Environments

Announcements

Iteration Review

Spring 2023 Midterm 1, Question 3(a)

Definition: A positive integer n is a *repeating sequence* of positive integer m if n is written by repeating the digits of m one or more times. For example, 616161 is a repeating sequence of 61, but 61616 is not.

Hint: pow(10, 3) is 1000, and 654321 % pow(10, 3) is 321 (the last 3 digits).

Implement repeating which takes positive integers t and n. It returns whether n is a repeating sequence of some t-digit integer.

```
def repeating(t, n):
    """Return whether t digits repeat to form positive integer n.

    >>> repeating(1, 616161)
    False
    >>> repeating(2, 616161)  # repeats 61 (2 digits)
    True
```

```
616161

6161

61

0
```

An iterative appraoch: Repeatedly remove t digits from the end, and make sure that the last t digits never change.

Code structure: A while loop that checks the last t digits and returns **False** if they change.

Repeating (Spring 2023 Midterm 1 Q3a)

```
def repeating(t, n):
    """Return whether t digits repeat to form positive integer n.

    >>> repeating(1, 6161)
    False
    >>> repeating(2, 6161)  # repeats 61 (2 digits)
    True
    >>> repeating(3, 6161)
    False
    >>> repeating(4, 6161)  # repeats 6161 (4 digits)
    True
    >>> repeating(5, 6161)  # there are only 4 digits
    False
    """
    if pow(10, t-1) > n:  # make sure n has at least t digits
        return False
    rest = n
    while rest:
        if rest % pow(10, t) != n % pow(10, t):
            return False
        rest = rest // pow(10, t)
    return True
```

Go through digits, looking for something

The iterative process to implement "whether" functions is often to look for something that determines the function's output, and return when it's found.

Environments for Higher-Order Functions

Student advice from the Fall 2024 final survey:

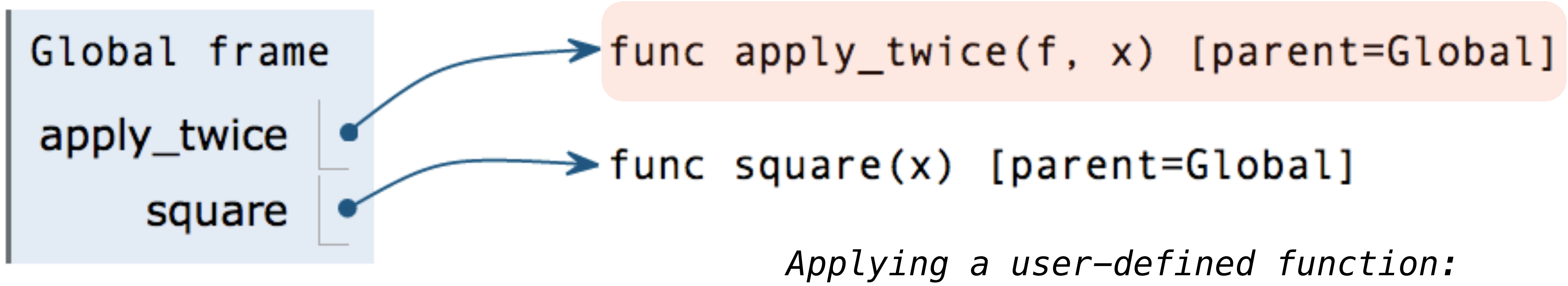
"ENVIRONMENT DIAGRAMS ARE EXTREMELY IMPORTANT! Taking this class with no prior Python experience and minimal overall programming experience, taking time to understand environment diagrams helped me fully understand step-by-step how my code is interpreted, and any areas where my code may be going wrong. This made coding more intuitive for me, as it helped me gain a understanding of the connections being made between my code and carried out functions."

Names can be Bound to Functional Arguments

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

def square(x):
    return x * x

result = apply_twice(square, 2)
```



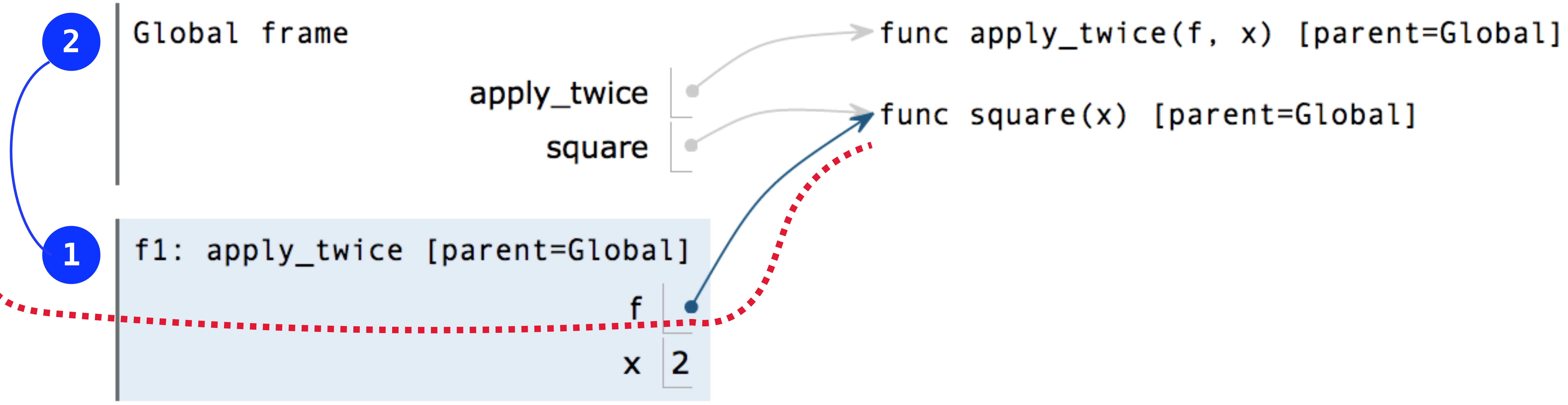
Applying a user-defined function:

- 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)
```

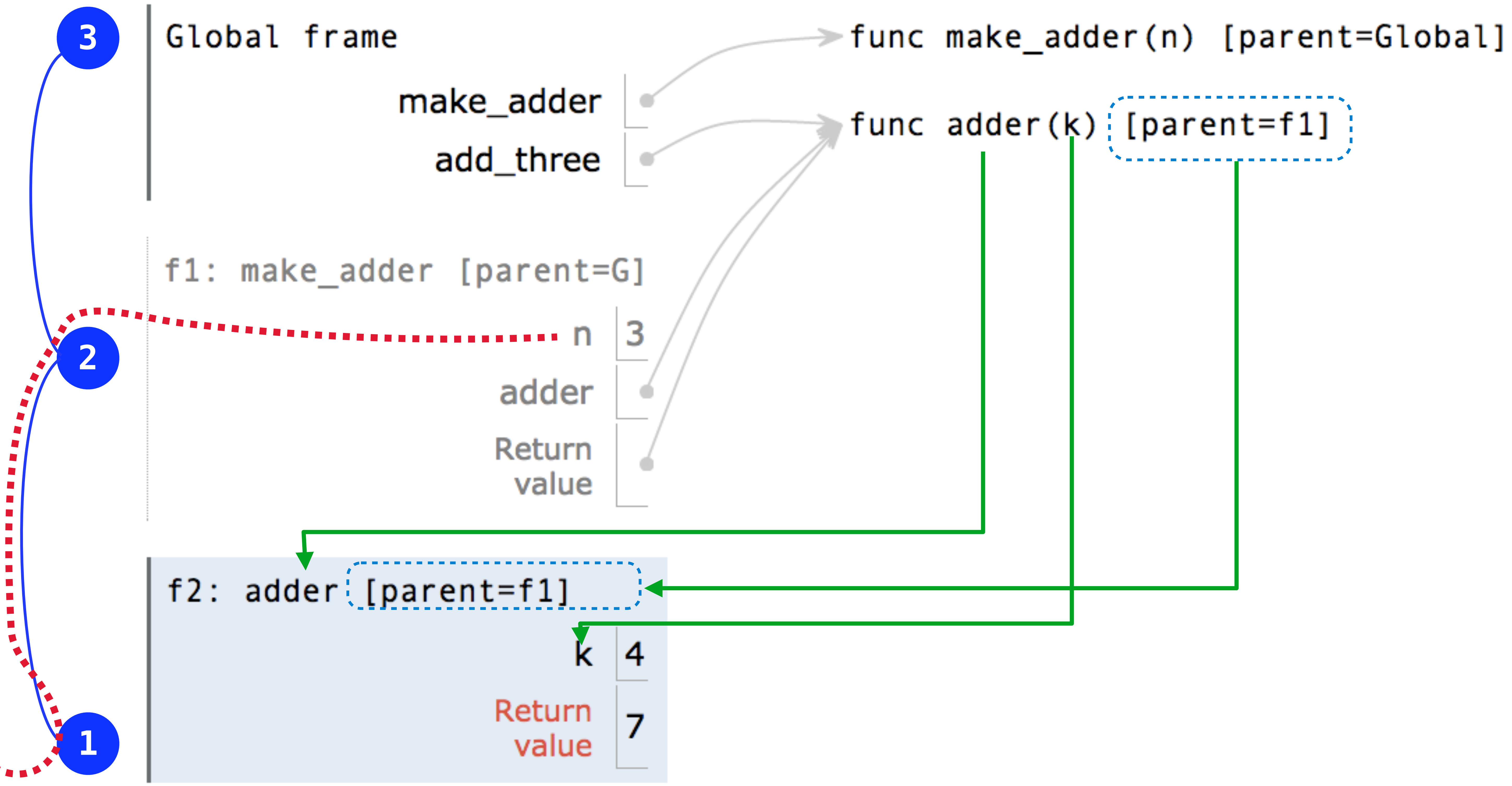


Environment Diagrams for Nested Def Statements

Nested def

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

add_three = make_adder(3)
add_three(4)
```



- 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

http://pythontutor.com/composingprograms.html#code=def%20make_adder%28n%29%3A%0A%20%20%20%20def%20adder%28k%29%3A%0A%20%20%20%20%20%20%20%20return%20k%20%2B%20n%0A%20%20%20%20return%20adder%0A%20%20%20%20%0Athree_more_than%20%3D%20make_adder%283%29%0Aresult%20%3D%20three_more_than%284%29&cumulative=false&curInstr=0&mode=display&origin=composingprograms.js&py=3&rawInputLstJSON=%5B%5D

How to Draw an Environment Diagram

When a function is defined:

Create a function value: func <name>(<formal parameters>) [parent=<label>]

Its parent is the current frame.

f1: make_adder   func adder(k) [parent=f1]

Bind <name> to the function value in the current frame

When a function is called:

1. Add a local frame, titled with the <name> of the function being called.
2. Copy the parent of the function to the local frame: [parent=<label>]
3. Bind the <formal parameters> to the arguments in the local frame.
4. Execute the body of the function in the environment that starts with the local frame.

Lambda Expressions

(Demo)

https://pythontutor.com/cp/composingprograms.html#code=def%20apply_twice%28f,%20x%29%3A%0A%20%20%20%20return%20f%28f%28x%29%29%0A%20%20%20%20%0Ax%20%3D%203%0Aresult%20%3D%20apply_twice%28lambda%20y%3A%20x%20*%20y,%202%29&cumulative=true&curInstr=0&mode=display&origin=composingprograms.js&py=3&rawInputLstJSON=%5B%5D

https://pythontutor.com/cp/composingprograms.html#code=bear%20%3D%20-1%0Aoski%20%3D%20lambda%20print%3A%20print%28bear%29%0Abear%20%3D%20-2%0Aprint%28oski%28abs%29%29&cumulative=true&curInstr=0&mode=display&origin=composingprograms.js&py=3&rawInputLstJSON=%5B%5D

Break: 5 minutes

Zero-Argument Functions

(Demo)

Currying

Function Currying

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

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

There's a general relationship between these functions

(Demo)

Curying Conceptualization: Transform a single multiple-argument function into multiple single-argument functions (using the power of higher-order functions!)

Why Curry?:
1. Fix some arguments now, pass the rest of the arguments later. As seen in demo.

2. Create specialized instances of general functions. As seen in demo and next example...

Example: Reverse

The square function can be defined in terms of the built-in pow function:

```
def square(x):
    """Square x.

    >>> square(3)
    9
    """
    return pow(x, 2)
```

```
def cube(x):
    """Cube x.

    >>> cube(3)
    27
    """
    return pow(x, 3)
```

Define square and cube in one line without using lambda or ** (using curry and reverse).

```
def reverse(f):
    return lambda x, y: f(y, x)
```

```
def curry(f):
    def g(x):
        def h(y):
            return f(x, y)
        return h
    return g
```

```
square = curry(reverse(pow))(2)
cube   = curry(reverse(pow))(3)
```

Lamb Curry

(Currying with Lambdas)

(Demo)