Recursion
Announcements
Abstraction
Functional Abstractions

- Square takes one argument.
- Square has the intrinsic name square.
- Square computes the square of a number.
- Square computes the square by calling \texttt{mul}.

If the name “square” were bound to a built-in function, \texttt{sum_squares} would still work identically.

\begin{verbatim}

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

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

def square(x):
    return mul(x, x-1) + x

\end{verbatim}

What does \texttt{sum_squares} need to know about \texttt{square}?

- Square takes one argument. \hspace{1cm} Yes
- Square has the intrinsic name square. \hspace{1cm} No
- Square computes the square of a number. \hspace{1cm} Yes
- Square computes the square by calling \texttt{mul}. \hspace{1cm} No

\begin{verbatim}

def square(x):
    return pow(x, 2)

\end{verbatim}
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**).
Recursive Functions
Recursive Functions

Definition: A function is called recursive if the body of that function calls itself, either directly or indirectly.

Implication: Executing the body of a recursive function may require applying that function.
Digit Sums

• If a number a is divisible by 9, then \( \text{sum_digits}(a) \) is also divisible by 9
• Useful for typo detection!

\[
2+0+1+9 = 12
\]

• Credit cards actually use the Luhn algorithm, which we'll implement after \( \text{sum_digits} \)
The Problem Within the Problem

The sum of the digits of 6 is 6.

Likewise for any one-digit (non-negative) number (i.e., < 10).

The sum of the digits of 2019 is

That is, we can break the problem of summing the digits of 2019 into a smaller instance of the same problem, plus some extra stuff.

We call this recursion
def split(n):
    """Split positive n into all but its last digit and its last digit."""
    return n // 10, n % 10

def sum_digits(n):
    """Return the sum of the digits of positive integer n."""
    if n < 10:
        return n
    else:
        all_but_last, last = split(n)
        return sum_digits(all_but_last) + last
The Anatomy of a Recursive Function

• The **def statement header** is similar to other functions
• Conditional statements check for **base cases**
• Base cases are evaluated **without recursive calls**
• Recursive cases are evaluated **with recursive calls**

```python
def sum_digits(n):
    """Return the sum of the digits of positive integer n."""
    if n < 10:
        return n
    else:
        all_but_last, last = split(n)
        return sum_digits(all_but_last) + last
```

(Demo)
Recursion in Environment Diagrams
Recursion in Environment Diagrams

- The same function `fact` is called multiple times.
- Different frames keep track of the different arguments in each call.
- What `n` evaluates to depends upon the current environment.
- Each call to `fact` solves a simpler problem than the last: smaller `n`.

(Demo)

```python
def fact(n):
    if n == 0:
        return 1
    else:
        return n * fact(n-1)

fact(3)
```

Global frame

```
func fact(n) [parent=Global]
```

```
f1: fact [parent=Global]
```

```
f2: fact [parent=Global]
```

```
f3: fact [parent=Global]
```

```
f4: fact [parent=Global]
```

```
Return value
```

```
n 3
```

```
n 2
```

```
n 1
```

```
n 0
```

1 2 3 4 5 6 7
Iteration vs Recursion

Iteration is a special case of recursion

\[ 4! = 4 \cdot 3 \cdot 2 \cdot 1 = 24 \]

Using while:

```python
def fact_iter(n):
    total, k = 1, 1
    while k <= n:
        total, k = total * k, k + 1
    return total
```

Using recursion:

```python
def fact(n):
    if n == 0:
        return 1
    else:
        return n * fact(n - 1)
```

Math:

\[ n! = \prod_{k=1}^{n} k \]

Names: \( n, \) total, \( k, \) fact_iter

\[ n! = \begin{cases} 
1 & \text{if } n = 0 \\
n \cdot (n - 1)! & \text{otherwise} 
\end{cases} \]

Names: \( n, \) fact
Verifying Recursive Functions
The Recursive Leap of Faith

def fact(n):
    if n == 0:
        return 1
    else:
        return n * fact(n-1)

Is fact implemented correctly?

1. Verify the base case

2. Treat fact as a functional abstraction!

3. Assume that fact(n-1) is correct

4. Verify that fact(n) is correct
Mutual Recursion
The Luhn Algorithm

Used to verify credit card numbers


• **First:** From the rightmost digit, which is the check digit, moving left, double the value of every second digit; if product of this doubling operation is greater than 9 (e.g., 7 * 2 = 14), then sum the digits of the products (e.g., 10: 1 + 0 = 1, 14: 1 + 4 = 5)

• **Second:** Take the sum of all the digits

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>3</th>
<th>8</th>
<th>7</th>
<th>4</th>
<th>3</th>
</tr>
</thead>
</table>
|   | 2 | 3 | 1+6=7 | 7 | 8 | 3 | = 30

The Luhn sum of a valid credit card number is a multiple of 10 (Demo)
Recursion and Iteration
Converting Recursion to Iteration

Idea: Figure out what state must be maintained by the iterative function.

```python
def sum_digits(n):
    """Return the sum of the digits of positive integer n."""
    if n < 10:
        return n
    else:
        all_but_last, last = split(n)
        return sum_digits(all_but_last) + last
```

Demo
Converting Iteration to Recursion

Idea: The state of an iteration are passed as arguments.

```python
def sum_digits_iter(n):
    digit_sum = 0
    while n > 0:
        n, last = split(n)
        digit_sum = digit_sum + last
    return digit_sum

def sum_digits_rec(n, digit_sum):
    if n > 0:
        n, last = split(n)
        return sum_digits_rec(n, digit_sum + last)
    else:
        return digit_sum
```
Order of Recursive Calls
The Cascade Function

Each cascade frame is from a different call to `cascade`. Until the Return value appears, that call has not completed. Any statement can appear before or after the recursive call.
Two Definitions of Cascade

(Demo)

def cascade(n):
    if n < 10:
        print(n)
    else:
        print(n)
        cascade(n//10)
        print(n)

def cascade(n):
    print(n)
    if n >= 10:
        cascade(n//10)
        print(n)

• If two implementations are equally clear, then shorter is usually better
• In this case, the longer implementation is more clear (at least to me)
• When learning to write recursive functions, put the base cases first
• Both are recursive functions, even though only the first has typical structure
Example: Inverse Cascade
Write a function that prints an inverse cascade:

```python
def inverse_cascade(n):
    grow(n)
    print(n)
    shrink(n)

def f_then_g(f, g, n):
    if n:
        f(n)
        g(n)

grow = lambda n: f_then_g(grow, print, n // 10)
shrink = lambda n: f_then_g(print, shrink, n // 10)
```