61A Lecture 8

Friday, February 6
Announcements
Abstraction
Functional Abstractions

```python
def square(x):
    return mul(x, x)
```

```python
def sum_squares(x, y):
    return square(x) + square(y)
```

What does `sum_squares` need to know about `square`?

- Square takes one argument.  Yes
- Square has the intrinsic name `square`.  No
- Square computes the square of a number.  Yes
- Square computes the square by calling `mul`.  No

```python
def square(x):
    return pow(x, 2)
def square(x):
    return mul(x, x-1) + x
```

If the name “square” were bound to a built-in function, `sum_squares` would still work identically.
Choosing Names

Names typically don’t matter for correctness

*but*

they matter a lot for composition

<table>
<thead>
<tr>
<th>From:</th>
<th>To:</th>
</tr>
</thead>
<tbody>
<tr>
<td>true_false</td>
<td>rolled_a_one</td>
</tr>
<tr>
<td>d</td>
<td>dice</td>
</tr>
<tr>
<td>helper</td>
<td>take_turn</td>
</tr>
<tr>
<td>my_int</td>
<td>num_rolls</td>
</tr>
<tr>
<td>l, I, 0</td>
<td>k, i, m</td>
</tr>
</tbody>
</table>

Names should convey the meaning or purpose of the values to which they are bound.

The type of value bound to the name is best documented in a function's docstring.

Function names typically convey their effect (print), their behavior (triple), or the value returned (abs).
Which Values Deserve a Name

Reasons to add a new name

Repeated compound expressions:

```python
if sqrt(square(a) + square(b)) > 1:
    x = x + sqrt(square(a) + square(b))
```

```python
hypotenuse = sqrt(square(a) + square(b))
if hypotenuse > 1:
    x = x + hypotenuse
```

Meaningful parts of complex expressions:

```python
x = (-b + sqrt(square(b) - 4 * a * c)) / (2 * a)
```

```python
discriminant = sqrt(square(b) - 4 * a * c)
x = (-b + discriminant) / (2 * a)
```

More Naming Tips

- Names can be long if they help document your code:

  ```python
  average_age = average(age, students)
  ```

  is preferable to

  ```python
  # Compute average age of students
  aa = avg(a, st)
  ```

- Names can be short if they represent generic quantities: counts, arbitrary functions, arguments to mathematical operations, etc.

  ```python
  n, k, i - Usually integers
  x, y, z - Usually real numbers
  f, g, h - Usually functions
  ```
Testing
Test-Driven Development

Write the test of a function before you write the function.

A test will clarify the domain, range, & behavior of a function.

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.

Bonus idea: Run your code interactively.

Don't be afraid to experiment with a function after you write it.

Interactive sessions can become doctests. Just copy and paste.

(Demo)
Currying
Function Currying

```python
def make_adder(n):
    return lambda k: n + k
```

```python
>>> make_adder(2)(3)
5
>>> add(2, 3)
5
```

There's a general relationship between these functions

Curry: Transform a multi-argument function into a single-argument, higher-order function
Function Decorators

(Demo)

@trace1
def triple(x):
    return 3 * x

is identical to

Why not just use this?

def triple(x):
    return 3 * x

triple = trace1(triple)
Review
What Would Python Print?

The print function returns None. It also displays its arguments (separated by spaces) when it is called.

```python
from operator import add, mul

def square(x):
    return mul(x, x)

def delay(arg):
    print('delayed')
    def g():
        return arg
    return g

print(print(5))

print(delay(print)(5))

print(delay(delay)(6))

print(delay(print)(4))
```

<table>
<thead>
<tr>
<th>This expression</th>
<th>Evaluates to</th>
<th>Interactive Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>print(5)</td>
<td>None</td>
<td>5 None</td>
</tr>
<tr>
<td>print(print(5))</td>
<td>None</td>
<td>5 None</td>
</tr>
<tr>
<td>delay(delay)(6)()</td>
<td>6</td>
<td>delayed 6</td>
</tr>
<tr>
<td>delay(print)(4)</td>
<td>None</td>
<td>delayed 4 None</td>
</tr>
</tbody>
</table>
```python
def horse(mask):
    horse = mask
    def mask(horse):
        return horse
    return horse(mask)

def horse(mask):
    horse = mask
    def mask(horse):
        return horse
    return horse(mask)
```

\[
\text{Global frame}
\]

\[
\text{func horse(mask) [parent=Global]}
\]

\[
\text{func \lambda(horse) [parent=Global]}
\]

\[
\text{func mask(horse) [parent=f1]}
\]

\[
\text{func mask(horse) [parent=f1]}
\]