INSTRUCTIONS

- You have 2 hours to complete the exam.
- The exam is closed book, closed notes, closed computer, closed calculator, except one hand-written 8.5" × 11" crib sheet of your own creation and the two official 61A midterm study guides attached to the back of this exam.
- Mark your answers ON THE EXAM ITSELF. If you are not sure of your answer you may wish to provide a brief explanation.

<table>
<thead>
<tr>
<th>Last name</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>First name</td>
<td></td>
</tr>
<tr>
<td>SID</td>
<td></td>
</tr>
<tr>
<td>Login</td>
<td></td>
</tr>
<tr>
<td>TA &amp; section time</td>
<td></td>
</tr>
<tr>
<td>Name of the person to your left</td>
<td></td>
</tr>
<tr>
<td>Name of the person to your right</td>
<td></td>
</tr>
</tbody>
</table>

All the work on this exam is my own. (please sign)

For staff use only

<table>
<thead>
<tr>
<th></th>
<th>Q. 1</th>
<th>Q. 2</th>
<th>Q. 3</th>
<th>Q. 4</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>/16</td>
<td>/12</td>
<td>/14</td>
<td>/8</td>
<td>/50</td>
</tr>
</tbody>
</table>
1. (16 points) Expressionism

(a) (8 pt) For each of the following expressions, write the \texttt{repr} string of the value to which the expression evaluates. Special cases: If an expression evaluates to a function, write \texttt{FUNCTION}. If evaluation would never complete, write \texttt{FOREVER}. None of these expressions cause an error.

Assume that the expressions are evaluated in order. Evaluating the first may affect the value of the second, etc.

Assume that you have started Python 3 and executed the following statements:

```python
def fruit(y):
    def ninja(angry):
        nonlocal y
        if y < len(angry):
            return angry
        y = y - 2
        return pig(angry)
    def pig(bird):
        bird.append(y)
        return ninja(bird)
    return pig

slingshot = fruit(5)
green = lambda x: fruit(x)([x])
```

<table>
<thead>
<tr>
<th>Expression</th>
<th>Evaluates to</th>
</tr>
</thead>
<tbody>
<tr>
<td>5*5</td>
<td>25</td>
</tr>
<tr>
<td>slingshot([1, 2, 3, 4])</td>
<td></td>
</tr>
<tr>
<td>green(5)</td>
<td></td>
</tr>
<tr>
<td>slingshot([6, 7])</td>
<td></td>
</tr>
<tr>
<td>green(8)</td>
<td></td>
</tr>
</tbody>
</table>
(b) (8 pt) For each of the following expressions, write the `repr` string of the value to which the expression evaluates. Special cases: If an expression evaluates to a function, write `FUNCTION`. If evaluation would never complete, write `FOREVER`. None of these expressions cause an error.

Assume that the expressions are evaluated in order. Evaluating the first may affect the value of the second, etc.

Assume that you have started Python 3 and executed the following statements:

```python
class Student(object):
    def __init__(self, s):
        if len(s) < 2:
            self.s = s
        else:
            self.s = Student(s[1:])

    def __repr__(self):
        return 'Student(' + repr(self.s) + ')

    def learn(self):
        if hasattr(self, 'teach'):
            return self.teach()
        while type(self.s) == Student:
            self.s = self.s.s
        return self.s

class Teacher(Student):
    def teach(self):
        return 'Good Job'

luke = Student([1, 3])
yoda = Teacher([5, luke])
```

<table>
<thead>
<tr>
<th>Expression</th>
<th>Evaluates to</th>
</tr>
</thead>
<tbody>
<tr>
<td>5*5</td>
<td>25</td>
</tr>
<tr>
<td>luke.learn()</td>
<td></td>
</tr>
<tr>
<td>luke</td>
<td></td>
</tr>
<tr>
<td>yoda</td>
<td></td>
</tr>
<tr>
<td>Student.learn(yoda)</td>
<td></td>
</tr>
</tbody>
</table>
2. (12 points) Picture Frame

(a) (5 pt) 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.
A complete answer will:
• Add all missing names, labels, and parent annotations to all local frames.
• Add all missing values created during execution.
• Show the return value for each local frame.

```
def make(up):
    exam = [2012]
    def wed():
        nonlocal exam
        if exam[0] >= up:
            return list(exam)
        a = exam[0] + 1
        exam = [a, exam]
        return wed()
    return wed()
make(2014)
```

(b) (1 pt) Circle True or False: This diagram would change if the nonlocal statement were removed.
(c) (5 pt) 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.

A complete answer will:

- Add all missing names, labels, and parent annotations to all local frames.
- Add all missing values created during execution.
- Show the return value for each local frame.

```python
def beep(oo):
    b[oo] = [ee, oo, b[ee]]
    return b[len(b)-1]

b = list(range(3, 7))
ee = 1
beep(0).append('not found')
```

(d) (1 pt) What will `print(b)` output after executing this code?
3. (14 points) Objets d'Art

(a) (6 pt) Cross out whole lines in the implementation below so that the doctests for \texttt{Vehicle} pass. In addition, cross out all lines that \textbf{have no effect}. Don’t cross out docstrings, doctests, or decorators.

```python
class Vehicle(object):
    """
    >>> c = Car('John', 'CS61A')
    >>> c.drive('John')
    John is driving
    >>> c.drive('Jack')
    Car stolen: John CS61A
    >>> c.pop_tire()
    3
    >>> c.pop_tire()
    2
    >>> c.fix()
    >>> c.pop_tire()
    """
    def __init__(self, owner):
        self.owner = owner
    def move(self):
        print(self.owner + ' is driving')

class Car(Vehicle):
    tires = 4
    Car.tires = 4
    def __init__(self, owner, license_plate):
        Vehicle.__init__(owner)
        Vehicle.__init__(self, owner)
        self.plate = license_plate
        self.tires = tires
        self.tires = Car.tires
    def drive(self, person):
        if person != self.owner:
            if self.person != self.owner:
                print('Car stolen: ' + self.identification)
                print('Car stolen: ' + self.identification())
                print('Car stolen: ' + self.identification())
            else:
                Car.move(self)
        @property
        def identification(self):
            return self.owner + ' ' + self.plate
    def pop_tire(self):
        self.tires -= 1
        return self.tires
    def fix(self):
        setattr(Car, 'tires', self.tires)
        setattr(self, 'tires', type(self).tires)
        setattr(self, 'tires', self.Car.tires)
```
(b) (4 pt) The \texttt{max_path} function takes an instance of the \texttt{Tree} class from Study Guide 2. It is meant to return the maximal sum of internal \texttt{entry} values on a path from the root to a leaf of the tree.

```python
def max_path(tree):
    """Return the sum of entries in a maximal path from the root to a leaf."

    paths = [0]
    if tree.right is not None:
        paths.append(max_path(tree.right))
    if tree.left is not None:
        paths.append(max_path(tree.left))
    if min(paths) < 0:
        return tree.entry + max([p for p in paths if p < 0])
    return tree.entry + max(paths)
```

Circle \textbf{True} or \textbf{False} to indicate whether each of the following statements about \texttt{max_path} is true.

i. (\textbf{True}/\textbf{False}) It returns the correct result for all doctests shown.

ii. (\textbf{True}/\textbf{False}) It returns the correct result for all valid trees with integer entries.

(c) (4 pt) Define a function \texttt{coerce} that takes a \texttt{Tree} instance and returns a dispatch dictionary representing the same tree using messages \texttt{left}, \texttt{right}, and \texttt{entry}. The base case has been provided for you.

```python
def coerce(tree):
    """Return a dispatch dictionary representing the tree."

    if tree is None:
        return None
```

```python
>>> t = Tree(4, None, Tree(5, None, Tree(1, Tree(3, Tree(5), Tree(2)))))
>>> d = coerce(t)
>>> d['entry']
4
>>> d['left'] is None
True
>>> d['right']['right']['left']['entry']
3
```

```python
if tree is None:
    return None
```
4. (8 points) Form and Function

(a) (4 pt) You have been hired to work on AI at UnitedPusherElectric, the leading manufacturer of Pusher Bots. The latest model, PusherBot 5, keeps pushing people down stairs when it gets lost. Fix it!
Assume that you have an abstract data type `position` that combines x and y coordinates (in meters).

```python
>>> pos = position(3, 4)
>>> x(pos)
3
>>> y(pos)
4

pathfinder should return a visit function that takes a position argument. visit returns True unless:
  i. Its argument position is more than 6 meters from position(0, 0), or
  ii. Its argument position has been visited before.

The implementation below is incorrect. Cross out each line (or part of a line) that must change and write a revised version next to it, so that pathfinder is correct and does not depend on the implementation of position. Assume your corrections have the same indentation as the lines they replace. You may not add or remove lines. Make as few changes as necessary.

```python
from math import sqrt
def equal(position, other):
    return x(position) == x(other) and y(position) == y(other)

def pathfinder():
    """Return a visit function to help with path-finding."
    >>> visit1, visit2 = pathfinder(), pathfinder()
    >>> visit1(position(3, 4))
    True
    >>> visit1(position(5, 12)) # Too far away
    False
    >>> visit1(position(3, 4)) # Already visited
    False
    >>> visit2(position(3, 4))
    True
    """
    visited = ()

def visit(pos):
    if sqrt( x(pos)*x(pos) + y(pos)*y(pos) ) > 6:
        return False
    for p in visit:
        if p == pos:
            return True
    visited.append(pos)
    return True

return visited
(b) (4 pt) Fill in missing expressions in the implementation for subwords, which lists all of the n letter subwords of a given word w. A subword consists of some subset of the letters in a word, in their original order. You may assume that the word has no repeated letters. Some hints about string slicing appear in the doctest.

def subwords(w, n):
    """List all subwords of word w that have length n.
   "
    if n == 0:
        return ______________________________________________________
    if w == '':
        return ______________________________________________________
    r = [w[0] + x for x in ______________________________________________________]
    return r + ______________________________________________________

(c) (0 pt) Draw a picture of PusherBot 5.
### 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.
- Frames (right):
  - A frame extends the environment that begins with its parent frame.
  - The global environment: the environment with only the global frame.
  - When a frame or function has no label (parent=None), then its parent is always the global frame.

### 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:
1. Each clause is considered in order.
2. If it is a true value, execute the suite, then skip the remaining clauses in the statement.

### Execution rule for or expressions:
1. Evaluate the subexpression on the left.
2. If the result is a true value, then the expression evaluates to True.
3. Otherwise, the expression evaluates to the value of the subexpression on the right.

### Evaluation rule for not expressions:
1. Evaluate the operand subexpression.
2. If the result is a false value, then the expression evaluates to True.
3. Otherwise, the expression evaluates to the value of the operand subexpression.

### 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.

### Higher-order function:
- A function that takes a function value as an argument (function argument):
  - A function that takes a function value as an argument:
    - def add_three:
      - The cube function is passed as an argument value.
    - A formal parameter that will be bound to a function:
      - Function of a single argument (not called term):
A function
with formal parameters x and y and body "return x * y"
Must be a single expression

square = lambda x, y: x * y

def square(x, y):
    return x * y

What does sum_squares need to know about square?
Yes
Square takes one argument. Yes
Square has the intrinsic name square. Yes
Square computes the square of a number. Yes
Square computes the square by calling mul. No

The name add_three is bound to a function

def add_three(k):
    return k + n

The name add_three is identical to triple(k):

def triple(k):
    return 3 * k

def rest(s):
    return s[1:]

def first(s):
    return s[0]

def make_adder(k):
    return adder(k)

def adder(k):
    return k + n

def empty_rlist:
    return

def list_from(first, rest):
    if rest:
        return (first, rest)
    else:
        return first

def length(s):
    if empty_rlist(s):
        return 0
    else:
        return length(rest(s)) + 1

def rest(s):
    return s[1:]

def first(s):
    return s[0]

def make_adder(k):
    return adder(k)

def adder(k):
    return k + n

def triple(k):
    return 3 * k

def rest(s):
    return s[1:]

def first(s):
    return s[0]

square = lambda x: x * x

def square(x):
    return x * x

1. Both create a function with the same arguments & behavior
2. Both of those functions are associated with the environment in which they are defined
3. Both bind that function to the name "square"
4. Only the def statement gives the function an intrinsic name

How to find the square root of 2?
1. Compute the value of f at the guess: f(x)
2. Compute the derivative of f at the guess: f'(x)
3. Update guess to be: x - f(x)

def newton_update(f):
    return update(f)

def update(guess):
    return guess

def approx_derivative(f, x):
    return df(x)

def df(f, x):
    return f(x) - f(x - h)

def find_root(f, guess):
    return guess

def find_zero(f, guess):
    return guess

def golden_update(guess, x):
    return guess

def iter_improve(golden_update, golden_test):
    return update(golden_update)

def golden_test(guess, x):
    return guess

def divide_exact(n, d):
    return n / d

class Rational:
    Constructor
    Selectors

def rational(numer, denominator):
    return Rational(numer, denominator)

def add_rational(numer1, denominator1, numer2, denominator2):
    return Rational(numer1 * denominator2 + numer2 * denominator1, denominator1 * denominator2)

def eq_rational(numer1, denominator1, numer2, denominator2):
    return numer1 * denominator2 == numer2 * denominator1

def numerator(rational):
    return rational.numer

def denominator(rational):
    return rational.denominator

def simplify(rational):
    return Rational(numer, denominator)

def rationalize(rational):
    return Rational(numer, denominator)

def is_rational(n, d):
    return numerator(n) == 0 and denominator(d) == 1

def to_float(rational):
    return float(numerator) / denominator

def to_int(rational):
    return int(numerator) / denominator

def rationalize(n, d):
    return Rational(numer, denominator)

from operator import add

def add(a, b):
    return a + b

from operator import sub

def sub(a, b):
    return a - b

from operator import mul

def mul(a, b):
    return a * b

from operator import truediv

def truediv(a, b):
    return a / b

from operator import floordiv

def floordiv(a, b):
    return a // b

from operator import mod

def mod(a, b):
    return a % b

from operator import pow

def pow(a, b):
    return a ** b

from operator import pos

def pos(a):
    return a

from operator import neg

def neg(a):
    return -a

from operator import abs

def abs(a):
    return a

from operator import invert

def invert(a):
    return ~a

from operator import bitand

def bitand(a, b):
    return a & b

from operator import bitor

def bitor(a, b):
    return a | b

from operator import bitxor

def bitxor(a, b):
    return a ^ b

from operator import lshift

def lshift(a, b):
    return a << b

from operator import rshift

def rshift(a, b):
    return a >> b

from operator import pos

def pos(n):
    return n

from operator import neg

def neg(n):
    return -n

from operator import bool

def bool(n):
    return n

from operator import not_  

def not_(n):
    return not n

from operator import xor

def xor(n):
    return n

from operator import or_

def or_(n):
    return n

from operator import and_

def and_(n):
    return n

from operator import eq

def eq(n, m):
    return n == m

from operator import ne

def ne(n, m):
    return n != m

from operator import le

def le(n, m):
    return n <= m

from operator import lt

def lt(n, m):
    return n < m

from operator import ge

def ge(n, m):
    return n >= m

from operator import gt

def gt(n, m):
    return n > m

from operator import is_

def is_(n, m):
    return n is m

from operator import is_not

def is_not(n, m):
    return n is not m

from operator import instance

def instance(n, m):
    return n is instance of m

from operator import instance_not

def instance_not(n, m):
    return n is not instance of m

from operator import issubclass

def issubclass(n, m):
    return n issubclass of m

from operator import issubclass_not

def issubclass_not(n, m):
    return n is not subclass of m

from operator import true

def true(n):
    return True

from operator import false

def false(n):
    return False

from operator import truth

def truth(n):
    return bool(n)

from operator import falsity

def falsity(n):
    return not bool(n)

from operator import lnot

def lnot(n):
    return not n

from operator import bool

def bool(n):
    return n

from operator import not_  

def not_(n):
    return not n

from operator import xnor

def xnor(n):
    return n

from operator import iand

def iand(n):
    return n

from operator import ior

def ior(n):
    return n

from operator import ixor

def ixor(n):
    return n

from operator import ilshift

def ilshift(n):
    return n

from operator import irshift

def irshift(n):
    return n

from operator import int

def int(n):
    return n

from operator import float

def float(n):
    return n

from operator import str

def str(n):
    return n

from operator import bool

def bool(n):
    return n

from operator import not_  

def not_(n):
    return not n

from operator import xor

def xor(n):
    return n

from operator import or_

def or_(n):
    return n

from operator import and_

def and_(n):
    return n

from operator import eq

def eq(n, m):
    return n == m

from operator import ne

def ne(n, m):
    return n != m

from operator import le

def le(n, m):
    return n <= m

from operator import lt

def lt(n, m):
    return n < m

from operator import ge

def ge(n, m):
    return n >= m

from operator import gt

def gt(n, m):
    return n > m

from operator import is_

def is_(n, m):
    return n is m

from operator import is_not

def is_not(n, m):
    return n is not m

from operator import instance

def instance(n, m):
    return n is instance of m

from operator import instance_not

def instance_not(n, m):
    return n is not instance of m

from operator import issubclass

def issubclass(n, m):
    return n issubclass of m

from operator import issubclass_not

def issubclass_not(n, m):
    return n is not subclass of m

from operator import true

def true(n):
    return True

from operator import false

def false(n):
    return False

from operator import truth

def truth(n):
    return bool(n)

from operator import falsity

def falsity(n):
    return not bool(n)

from operator import lnot

def lnot(n):
    return not n

from operator import bool

def bool(n):
    return n

from operator import not_  

def not_(n):
    return not n

from operator import xnor

def xnor(n):
    return n

from operator import iand

def iand(n):
    return n

from operator import ior

def ior(n):
    return n

from operator import ixor

def ixor(n):
    return n

from operator import ilshift

def ilshift(n):
    return n

from operator import irshift

def irshift(n):
    return n

from operator import int

def int(n):
    return n

from operator import float

def float(n):
    return n

from operator import str

def str(n):
    return n
Dictionary keys do have two restrictions: size of the problem evaluated when the list comprehension is evaluated.

Unlike generator expressions, the map expression is

2.

1.

• A key of a dictionary

• Two keys cannot be equal. There can be at most one value for a key.

A range is a sequence of consecutive integers.

..., -5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, ...

Length. A sequence has a finite length.

Element selection. A sequence has an element corresponding to any non-negative integer index less than its length, starting at 0 for the first element.

Generator expressions

{<map exp> for <name> in <iter exp> if <filter exp>}

• Evaluates to an iterable object.

• <iter exp> is evaluated when the generator expression is evaluated.

• Remaining expressions are evaluated when elements are accessed.

List comprehensions

{<map exp> for <name> in <iter exp> if <filter exp>}

Short version: {<map exp> for <name> in <iter exp>}

Unlike generator expressions, the map expression is evaluated when the list comprehension is evaluated.

Mutable values can be changed without a nonlocal statement.

def pig_latin(w):
    if starts_with_a_vowel(w):
        return w + 'ay'
    return start_with_a_vowel(w[1:]) + w[0]

Python pre-computes which frame contains each name before executing the body of a function.

Therefore, within the body of a function, all instances of a name must refer to the same frame.

Every object that is an instance of a user-defined class has a unique identity:

>>> a = Account('Jim')
>>> b = Account('Jack')

Identity testing is performed by "is" and "is not" operators. Binding an object to a new name using assignment does not create a new object:

>>> a is a
True
>>> c = a
>>> a is c
True
>>> a is not b
True

Defining a class:

def make_withdraw(balance)
    def withdraw(amount):
        if amount > balance:
            return 'Insufficient Funds'
        return withdraw
    return withdraw

B. No nonlocal statement

"x" is also bound locally

SyntaxError: no binding for nonlocal "x" found

C. No nonlocal statement

"x" is bound locally

Re-bind "x" to 2 in the first frame of the current environment.

A function with a parent frame

Every call changes the balance

Status

- No nonlocal statement
- "x" is not bound locally

Effect: Create a new binding from name "x" to object 2 in the first frame of the current environment.

- No nonlocal statement
- "x" is bound locally

Re-bind "x" to 2 in the first frame of the current env.

- nonlocal x
- "x" is bound in a non-local frame (but not the global frame)

SyntaxError: no binding for nonlocal "x" found

- nonlocal x
- "x" is bound in a non-local frame

- "x" also bound locally

- nonlocal x
- "x" is bound in a non-local frame

- "x" is parameter and nonlocal

Python Docs: an "enclosing scope"

The parent contains local state
Assignment statements with a dot expression on their left-hand side affect attribute values for the object of that dot expression.

1. If the object is an instance, then assignment sets an instance attribute.
2. If the object is a class, then assignment sets a class attribute.

Instance Attribute Assignment

>>> jim_account = Account('Jim')
>>> jim_account.balance = 0
>>> tom_account = Account('Tom')
>>> tom_account.balance = 0

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.

class Account(object):
    def __init__(self, account_holder):
        self.balance = 0
        self.account_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
    def __str__(self):
        return 'Account: %s, balance: %d' % (self.account_holder, self.balance)

Assignment affects instance attributes

class Account(object):
    def __init__(self, account_holder):
        self.balance = 0
        self.account_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
    def __str__(self):
        return 'Account: %s, balance: %d' % (self.account_holder, self.balance)

class CheckingAccount(Account):
    def withdraw_fee(self):
        return 1
    def __init__(self, account_holder):
        Account.__init__(self, account_holder)
        self.withdraw_fee = 1
    def withdraw(self, amount):
        return Account.withdraw(self, amount + self.withdraw_fee)

class SavingsAccount(Account):
    def deposit_fee(self, amount):
        return Account.deposit(self, amount - self.deposit_fee)
    def __init__(self, account_holder):
        Account.__init__(self, account_holder)
        self.deposit_fee = 2
    def deposit(self, amount):
        return Account.deposit(self, amount - self.deposit_fee)

class AsSeenOnTvAccount(CheckingAccount, SavingsAccount):
    def __init__(self, account_holder):
        CheckingAccount.__init__(self, account_holder)
        SavingsAccount.__init__(self, account_holder)
        self.deposit_fee = 2
        self.withdraw_fee = 1

Type dispatching: Define a different function for each possible combination of types for which an operation is valid

def iscomplex(z):
    return type(z) in (complex, complex8, complex16)
def isrational(z):
    return type(z) == Rational

def add_complex_and_rational(z, r):
    return ComplexR(z.real + r.real, z.imag + r.imag)

def add_by_type dispatching(z1, z2):
    if iscomplex(z1) and iscomplex(z2):
        return add_complex_and_rational(z1, z2)
    elif iscomplex(z1) and isrational(z2):
        return add_complex_and_rational(z1, z2)
    elif isrational(z1) and iscomplex(z2):
        return add_complex_and_rational(z1, z2)
    else:
        return add_rational(z1, z2)