Lecture #4: Higher-Order Functions
Announcements

• Pair-programming demo (Pamela Fox & Patricia Ouyang).
• Homework 1 due Thursday.
• Project 1 (Hog) release today.
• Nine new tutorials added:
  - 2 on Wed. @4PM
  - 1 on Thu. @7AM
  - 1 on Thu. @8AM
  - 3 on Thu. @11AM
  - 2 on Thu. @12PM
• “Lost” sections starting Friday at 12-2PM and 4-6PM. See Piazza @239.
• Ask questions on the Piazza thread for today’s lecture (@245).
Comments on Functions in General: Terminology

- The set of possible argument values of a function is known as its **domain**.

- The set of values that the function can return (all values that result from inputting some value from its domain) is called its **range**.

- The **codomain** of a function is a set of values that includes the range, and possibly other values.

- Thus, we might say that the square function has the real numbers as its domain, and the non-negative numbers as its range. We can choose to describe its codomain as the real numbers or as just the non-negative real numbers.
Documenting Functions

• Ideally, a *documentation comment* for a function provides enough information so that a programmer can use the function properly and understand what it does *without* having to read its body.

• It should make clear what inputs are valid or under what conditions the function may be called. This is the *precondition*.

• Likewise, it should make clear what the resulting output or effect of the function will be for correct inputs. This is the *postcondition*.

• Together, these are the *behavior* or *semantics* (meaning) of the function.
Two Design Principles

• Functions should do one well-defined thing (a complicated documentation comment might suggest your function does too much).

• **DRY** (Don’t Repeat Yourself).
  - Multiple segments of code that look really similar to each other cry out for *refactoring*…
  - That is, for replacing the segments with simple calls to a single general function that states their shared structure just once, with parameters used to specialize to the various cases.
Functions As Templates

- If we think of a function body as a template for a computation, parameters are “blanks” in that template.

- For example:

```python
def sum_squares(N):
    """Returns the sum of x**2 for all integers x with 1 <= x <= N.""
    k = 1
    sum = 0
    while k <= N:
        sum += k**2
        k += 1
    return sum
```

is a template for an infinite set of computations that add squares of numbers up to 0, 1, 2, 3, \ldots, in place of the \texttt{N}.

- But the \texttt{sum_squares} function is specialized to the summing \( k^2 \).

- A function for summing \( k^3 \), \( \sin k \), or \( 1/k \) would have the same structure, differing only in what comes after \texttt{sum +=}.

- How do we practice DRY here?
Functions on Functions

• Function parameters allow us to have templates with slots for *computations*:

```python
def summation(N, term):
    k = 1
    sum = 0
    while k <= N:
        sum += term(k)
        k += 1
    return sum
```

• Generalizes *sum_squares*. We can write *sum_squares(5)* as:

```python
def square(x):
    return x*x
summation(5, square)
```

• or (if we don't really need a “square” function elsewhere), we can create the function argument anonymously on the fly:

```python
summation(5, lambda x: x*x)
```
Quick Review of Lambda

• In Python, \texttt{lambda} is just an abbreviation.

• Writing \texttt{lambda \texttt{PARAMS}: \texttt{EXPRESSION}} is the same as writing \texttt{NEWNAME}, where \texttt{NEWNAME} is a name that appears nowhere else in the program and is defined by

\begin{verbatim}
def \texttt{NEWNAME}(\texttt{PARAMS}):
    \texttt{return \texttt{EXPRESSION}}
\end{verbatim}

evaluated in the same environment in which the original \texttt{lambda} was.

• There is no return: the body must be a single expression.

• Now we can write any number of summations succinctly:

\begin{verbatim}
summation(10, \texttt{lambda x: x**3})  # Sum of cubes
summation(10, \texttt{lambda x: 1 / x})  # Harmonic series
summation(10, \texttt{lambda k: x**(k-1) / factorial(k-1)})  
    # Approximate e**x
\end{verbatim}
Functions that Produce Functions

- Functions are **first-class values**, meaning that we can assign them to variables, pass them to functions, and return them from functions.

- Example: let’s generalize the class of functions that—like

```python
    def h(x): return sin(x) + cos(x)
```

—add the results of applying two functions to the same argument:

```python
>>> def add_func(f, g):
...     """Return function that returns F(x)+G(x) for argument x."""
...     def adder(x):
...         return f(x) + g(x)  # or return lambda x: f(x) + g(x)
...     return adder
```

```python
>>> from math import sin, cos, pi
>>> h = add_func(sin, cos)
>>> sin(pi/4) + cos(pi/4)
1.414213562373095
>>> h(pi / 4)
1.414213562373095
```
Generalize!

• Let’s make a general function-combining function (that goes beyond addition):

```python
>>> def combine_funcs(op):
...    """combine_funcs(OP)(f, g)(x) = OP(f(x), g(x)).""
...    def combined(f, g):
...        def val(x):
...            return op(f(x), g(x))
...        return val
...    return combined
```

• Now `add_func` itself can be constructed by a call to `combine_funcs`:

```python
>>> from operator import add
>>> add_func = ??
>>> from math import sin, cos, pi
>>> h = add_func(sin, cos)
>>> h(pi / 4)
1.414213562373095
```

• What do the environments look like here? Think about it and try it out.
Generalize!

• Let’s make a general function-combining function (that goes beyond addition):

```python
>>> def combine_funcs(op):
...     """combine_funcs(OP)(f, g)(x) = OP(f(x), g(x))."""
...     def combined(f, g):
...         def val(x):
...             return op(f(x), g(x))
...         return val
...     return combined
```

• Now `add_func` itself can be constructed by a call to `combine_funcs`:

```python
>>> from operator import add
>>> add_func = combine_funcs(add)
>>> from math import sin, cos, pi
>>> h = add_func(sin, cos)
>>> h(pi / 4)
1.414213562373095
```

• What do the environments look like here? Think about it and try it out.
Generalize!

• Let's make a general function-combining function (that goes beyond addition):

```python
>>> def combine_funcs(op):
...     """combine_funcs(OP)(f, g)(x) = OP(f(x), g(x)).""
...     def combined(f, g):
...         def val(x):
...             return op(f(x), g(x))
...         return val
...     return combined
```

• Now `add_func` itself can be constructed by a call to `combine_funcs`:

```python
>>> from operator import add
>>> add_func = combine_funcs(lambda x, y: x + y)
>>> from math import sin, cos, pi
>>> h = add_func(sin, cos)
>>> h(pi / 4)
1.414213562373095
```

• What do the environments look like here? Think about it and try it out.
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)
The Environment Picture (II)

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(sin, cos)
The Environment Picture (III)

```python
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(sin, cos)
h(-5)
```

+ local frames for calls to
- `add` (value of `op`),
- `sin` (value of `f`), and
- `cos` (value of `g`)
Challenge: Conditional Function?

- Write a Python function, `if_func`, such that, for example
  
  ```python
  if_func(1/x, x > 0, 0)
  
  always returns the same value as the conditional expression
  
  1/x if x > 0 else 0
  ```
Challenge: Conditional Function?

- Write a Python function, `if_func`, such that, for example
  ```python
  if_func(1/x, x > 0, 0)
  ```
  always returns the same value as the conditional expression
  ```python
  1/x if x > 0 else 0
  ```

**Answer:** IMPOSSIBLE! Function calls *always* evaluate all their operands.
**Challenge: Conditional Function?**

- Write a Python function, `if_func`, such that, for example
  
  ```python
  if_func(1/x, x > 0, 0)
  ```

  always returns the same value as the conditional expression
  
  ```python
  1/x if x > 0 else 0
  ```

  **Answer:** IMPOSSIBLE! Function calls *always* evaluate all their operands.

- But all is not lost, because we can define instead
  
  ```python
  def if_func(then_expr, condition, else_expr):
      return then_expr() if condition else else_expr()
  ```

  and call
  
  ```python
  if_func(lambda: 1/x, x > 0, lambda: 0)
  ```

  (The jargon term for those parameterless lambdas is *thunks.*)

- Why don’t we need a thunk for the condition?
Challenge: Conditional Function?

- Write a Python function, `if_func`, such that, for example
  ```python
  if_func(1/x, x > 0, 0)
  ```
  always returns the same value as the conditional expression
  ```python
  1/x if x > 0 else 0
  ```
  **Answer:** **IMPOSSIBLE!** Function calls *always* evaluate all their operands.

- But all is not lost, because we can define instead
  ```python
  def if_func(then_expr, condition, else_expr):
      return then_expr() if condition else else_expr()
  ```
  and call
  ```python
  if_func(lambda: 1/x, x > 0, lambda: 0)
  ```
- (The jargon term for those parameterless lambdas is *thunks*.)

- Why don't we need a thunk for the condition?
  **Answer:** Because the condition parameter must always be evaluated first anyway.