Tech Fame

Fill in the environment diagram that results from executing the code below until the entire program is finished, an error occurs, or all frames are filled. *You may not need to use all of the spaces or frames.*

There are 20 blanks total you need to fill out!

A complete answer will:

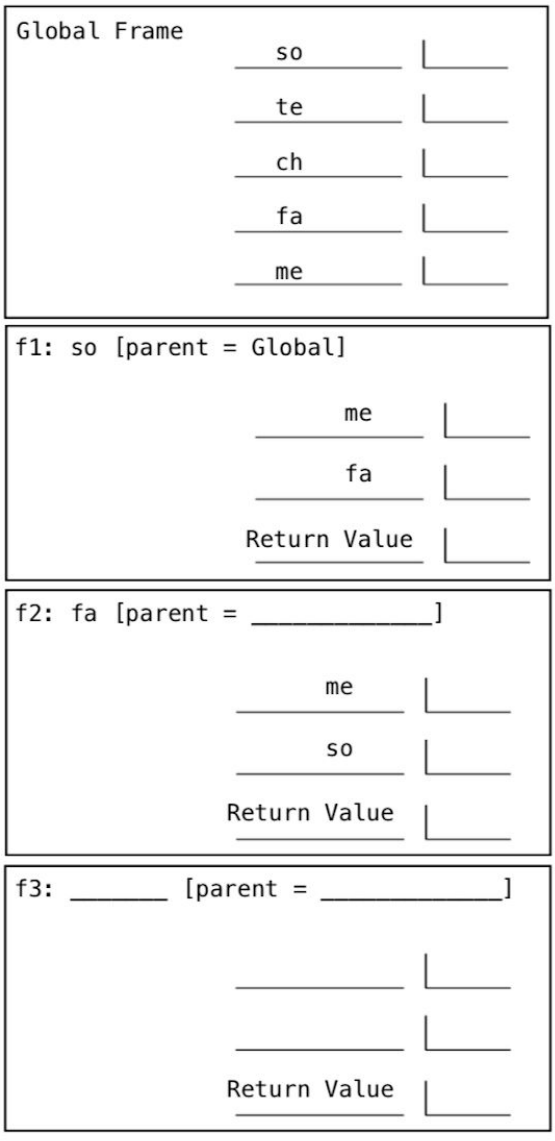
- Add all missing names and parent annotations to all local frames.
- Draw any necessary arrows to function names.
- Add all missing values created or referenced during execution.
- Show the return value for each local frame.

```
so = 5
te = 6
ch = [2, 4]

def so(me):
    me = 8
    def fa(me, so):
        so.append(me)
        return me + 1
    return fa

def fa(me, so):
    return [me] + so

te = so(te)(te, ch)
me = fa
me(['c', 'h'], ch)
```



func so(me) [parent = Global]

func fa(me, so) [parent = Global]

func fa(me, so) [parent = _____]

3. Warriors in 6

Answer the following SQL questions given tables **`Players`** and **`Stats`** of the following form:

```
Table: Players
name             | team           | college        | age
-------------------------------------------------------
DeMarcus Cousins | Golden State   | Kentucky       | 28
Kevin Durant     | Golden State   | Texas          | 30
James Harden     | Houston        | Arizona        | 29
Kawhi Leonard    | Toronto        | San Diego      | 27
Oski Bear        | Memphis        | California     | 22

Table: Stats
name             | minutes  | points   | rebounds | assists
-----------------------------------------------------------
DeMarcus Cousins | 0        | 0        | 0        | 0
Kevin Durant     | 28       | 35       | 5        | 3
James Harden     | 33       | 35       | 4        | 6
Kawhi Leonard    | 15       | 18       | 10       | 10
Oski Bear        | 24       | 101      | 39       | 31
```

A. What is the output of the following SQL query. Not all boxes will be necessary.

```
SELECT name, rebounds+assists, points FROM Stats WHERE points > minutes
ORDER BY points, name
```


B. Write a SQL query that retrieves the **name** of all players who had more rebounds than assists.

C. Write a SQL query that retrieves the **name** and their **points per minute** for all players who played at least 1 minute.

D. Write a SQL query that retrieves the **name**, **college**, and **points** of all players. Note: your query output should NOT repeat any rows.

E. Write a SQL query that retrieves all *unique pairs* of player **names** if the sum of the 2 players' points is greater than 60. Order the pair of names in each row by alphabetical order, and order the rows in alphabetical order by the first player in the pair. Here is the expected output:

DeMarcus Cousins	Oski Bear
James Harden	Kevin Durant
James Harden	Oski Bear
Kawhi Leonard	Oski Bear
Kevin Durant	Oski Bear

---

4. Find the Mayor

In a city of *N* people, represented by integers 1 to N, you are tasked in finding which person out of all of them is the mayor. Only one person can be mayor. You are given `pairs`, a list of 2-element lists in the form of [a,b]. Each pair [a, b] denotes that person a trusts person b.

The mayor has two important properties:

1. The mayor is trusted by all of the other people.
2. The mayor trusts no one.

Complete the main function and helper functions below to return the integer that represents the mayor, or -1 if the mayor does not exist. You can assume `pairs` is not an empty list and N > 1.

A. First, complete the `createTrusted` helper function.

```
def createTrusted(pairs):
    """ Returns a dictionary mapping a person to a list of people who
    trust them. The order of the list of people does not matter.

    >>> createTrusted([[1,3], [2,3], [3,1]])
    {3: [1, 2], 1: [3]}
    >>> createTrusted([[1,3], [1,4], [2,3], [2,4], [4,3]])
    {3: [1, 2, 4], 4: [1, 2]}
    """

    trusted = {}

    ____________________________________________________________

    ____________________________________________________________

    ____________________________________________________________

    ____________________________________________________________

    ____________________________________________________________

    ____________________________________________________________

    return trusted
```

B. Next, complete the `createTrusts` helper function.

```
def createTrusts(pairs):
    """ Returns a dictionary mapping a person to a list of people they
    trust. The order of the list of people does not matter.

    >>> createTrusts([[1,3], [2,3], [3,1]])
    {1: [3], 2: [3], 3: [1]}
    >>> createTrusts([[1,3], [1,4], [2,3], [2,4], [4,3]])
    {1: [3, 4], 2: [3, 4], 4: [3]}
    """

    trusts = {}

    ____________________________________________________________

    ____________________________________________________________

    ____________________________________________________________

    ____________________________________________________________

    ____________________________________________________________

    ____________________________________________________________

    return trusts
```

C. Finally, complete the findMayor function to solve our original problem. You may use `createTrusted` and `createTrusts` from above and can assume they work properly.

```
def findMayor(N, pairs):
    """ Return the integer representing the mayor with the properties:
    1. The mayor is trusted by all of the other people.
    2. The mayor trusts no one.
    Return -1 if no such mayor exists.

    >>> findMayor(2, [[1,2]])
    2 # 1 trusts 2, 2 doesn't trust anyone, so 2 is the mayor
    >>> findMayor(3, [[1,3], [2,3]])
    3 # everyone trusts 3, but 3 trusts no one, so 3 is mayor
    >>> findMayor(3, [[1,3], [2,3], [3,1]])
    -1 # everyone trusts 3, but 3 trusts 1, so not mayor
    >>> findMayor(3, [[1,2], [2,3]])
    -1 # No one is trusted by everyone, so no mayor
    >>> findMayor(4, [[1,3], [1,4], [2,3], [2,4], [4,3]])
    3 # everyone trusts 3, but 3 trusts no one, so 3 is mayor
    """

    trusted = ____________________________________________________

    trusts = _____________________________________________________

    ______________________________________________________________

    ______________________________________________________________

    ______________________________________________________________

    ______________________________________________________________

    ______________________________________________________________

    ______________________________________________________________

    ______________________________________________________________

    ______________________________________________________________

    ______________________________________________________________

    return -1
```

5. Perfect Numbers

A perfect number is a positive integer that is equal to the sum of its proper positive divisors, that is, the sum of its positive divisors excluding the number itself.

A. First, write a function that returns the list of all proper divisors of a number **n**. A proper divisor of **n** is a positive integer that evenly divides **n** and is not equal to **n**. Assume **n** is a positive integer and we only want divisors that are also positive integers.

Definition: x is a divisor of n if n % x == 0
Definition: x is a proper divisor of n if x is a divisor of n and x != n

```
def get_proper_divisors(n):
    """
    >>> get_proper_divisors(1)
    [] # 1 is the only divisor of 1, but is not a proper divisor
    >>> get_proper_divisors(2)
    [1] # 1 and 2 are divisors of 2, but 1 is the only proper divisor
    >>> get_proper_divisors(3)
    [1]
    >>> get_proper_divisors(4)
    [1, 2]
    >>> get_proper_divisors(5)
    [1]
    >>> get_proper_divisors(6)
    [1, 2, 3]
    """

    ------------------------------------------------------------------

    ------------------------------------------------------------------

    ------------------------------------------------------------------

    ------------------------------------------------------------------

    ------------------------------------------------------------------

    ------------------------------------------------------------------

    ------------------------------------------------------------------
```

B. Write a generator function perfect_nums() that continually yields successive perfect numbers. Perfect numbers are positive numbers that are equal to the sum of their proper divisors. You can assume that get_proper_divisors() is implemented correctly and may use it in this problem.

```
def perfect_nums():
    """Generate each successive perfect number.
    >>> perfect_num_gen = perfect_nums()
    >>> next(perfect_num_gen)
    6 # 6 is the first perfect number because its proper divisors are 1,
        2, 3 which sum to itself
    >>> next(perfect_num_gen)
    28 # 28 is the second perfect number because its proper divisors are
         1, 2, 4, 7, 14 which sum to itself
    """

    ------------------------------------------------------------------

    ------------------------------------------------------------------

    ------------------------------------------------------------------

    ------------------------------------------------------------------

    ------------------------------------------------------------------

    ------------------------------------------------------------------

    ------------------------------------------------------------------

    ------------------------------------------------------------------

    ------------------------------------------------------------------
```

6. Time Is Money

Fill in the __next__ method in Timer and the pass_time method in KitchenCounter. A timer should step forward one second each time next is called. Once the timer runs out, you should print out a message that says the food is ready. KitchenCounter maintains a list of timers; pass_time should step forward all of the timers by the amount of seconds specified by the time and unit arguments. The timers should always be within one second of each other (i.e. increment all of the timers once before incrementing any timer twice.) **TIP: Don't forget about StopIteration Error.**

```
class Timer:
    """
    >>> a = Timer("Pete Zaroll", 2, "seconds")
    >>> b = [i for i in a]
    Pete Zaroll is ready!
    """
    # Maps a unit string to a multiplier that converts it to seconds
    unit2Seconds = {"seconds" : 1, "minutes" : 60, "hours" : 60*60}
    def __init__(self, food, time, unit):
        self.food = food
        self.current = 1
        self.time = time * self.unit2Seconds[unit]

    def __iter__(self):
        return self

    def ready(self):
         print(self.food + " is ready!")

    def __next__(self):

        ____________________________________________________________

        ____________________________________________________________

        ____________________________________________________________

        ____________________________________________________________

        ____________________________________________________________

        ____________________________________________________________

        ____________________________________________________________

        ____________________________________________________________
```

```
class KitchenCounter:
    """
    >>> a = Timer("Pete Zaroll", 15, "minutes")
    >>> b = Timer("Chim E Changa", 20.5, "minutes")
    >>> c = Timer("Pho Lah Phil", 12, "seconds")
    >>> k = KitchenCounter()
    >>> k.add_timers([a, b, c])
    >>> k.pass_time(12, "seconds")
    Pho Lah Phil is ready!
    12 seconds passed
    >>> k.pass_time(15, "minutes")
    Pete Zaroll is ready!
    15 minutes passed
    >>> k.pass_time(5.5, "minutes")
    Chim E Changa is ready!
    5.5 minutes passed
    """
    unit2Seconds = {"seconds" : 1, "minutes" : 60, "hours" : 60*60}
    def __init__(self):
        self.timers = []

    def add_timers(self, timers):
        self.timers += timers

    def pass_time(self, time, units):
        """Increment each timer in self.timers by the appropriate amount of
        seconds. Remove any timer from the list of timers once its time has run
        out. Hint: lists have a remove method. Hint: StopIteration
        """

        seconds = int(self.unit2Seconds[units]*time)

        ____________________________________________________________

        ____________________________________________________________

        ____________________________________________________________

        ____________________________________________________________

        ____________________________________________________________

        ____________________________________________________________

        ____________________________________________________________

        ____________________________________________________________

        print(str(time) + " " + str(units) + " passed")
```

7. Class Is in Session

Implement the 3 classes to match the interactive outputs below:

```
$ python3
>>> andrew = Person("Andrew")
>>> andrew.say()
Hi I'm Andrew
>>> alex = TA("Alex")
>>> amir = Student("Amir", alex)
>>> amir.say()
Hi I'm Amir and I'm in Alex's lab
>>> alex.add_student(amir)
>>> alex.add_student(Student("Jessica", alex))
>>> alex.say()
Hi I'm Alex and my students are Amir Jessica
>>> alex.add_student(Student(“Gerald”, alex))
>>> alex.say()
Hi I'm Alex and my students are Amir Jessica Gerald
```

```
class Person:

    def __init__(self, name):
        self.name = name

    def say(self):
        print("Hi I'm " + self.name)


class ____________________________________________________________:

    def __init__(self, name, ta):

        super.__init__(____________, ____________)

        ____________________________________________________________

        ____________________________________________________________


    def say(self):

        ____________________________________________________________

        print(____________________________________________________)
```

```
class ____________________________________________________________:

    def __init__(_____________, _____________):

        ____________________________________________________________

        ____________________________________________________________


    def add_student(self, student):

        ____________________________________________________________

        ____________________________________________________________

        ____________________________________________________________


    def say(self):

        ____________________________________________________________

        ____________________________________________________________

        ____________________________________________________________

        ____________________________________________________________

        ____________________________________________________________

        print(_____________________________________________________)
```

Numeric types in Python:

```
>>> type(2)
<class 'int'>
>>> type(1.5)
<class 'float'>
>>> type(1+1j)
<class 'complex'>
```

Represents integers exactly

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

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

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

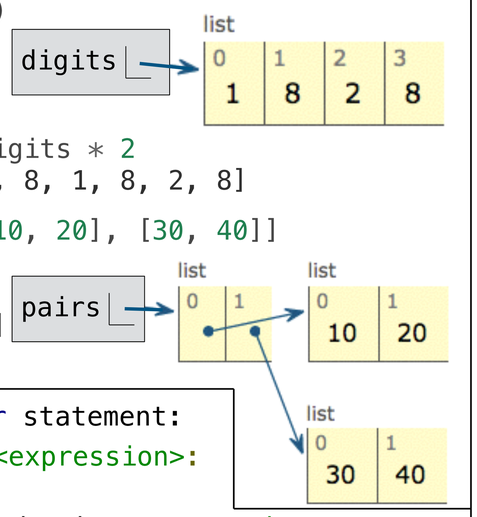
This function represents a rational number

Constructor is a higher-order function

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:

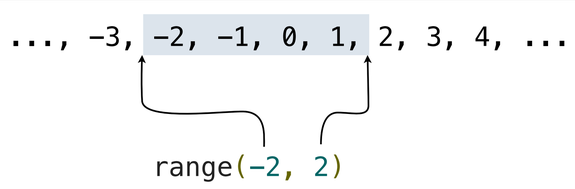
A sequence of fixed-length sequences

```
>>> 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
```



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

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'
```

```
>>> 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

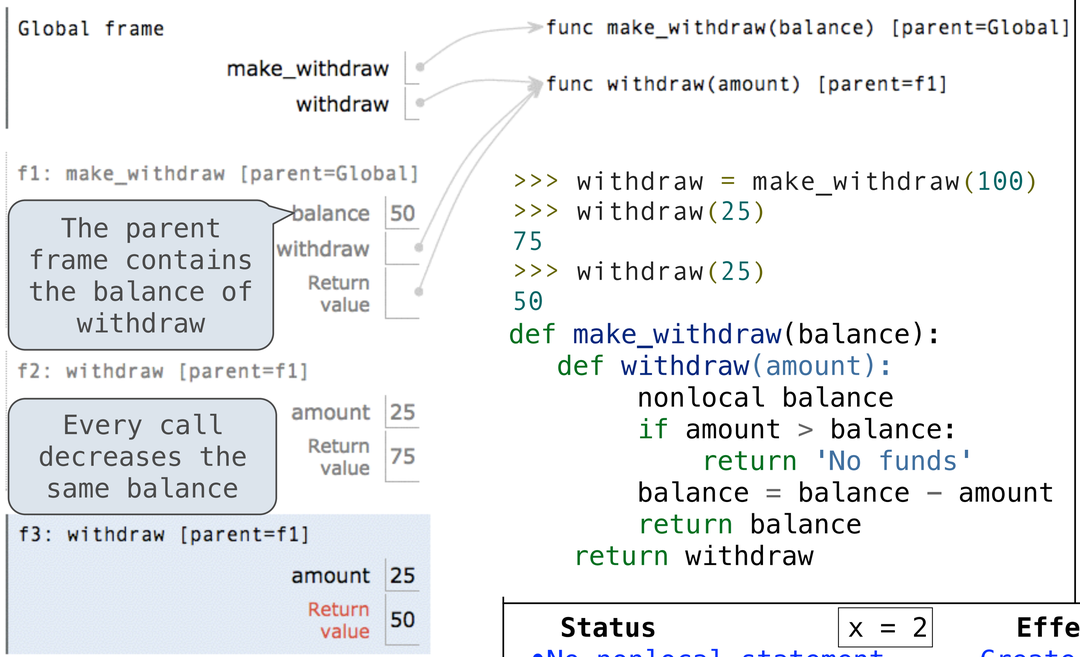
Add all values

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.




```
>>> 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
```

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}
```

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
```

Membership:

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

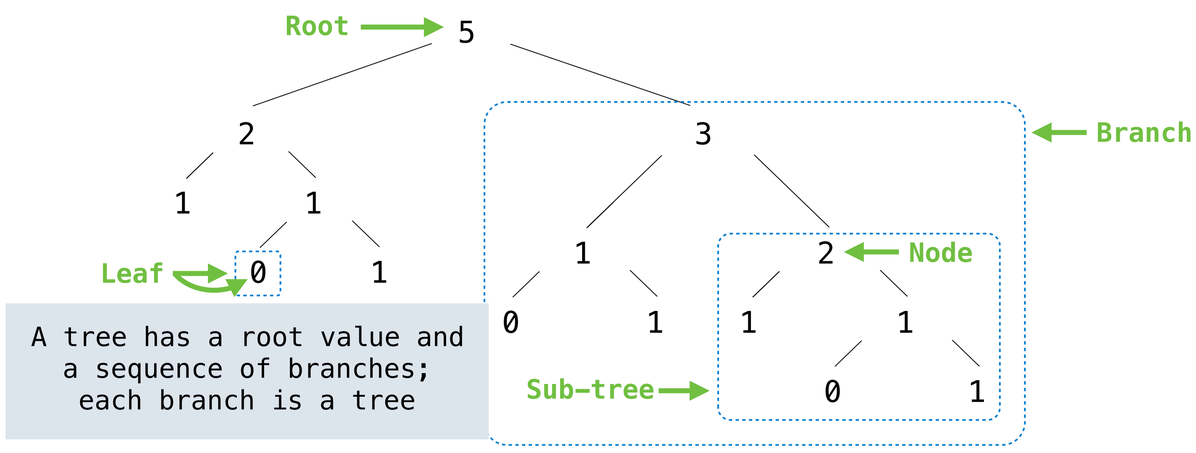
Slicing:

```
>>> digits[0:2]
[1, 8]
>>> digits[1:]
[8, 2, 8]
```

Slicing creates a new object

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:



A tree has a root value and a sequence of branches; each branch is a tree

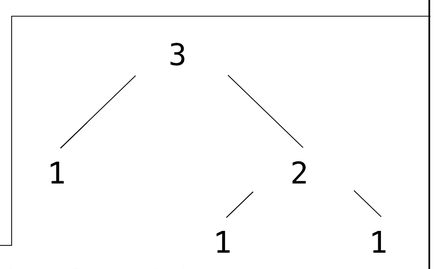
```
def tree(root, branches=[]):
    for branch in branches:
        assert is_tree(branch)
    return [root] + list(branches)
def root(tree):
    return tree[0]
def branches(tree):
    return tree[1:]
def is_tree(tree):
    if type(tree) != list or len(tree) < 1:
        return False
    for branch 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
def leaves(tree):
    if tree.is_leaf():
        return [tree.entry]
    else:
        return sum([leaves(b) for b in tree.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])
```

```
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)
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))
```

Some zero length sequence

Sequence abstraction special names:

`__getitem__`	Element selection []
`__len__`	Built-in len function

Yes, this call is recursive

Contents of the repr string of a Link instance

```
>>> 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

```
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
```

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

Function call: all arguments within parentheses

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

```
>>> type(Account.deposit)
<class 'function'>
>>> type(a.deposit)
<class 'method'>
>>> Account.deposit(a, 5)
10
>>> a.deposit(2)
12
```

Dot expression

Call 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
```