Lecture #5: Higher-Order Functions
def combine_funcs(op):
    def combined(f, g):
        def val(x):
            return op(f(x), g(x))
        return val
    return combined

add_func = combine_funcs(add)
def combine_funcs(op):
    def combined(f, g):
        def val(x):
            return op(f(x), g(x))
        return val
    return combined

add_func = combine_funcs(add)

h = add_func(abs, neg)
def combine_funcs(op):
    def combined(f, g):
        def val(x):
            return op(f(x), g(x))
        return val
    return combined

add_func = combine_funcs(add)
h = add_func(abs, neg)
h(-5)

...and local frames for
• add (value of op),
• abs (value of f), and
• neg (value of g)
Do You Understand the Machinery? (IV)

What is printed: (1, infinite loop, or error) and why?

def g(x):
    print(x)

def f(f):
    f(1)

f(g)

This prints 1. When we reach $f(1)$ inside $f$, the call expression, and therefore the name $f$, evaluated in the environment $E$, where the value of $f$ is the global function bound to $g$:

```
def g(x):
    print(x)
def f(f):
    f(1)
f(g)
```

```
Global frame
  g
  f
    func g() [↑ Global]

f1: f [↑ Global]
  f

f(1) evaluated here
```
Do You Understand the Machinery? (V)

What is printed: (0, 1, or error) and why?

```python
def f():
    return 0

def g():
    return f()

def h(k):
    def f():
        return 1
    p = k
    return p()

print(h(g))
```

Last modified: Fri Jan 29 16:37:11 2016
Answer (V)

This prints 0. Function values are attached to current environments when they are first created (by `lambda` or `def`). Assignments (such as to `p`) don’t themselves create new values, but only copy old ones, so that when `p` is evaluated, it is equal to `k`, which is equal to `g`, which is attached to the global environment.
Observation: Environments Reflect Nesting

• From what we’ve seen so far:

    Linking of environment frames ⇔ Nesting of definitions.

• For example, given

    ```python
def f(x):
        def g(x):
            def h(x):
                print(x)
            ...
    ...
```

    The structure of the program tells you that the environment in which `print(x)` is evaluated will always be a chain of 4 frames:

    - A local frame for `h` linked to …
    - A local frame for `g` linked to …
    - A local frame for `f` linked to …
    - The global frame.

• However, when there are multiple local frames for a particular function lying around, environment diagrams can help sort them out.
Do You Understand the Machinery? (VI)

What is printed: (0, 1, or error) and why?

def f(p, k):
    def g():
        print(k)
        if k == 0:
            f(g, 1)
        else:
            p()
    f(None, 0)
This prints 0. There are two local frames for $f$ when $p()$ is called ($f_1$ and $f_2$). The call to $p()$ creates an instantiation of $g$ whose parent is $f_1$.

def f(p, k):
    def g():
        print(k)
        if k == 0:
            f(g, 1)
        else:
            p()
    f(None, 0)
Higher-Order Functions at Work: Iterative Update

- A general strategy for solving an equation:
  - Guess a solution
  - while your guess isn’t good enough:
    * update your guess
- The three underlined segments are parameters to the process.
- The last two segments clearly require functions for their representation—a \textit{predicate} function (returning true/false values), and a function from values to values.

- In code,

  ```python
def iter_solve(guess, done, update):
    """Return the result of repeatedly applying UPDATE, starting at GUESS, until DONE yields a true value when applied to the result.""
  ```
Recursive Versions

def iter_solve(guess, done, update):
    """Return the result of repeatedly applying UPDATE, starting at GUESS, until DONE yields a true value when applied to the result.""
    if done(guess):
        return guess
    else:
        return iter_solve(update(guess), done, update)

or

def iter_solve(guess, done, update):
    def solution(guess):
        if done(guess):
            return guess
        else:
            return solution(update(guess))
    return solution(guess)
def iter_solve(guess, done, update):
    """Return the result of repeatedly applying UPDATE,
    starting at GUESS, until DONE yields a true value
    when applied to the result."""
    while not done(guess):
        guess = update(guess)
    return guess
Adding a Safety Net

- In real life, we might want to make sure that the function doesn’t just loop forever, getting no closer to a solution.

```python
def iter_solve(guess, done, update, iteration_limit=32):
    """Return the result of repeatedly applying UPDATE, starting at GUESS, until DONE yields a true value when applied to the result. Causes error if more than ITERATION_LIMIT applications of UPDATE are necessary.""
```
• In real life, we might want to make sure that the function doesn’t just loop forever, getting no closer to a solution.

```python
def iter_solve(guess, done, update, iteration_limit=32):
    """Return the result of repeatedly applying UPDATE, starting at GUESS, until DONE yields a true value when applied to the result. Causes error if more than ITERATION_LIMIT applications of UPDATE are necessary.""

    def solution(guess, iteration_limit):
        if done(guess):
            return guess
        elif iteration_limit <= 0
            raise ValueError("failed to converge")
        else:
            return solution(update(guess), iteration_limit-1)
    return solution(guess, iteration_limit)
```
def iter_solve(guess, done, update, iteration_limit=32):
    """Return the result of repeatedly applying UPDATE, starting at GUESS, until DONE yields a true value
    when applied to the result. Causes error if more than ITERATION_LIMIT applications of UPDATE are necessary.""

    while not done(guess):
        if iteration_limit <= 0:
            raise ValueError("failed to converge")
        guess, iteration_limit = update(guess), iteration_limit-1
    return guess
Newton’s method takes a function, its derivative, and an initial guess, and produces a result to some desired tolerance (that is, to some definition of “close enough”).


Given a guess, $x_k$, compute the next guess, $x_{k+1}$ by

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}$$

```python
def newton_solve(func, deriv, start, tolerance):
    """Return x such that |FUNC(x)| < TOLERANCE, given initial estimate START and assuming DERIV is the derivatative of FUNC.""
    def close_enough(x):
        ___________
    def newton_update(x):
        ___________
    return iter_solve(start, close_enough, newton_update)
```
Using Iterative Solving for Newton’s Method (II)

\[ x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)} \]

def newton_solve(func, deriv, start, tolerance):
    """Return x such that |FUNC(x)| < TOLERANCE, given initial estimate START and assuming DERIV is the derivative of FUNC."""
    def close_enough(x):
        return abs(func(x)) < tolerance
    def newton_update(x):
        return x - func(x) / deriv(x)

    return iter_solve(start, close_enough, newton_update)
Using \texttt{newton\_solve} for $\sqrt{\cdot}$ and $\log_\cdot$. 

\begin{verbatim}
def square_root(a):
    return newton_solve(lambda x: x\*x - a, lambda x: 2 \* x,
                      a/2, 1e-5)

def logarithm(a, base = 2):
    return newton_solve(lambda x: base**x - a,
                        lambda x: x \* base**(x-1),
                        1, 1e-5)
\end{verbatim}
Dispensing With Derivatives

• What if we just want to work with a function, without knowing its derivative?

• Book uses an approximation:

```python
def find_root(func, start=1, tolerance=1e-5):
    def approx_deriv(f, delta = 1e-5):
        return lambda x: (func(x + delta) - func(x)) / delta
    return newton_solve(func, approx_deriv(func), start, tolerance)
```

• This is nice enough, but looks a little ad hoc (how did I pick delta?).

• Another alternative is the secant method.
The Secant Method

• Newton’s method was

\[ x_{k+1} = x_k - \frac{f(x)}{f'(x)} \]

• The secant method uses that last two values to get (in effect) a replacement for the derivative:

\[ x_{k+1} = x_k - f(x_k) \frac{x_k - x_{k-1}}{f(x_k) - f(x_{k-1})} \]


• But this is a problem for us: so far, we’ve only fed the update function the value of \( x_k \) each time. Here we also need \( x_{k-1} \).

• How do we generalize to allow arbitrary extra data (not just \( x_{k-1} \))?
def iter_solve2(guess, done, update, state=None):
    """Return the result of repeatedly applying UPDATE, starting at GUESS and STATE, until DONE yields a true value when applied to the result. Besides a guess, UPDATE also takes and returns a state value, which is also passed to DONE."""
    while not done(guess, state):
        guess, state = update(guess, state)
    return guess
Using Generalized iter_solve2 for the Secant Method

The secant method:

\[ x_{k+1} = x_k - f(x_k) \frac{x_k - x_{k-1}}{f(x_k) - f(x_{k-1})} \]

```python
def secant_solve(func, start0, start1, tolerance):
    def close_enough(x, state):
        return abs(func(x)) < tolerance
    def secant_update(xk, xk1):
        return (xk - func(xk) * (xk - xk1) / (func(xk) - func(xk1),
                                                xk)
    return iter_solve2(start1, close_enough, secant_update, start0)
```

Last modified: Fri Jan 29 16:37:11 2016