Higher-Order Functions
Announcements
Office Hours: You Should Go!

You are not alone!

https://cs61a.org/office-hours/
Example: Prime Factorization
Prime Factorization

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

... 

$8 = 2 \times 2 \times 2$  
$9 = 3 \times 3$  
$10 = 2 \times 5$  
$11 = 11$  
$12 = 2 \times 2 \times 3$  
...

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.

\[ 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987 \]
Go Bears!
Designing Functions
Describing Functions

A function's *domain* is the set of all inputs it might possibly take as arguments.

A function's *range* is the set of output values it might possibly return.

A pure function's *behavior* is the relationship it creates between input and output.

```python
def square(x):
    '''Return X * X.'""

x is a number

square returns a non-negative real number

square returns the square of x
```
A Guide to Designing Function

Give each function exactly one job, but make it apply to many related situations

```python
>>> round(1.23)
1
>>> round(1.23, 1)
1.2
>>> round(1.23, 0)
1
>>> round(1.23, 5)
1.23
```

Don’t repeat yourself (DRY): Implement a process just once, but execute it many times

(Demo)
Generalization
Generalizing Patterns with Arguments

Regular geometric shapes relate length and area.

Shape:

Area:

Finding common structure allows for shared implementation

(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."
    total, k = 0, 1
    while k <= n:
        total, k = total + term(k), k + 1
    return total

>>> summation(5, cube)
225
"""
```

- `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`
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

```
def make_adder(n):
    """Return a function that takes one argument k and returns k + n."
    def adder(k):
        return k + n
    return adder
```

The name add_three is bound to a function

```
>>> add_three = make_adder(3)
>>> add_three(4)
7
```

A def statement within another def statement

Can refer to names in the enclosing function

```
def identity(k):
    return k

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

def summation(n, term):
    """Sum the first n terms of a sequence."
    total, k = 0, 1
    while k <= n:
        total, k = total + term(k), k + 1
    return total

def pi_term(k):
    return 8 / (k * 4) / (k * 4 - 3)

# Local function definitions; returning functions

def make_adder(n):
    """Return a function that takes one argument k and returns k + n."
    def adder(k):
        return k + n
    return adder

def compose1(f, g):
    """Return a function that composes f and g.
    f, g functions of a single argument"
    def h(x):
        return f(g(x))
    return h
```
Call Expressions as Operator Expressions

An expression that evaluates to a function

Operator

An expression that evaluates to its argument

Operand

func make adder(n)

func make adder(1)

func adder(k)

make_adder(1) ( 2 )

3

1

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

func adder(k)

2