Higher-Order Functions
Announcements
Example: Prime Factorization
Prime Factorization

Each positive integer $n$ has a set of prime factors: primes whose product is $n$

...  
8  = 2 * 2 * 2  
9  = 3 * 3  
10 = 2 * 5  
11 = 11  
12 = 2 * 2 * 3  
...

One approach: Find the smallest prime factor of $n$, then divide by it

\[
858 = 2 \times 429 = 2 \times 3 \times 143 = 2 \times 3 \times 11 \times 13
\]

(Demo)
Example: Iteration
The Fibonacci Sequence

```
def fib(n):
    """Compute the nth Fibonacci number, for N >= 1."""
    pred, curr = 0, 1  # 0th and 1st Fibonacci numbers
    k = 1               # curr is the kth Fibonacci number
    while k < n:
        pred, curr = curr, pred + curr
        k = k + 1
    return curr
```

The next Fibonacci number is the sum of the current one and its predecessor.

```
The Fibonacci Sequence
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987
```
Go Bears!
Control and Call Expressions
def absolute_value(x):
    """Return the absolute value of x."""
    if x < 0:
        return -x
    elif x == 0:
        return 0
    else:
        return x
def absolute_value(x):
    """Return the absolute value of x."""
    if x < 0:
        return -x
    elif x == 0:
        return 0
    else:
        return x

Boolean Contexts

False values in Python: False, 0, "", None

True values in Python: Anything else (True)

(Demo)
Let's try to write a function that does the same thing as an if statement.

```python
def if_(c, t, f):
    if c:
        return t
    else:
        return f
```

Execution Rule for Conditional Statements:

Each clause is considered in order.

1. Evaluate the header's expression (if present).
2. If it is a true value (or an else header), execute the suite & skip the remaining clauses.

Evaluation Rule for Call Expressions:

1. Evaluate the operator and then the operand subexpressions
2. Apply the function that is the value of the operator to the arguments that are the values of the operands

(Demo)
Higher-Order Functions
Generalizing Over Computational Processes

The common structure among functions may be a computational process, rather than a number.

\[
\sum_{k=1}^{5} k = 1 + 2 + 3 + 4 + 5 = 15
\]

\[
\sum_{k=1}^{5} k^3 = 1^3 + 2^3 + 3^3 + 4^3 + 5^3 = 225
\]

\[
\sum_{k=1}^{5} \frac{8}{(4k - 3) \cdot (4k - 1)} = \frac{8}{3} + \frac{8}{35} + \frac{8}{99} + \frac{8}{195} + \frac{8}{323} = 3.04
\]

(Demo)
Summation Example

```python
def cube(k):
    return pow(k, 3)

def summation(n, term):
    """Sum the first n terms of a sequence."

>>> summation(5, cube)
225

total, k = 0, 1
while k <= n:
    total, k = total + term(k), k + 1
return total
0 + 1 + 8 + 27 + 64 + 125
```

Function of a single argument (not called “term”)

A formal parameter that will be bound to a function

The cube function is passed as an argument value

The function bound to term gets called here
Break
Types of Higher-Order Functions
Environments Enable Higher-Order Functions

Functions are first-class: Functions are values in our programming language

Higher-order function: A function that takes a function as an argument value or
A function that returns a function as a return value

(Demo)
Environments for Higher-Order Functions

(Demo)
Names can be Bound to Functional Arguments

Applying a user-defined function:
- Create a new frame
- Bind formal parameters (f & x) to arguments
- Execute the body: return f(f(x))

```python
def apply_twice(f, x):
    return f(f(x))

def square(x):
    return x * x

result = apply_twice(square, 2)
```
Functions as Return Values

(Demo)
Locally Defined Functions

Functions defined within other function bodies are bound to names in a local frame

A function that returns a function

```python
def make_adder(n):
    """
    Return a function that takes one argument k and returns k + n.
    """
    add_three = make_adder(3)
    add_three(4)
    7
    return adder
```

The name add_three is bound to a function

A def statement within another def statement

Can refer to names in the enclosing function
Call Expressions as Operator Expressions

An expression that evaluates to a function

An expression that evaluates to its argument

Operator

Operand

func make_adder(n)

func make_adder(1)

func adder(k)

make_adder(1) ( 2 )

make_adder( n ):

def adder(k):
    return k + n
    return adder

func adder(k)
Environments for Nested Definitions

(Demo)
Every user-defined function has a parent frame (often global).

The parent of a function is the frame in which it was defined.

Every local frame has a parent frame (often global).

The parent of a frame is the parent of the function called.
How to Draw an Environment Diagram

When a function is defined:

Create a function value: \( \text{func } \langle \text{name} \rangle(\langle \text{formal parameters} \rangle) \ [\text{parent}=\langle \text{label} \rangle] \)

Its parent is the current frame.

Bind \langle \text{name} \rangle to the function value in the current frame

When a function is called:

1. Add a local frame, titled with the \langle \text{name} \rangle of the function being called.
2. **Copy the parent of the function to the local frame: [parent=\langle \text{label} \rangle]**
3. Bind the \langle \text{formal parameters} \rangle to the arguments in the local frame.
4. Execute the body of the function in the environment that starts with the local frame.
Local Names

(Demo)
Local Names are not Visible to Other (Non-Nested) Functions

- An environment is a sequence of frames.
- The environment created by calling a top-level function (no def within def) consists of one local frame, followed by the global frame.

```python
def f(x, y):
    return g(x)

def g(a):
    return a + y

result = f(1, 2)
```

Error: "y" is not found, again

"y" is not found
Function Composition

(Demo)
The Environment Diagram for Function Composition

1. `def square(x):`
   return x * x

2. `def make_adder(n):`
   def adder(k):
     return k + n
   return adder

3. `def compose1(f, g):
   def h(x):
     return f(g(x))
   return h

4. `compose1(square, make_adder(2))(3)`

Return value of make_adder is an argument to compose1
Lambda Expressions

(Demo)
Lambda Expressions

```
>>> x = 10
>>> square = x * x
>>> square = lambda x: x * x
>>> square(4)
16
```

An expression: this one evaluates to a number

Also an expression: evaluates to a function

Important: No "return" keyword!

A function with formal parameter x that returns the value of "x * x"

Must be a single expression

Lambda expressions are not common in Python, but important in general

Lambda expressions in Python cannot contain statements at all!
Lambda Expressions Versus Def Statements

- Both create a function with the same domain, range, and behavior.
- Both bind that function to the name square.
- Only the def statement gives the function an intrinsic name, which shows up in environment diagrams but doesn't affect execution (unless the function is printed).

\[
\text{square} = \lambda x: x \times x
\]

\[
\text{def square}(x): \quad \text{return } x \times x
\]
Summary

- As we start to design functions ourselves, we want to think about giving them well-defined jobs that can apply to many situations. Functional abstraction!
- Well defined functions can help reduce redundancy in our code, which makes it more readable and adaptable
- Higher-order functions are functions that can take other functions as input, or produce other functions as output—they can help us further reduce redundancy in our code
- Functions have different behavior than control structures
- Functions can be nested within other functions
- Lambda expressions are a quick way to define simple functions within a single line