Announcements

☐ HW6 due next Thursday

☐ Trends project due on Tuesday
  ☐ Partners are required; find one in lab or on Piazza
  ☐ Will not work in IDLE
  ☐ New bug submission policy; see Piazza
Names typically don’t matter for correctness, but they matter tremendously for legibility

boolean \( \rightarrow \) turn_is_over \hspace{0.5cm} \text{d} \hspace{0.5cm} \text{dice} \hspace{0.5cm} \text{play_helper} \hspace{0.5cm} \text{take_turn}

Use names for repeated compound expressions

\[
\text{if } \sqrt{\text{square}(a) + \text{square}(b)} > 1: \\
\quad x = x + \sqrt{\text{square}(a) + \text{square}(b)}
\]

\[
\quad h = \sqrt{\text{square}(a) + \text{square}(b)} \\
\text{if } h > 1: \\
\quad x = x + h
\]

Use names for meaningful parts of compound expressions

\[
\quad x = (-b + \sqrt{\text{square}(b) - 4 * a * c}) / (2 * a)
\]

\[
\quad \text{disc_term} = \sqrt{\text{square}(b) - 4 * a * c} \\
\quad x = (-b + \text{disc_term}) / (2 * a)
\]
Sometimes, removing repetition requires restructuring the code

```python
def find_quadratic_root(a, b, c, plus=True):
    """Applies the quadratic formula to the polynomial ax^2 + bx + c.""
    if plus:
        return (-b + sqrt(square(b) - 4 * a * c)) / (2 * a)
    else:
        return (-b - sqrt(square(b) - 4 * a * c)) / (2 * a)
```

```python
def find_quadratic_root(a, b, c, plus=True):
    """Applies the quadratic formula to the polynomial ax^2 + bx + c.""
    disc_term = sqrt(square(b) - 4 * a * c)
    if not plus:
        disc_term *= -1
    return (-b + disc_term) / (2 * a)
```
Test-Driven Development

Write the test of a function before you write a function

- A test will clarify the (one) job of the function
- Your tests can help identify tricky edge cases

Develop incrementally and test each piece before moving on

- You can’t depend upon code that hasn’t been tested
- Run your old tests again after you make new changes
Hog Contest

Contest rules:

- All entries run against every other entry
- An entry wins a match if its true win rate is > 0.5
- All strategies must be deterministic, pure functions and must not use pre-computed data
- Extra credit for entries with the most wins or the highest cumulative win rate
- Total of 54 valid submissions

We used `itertools.combinations` to determine the set of matches
Top Finishers

Congratulations to the team of Colin Lockard and Sherry Xu, who achieved a perfect 53-0 record and the highest win rate (28.77)!

Second-most wins (51-2): Eric Holt and Anna Carey

Second-highest win rate (28.70): Don Mai and Jeechee Chen

Third-highest in both (50-3, 28.67): Sean Scofield and Frank Lu

Complete rankings will be posted on the website
Computing Win Rates Exactly

A state in the game:
(who rolls next?, player score, opponent score)

A strategy is a table

\[(me,0,0): 5\]
\[(me,0,70): 9\]
\[...\]
\[(me,96,99): 0\]
\[...\]
\[(me,99,99): 10\]

Each state has a chance to win

When rolling 2 dice:
\[\frac{1}{36} \times 1 + \frac{35}{36} \times 0\]
\[...\]
\[(you,88,99): 0\]
\[(you,90,99): 0\]
\[...\]
\[(you,98,99): 0\]
\[(you,100+,99): 1\]
\[(me,99,100+): 0\]

Requires access to both strategies, which must be deterministic
Achieving the Perfect Strategy

Optimal strategy given an opponent:

- At each state, compute probability of winning for each allowed number of dice
- Choose the number of dice that maximizes the probability

The perfect strategy: use iterative improvement!

- Initial guess: always roll 5
- Update to: optimal opponent of current strategy
- Done when: 0.5 win rate against optimal opponent

Takes only 16 steps to converge!

Can also compute perfect strategy directly using table
A Function with Evolving Behavior

Let's model a bank account that has a balance of $100

Return value: remaining balance

Argument: amount to withdraw

Different return value!

Second withdrawal of the same amount

Where's this balance stored?

Within the function!

>>> withdraw(25)
75

>>> withdraw(25)
50

>>> withdraw(60)
'Insufficient funds'

>>> withdraw(15)
35

>>> withdraw = make_withdraw(100)
Persistent Local State

A function with a parent frame

The parent contains local state

Every call changes the balance

Example: http://goo.gl/5LZ6F
Reminder: Local Assignment

**Execution rule for assignment statements:**

1. Evaluate all expressions right of `=`, from left to right.
2. Bind the names on the left the resulting values in the first frame of the current environment.

Example: [http://goo.gl/xkYgN](http://goo.gl/xkYgN)
def make_withdraw(balance):

    """Return a withdraw function with a starting balance."""

def withdraw(amount):
    nonlocal balance

    if amount > balance:
        return 'Insufficient funds'

    balance = balance - amount

    return balance

return withdraw
The Effect of Nonlocal Statements

```
nonlocal <name>, <name 2>, ...
```

Effect: Future assignments to that name change its pre-existing binding in the first non-local frame of the current environment in which that name is bound.

From the Python 3 language reference:

Names listed in a `nonlocal` statement must refer to pre-existing bindings in an enclosing scope. Names listed in a nonlocal statement must not collide with pre-existing bindings in the local scope.

http://docs.python.org/release/3.1.3/reference/simple_stmts.html#the-nonlocal-statement

http://www.python.org/dev/peps/pep-3104/
## Effects of Assignment Statements

<table>
<thead>
<tr>
<th>Status</th>
<th>Effect</th>
</tr>
</thead>
</table>
| • No nonlocal statement  
  • "x" is not bound locally | 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 name "x" to object 2 in the first frame of the current env. |
| • 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 it 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 |

\[ x = 2 \]
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.

```python
def make_withdraw(balance):
    def withdraw(amount):
        if amount > balance:
            return 'Insufficient funds'
        balance = balance - amount
        return balance
    return withdraw

wd = make_withdraw(20)
wd(5)
```

UnboundLocalError: local variable 'balance' referenced before assignment
Mutable values and Persistent State

Mutable values can be changed without a nonlocal statement.

Example: [link](http://goo.gl/cEpmz)
Creating Two Withdraw Functions

Example: http://goo.gl/glTyB
Multiple References to a Withdraw Function

Example: http://goo.gl/X2qG9
The Benefits of Non-Local Assignment

- Ability to maintain some state that is local to a function, but evolves over successive calls to that function.

- The binding for balance in the first non-local frame of the environment associated with an instance of withdraw is inaccessible to the rest of the program.

- An abstraction of a bank account that manages its own internal state.

<table>
<thead>
<tr>
<th>Account Type</th>
<th>Balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Weasley Account</td>
<td>$10</td>
</tr>
<tr>
<td>Potter Account</td>
<td>$1,000,000</td>
</tr>
</tbody>
</table>
Express are referentially transparent if substituting an expression with its value does not change the meaning of a program.

\[
\text{mul}(\text{add}(2, \text{mul}(4, 6)), 3) \\
\text{mul}(\text{add}(2, 24), 3) \\
\text{mul}(26, 3)
\]

Mutation is a side effect (like printing)

Side effects violate the condition of referential transparency because they do more than just return a value; they change the state of the computer.