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.

Last name

First name

SID

Login

TA & section time

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)

For staff use only

<table>
<thead>
<tr>
<th>Q. 1</th>
<th>Q. 2</th>
<th>Q. 3</th>
<th>Q. 4</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<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 `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 causes 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 countdown(s, t):
    buzz = [t]
    def nas(a):
        nonlocal t
        t = buzz[0] + 's'
        buzz.append(t)
        return s(a)
    def aldrin():
        return buzz
    return nas, aldrin

def endeavor(k):
    return k * len(discovery())

atlantis, discovery = countdown(endeavor, 'u')
```

<table>
<thead>
<tr>
<th>Expression</th>
<th>Evaluates to</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>square(5)</code></td>
<td><code>25</code></td>
</tr>
<tr>
<td><code>discovery()</code></td>
<td></td>
</tr>
<tr>
<td><code>atlantis(1)</code></td>
<td></td>
</tr>
<tr>
<td><code>atlantis(len(discovery()))</code></td>
<td></td>
</tr>
<tr>
<td><code>discovery()</code></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 causes 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:

class Lawyer(object):
    def __init__(self, s):
        if len(s) < 2:
            self.s = s
        else:
            self.s = Lawyer(s[2:])

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

def think(self):
    if hasattr(self, 'decide'):
        return self.decide()
    while type(self.s) == Lawyer:
        self.s = self.s.s
    return self.s

class CEO(Lawyer):
    def decide(self):
        return 'Denied'

obama = Lawyer(['a', 'b', 'c'])
romney = CEO(['x', 'y', 'z'])

<table>
<thead>
<tr>
<th>Expression</th>
<th>Evaluates to</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>square(5)</code></td>
<td>25</td>
</tr>
<tr>
<td><code>obama.think()</code></td>
<td></td>
</tr>
<tr>
<td><code>obama</code></td>
<td></td>
</tr>
<tr>
<td><code>romney</code></td>
<td></td>
</tr>
<tr>
<td><code>Lawyer.think(romney)</code></td>
<td></td>
</tr>
</tbody>
</table>
2. (12 points) Picture Frame

(a) (6 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 oski(bear):
    def cal():
        nonlocal bear
        if bear == 0:
            return bear
        furd = bear
        bear = bear - 1
        return (furd, cal())
    return cal()

oski(2)
```

Global frame

```
oski
    func oski(bear)
```

Return Value
(b) (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, ee):
    b[oo] = [b[ee], oo, [b[ee]]]
    return b[oo]

b = list(range(3, 6))
beep(0, 1).append('not found')
```

(c) (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 Vehicle pass. In addition, cross out all lines that 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(Car, ‘tires’, Car.tires)
        setattr(self, ‘tires’, self.tires)
        setattr(self, ‘tires’, type(self).tires)
        setattr(self, ‘tires’, self.Car.tires)
```
(b) (6 pt) The\ max\_path\ function\ takes\ an\ instance\ of\ the\ Tree\ class\ from\ Study\ Guide\ 2.\ It\ is\ meant\ to
return\ the\ maximal\ sum\ of\ internal\ 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.\"""

    if\ tree.right\ is\ not\ None:
        paths.append(max_path(tree.right))
    if\ tree.left\ is\ not\ None:
        paths.append(max_path(tree.left))
    tree.entry += max(paths)
    return tree.entry
```

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

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

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

iii. \textbf{True/FALSE} It may change (mutate) its argument value.

iv. \textbf{True/FALSE} It may run forever on a valid tree.

(c) (2 pt) Define a simple mathematical function $f(n)$ such that evaluating \texttt{max_path(tree)} on a tree with $n$ entries performs $\Theta(f(n))$ function calls.

$f(n) =$
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 `list_anagrams`, which lists all anagrams (reorderings of the letters) of a given word. You may assume that the word has no repeated letters. Some hints about string slicing appear in the doctest.

```python
def list_anagrams(w):
    """List all anagrams of word w."

    >>> w = 'ate'
    >>> w[:0]
    '',
    >>> w[len(w):]
    '',
    >>> list_anagrams(w)
    ['ate', 'aet', 'tae', 'tea', 'eat', 'eta']
    """

    if w == '':

        return ______________________________

    anagrams = []

    for i in range(len(w)):

        subgrams = ______________________________

        anagrams += [__________________________ for s in subgrams]

    return anagrams
```

(c) (0 pt) Draw a picture of PusherBot 5.
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:
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.

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.
3. If the result is a true value, evaluate to v.
4. Otherwise, evaluate to the value of the subexpression on the right.

Execution rule for for 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>.

Execution rule for while statements:
1. Evaluate the header's expression.
2. If it is a true value, return the (whole) suite, then return to step 1.
3. If the result is a false value, end the while statement.
def square(x):
    return x * x

# Example usage
square(3) # Returns 9
square(5) # Returns 25

# Exercise
def sum_squares(x, y):
    return square(x) + square(y)

# Example usage
sum_squares(2, 3) # Returns 13
sum_squares(4, 5) # Returns 41
1. Lists are mutable sequences.
2. Tuples are immutable sequences.

Example:

```python
>>> a = [1, 2, 3]
>>> b = (1, 2, 3)
```

A set is an unordered collection of unique elements.

```python
>>> a = {1, 2, 3}
>>> b = {1, 2, 2, 3}
```

### Conditionals

A sequence has a finite length.

```python
>>> suit = ('hearts', 'diamonds', 'spades', 'clubs')
```

### Dictionaries

A dictionary is an unordered collection of key-value pairs.

```python
>>> dict1 = {'a': 1, 'b': 2}
```

### Functions and Scope

- `def` statements define functions.
- `return` statements return values from functions.

Example:

```python
def char_to_num(char):
    if char.isalpha():
        return ord(char) - ord('a')
    else:
        return 0
```

### Global and Local Scope

- `nonlocal` keyword can access variables in a non-local frame.

Example:

```python
def outer_function():
    x = 10
    def inner_function():
        nonlocal x
        x = 20
    inner_function()
    print(x)  # Output: 20
```

### Generators

Generators are iterable expressions that yield values one at a time.

```python
>>> fib = (n for n in range(10))
```

### Generator Expressions

Generator expressions are similar to list comprehensions.

```python
>>> cubes = (n**3 for n in range(10))
```

### Tuples

Tuples are immutable collections of values.

```python
>>> b = (1, 2, 3)
```

### Status

- **Nonlocal Statement:**
  - `x` is not bound locally
  - `x` is bound locally
  - `x` is bound in a non-local frame

### Library Import

```python
import unicodedata
```

### Calls to `fib`

- Base cases are evaluated for sufficiently large values of `n`.

```python
def fib(n):
    if n <= 1:
        return n
    else:
        return fib(n-1) + fib(n-2)
```

### Identity Testing

Identity testing is performed by `is` and `is not` operators.

```python
>>> a = [1, 2, 3]
>>> b = [1, 2, 3]
```

### Python Pre-computes Which Frame Contains Each Name

Python precomputes which frame contains each name before executing the body of a function.

### Function Environment

- Function environment is created for each function call.

### Execution Frame

- Execution frame contains local variables and function body.

### Global Environment

- Global environment contains global variables.

### Call Stack

- Call stack tracks function calls and returns.

### Call to `fib`

- A recursive call is made to `fib(n-1)`.

### Function Environment

- Function environment is created for each function call.

### Global Environment

- Global environment contains global variables.

### Call Stack

- Call stack tracks function calls and returns.

### Call to `fib`

- A recursive call is made to `fib(n-1)`.

### Function Environment

- Function environment is created for each function call.

### Global Environment

- Global environment contains global variables.

### Call Stack

- Call stack tracks function calls and returns.

### Call to `fib`

- A recursive call is made to `fib(n-1)`.

### Function Environment

- Function environment is created for each function call.

### Global Environment

- Global environment contains global variables.

### Call Stack

- Call stack tracks function calls and returns.

### Call to `fib`

- A recursive call is made to `fib(n-1)`.

### Function Environment

- Function environment is created for each function call.

### Global Environment

- Global environment contains global variables.
To evaluate a dot expression: `<expression> . `<name>`:
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.

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.

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.

```
>>> jim_account = Account('Jim')
>>> jim_account.balance = 0
>>> jim_account.balance = 1
```

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

```
def __init__(self, account_holder):
    self.balance = 0
    self.holder = account_holder
```

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 (ComplexRI, ComplexMA)
def isrational(z):
    return type(z) == Rational
```

```
def __call__(self, x, y):
    if x == y:
        return 'No coercion possible.'
    key = (operator_name, x, y)
    coer = self.coercion_map.get(key, None)
    if coer:
        return coer(x, y)
    return 'No coercion possible.'
```