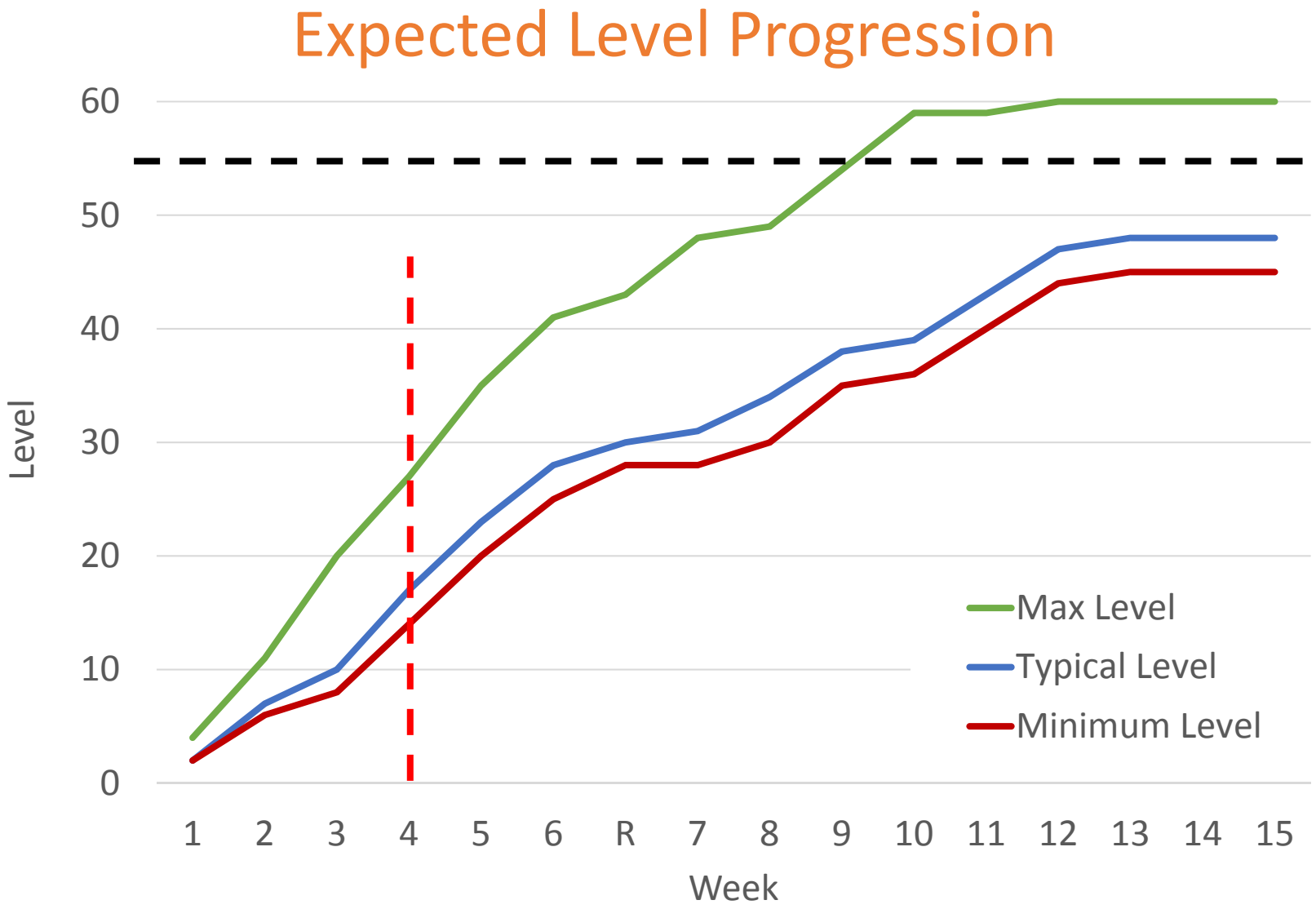
CS1010S Programming Methodology

Lecture 4

Higher-order Functions

5 Sep 2018


Expected Level Progression
60
50
40
30
20
10
0
Level
1
2
3
4
5
6
R
7
8
9
10
11
12
13
14
15
Week
Max Level
Typical Level
Minimum Level

Don't need to do
EVERY Side Quest

Just do ALL the main missions

Remedial Lessons

Time: Wed 6:30-8:30pm

Time: Thu 6:30-8:30pm

Venue: SoC COM1-209

Today's Agenda

- Count Change
  - Recursion
  - Order of Growth

- Higher-order Functions
  - Generalizing Common Patterns
  - Functions as arguments

Importing Modules

- `import X`
  - use `X.name` to refer to objects in `X`

```
import math
print(math.pi)
```

- `from X import a, b, c`
  - create references to specified objects
  - can now use `a` and `b` and `c` in your program

```
from math import pi
print(pi)
```

- `from X import *`
  - create references to all public objects in `X`
  - can use plain name

```
from math import *
print(pi)
```

Recap: Order of Growth

- Provide a useful indication of how we may expect the resource requirement of the algorithm to change as we change the size of the problem.

- Time requirement
  - Total number of operations (statements)

- Space requirement
  - Maximum usage of computer memory.

Time Requirement

- Iteration
  - Count the number of statements that will be executed
- Recursion
  - Count the number of function calls
  - How many operations per function call?
- The result of calculation is a function $f(n)$
  - Big-O notation: $O(f(n))$ shows order of growth
    - Keep dominant term only; ignore coefficient

Space Requirement

- Iteration
  - The number of variables (data structures) used
- Recursion
  - The maximum number of co-existing function calls
    - e.g. the depth of a binary tree
  - Any data structure used?
- Express the result of analysis in terms of problem size n, i.e. $\mathrm{O}(f(n))$

Rules of Thumb

- A loop of n iterations will lead to O(n) time complexity.

- When the domain of interest is reduced by a fraction (e.g. by 1/2, 1/3, or 1/10, etc.) in each iteration, it will lead to O($log\ n$) time complexity.

- For a recursive function, each call is usually O(1). So
  - if n calls are made – O(n)
  - if 2^n calls are made – O(2^n)

Recall: Recursion

- Relate given problem with sub-problem(s).
  - There is a straightforward, non-recursive solution for the simplest sub-problems.
  - Build solutions to more complex problems on top of that.

```
// general format
if base case(s) satisfied:
    return solution
else:
    make recursive call(s) to simpler cases
```


Counting
Change

Problem

Make change for $1, using coins

50¢, 20¢, 10¢, 5¢, 1¢

(assuming unlimited number of coins)

How many ways can we make the change?

e.g. 50¢ + 50¢

50¢ + 20¢ + 20¢ + 10¢

20¢ + 20 ¢ + 20¢ + 20¢ + 20¢

etc.

Counting Change

How many ways to do it?

Formulate the problem

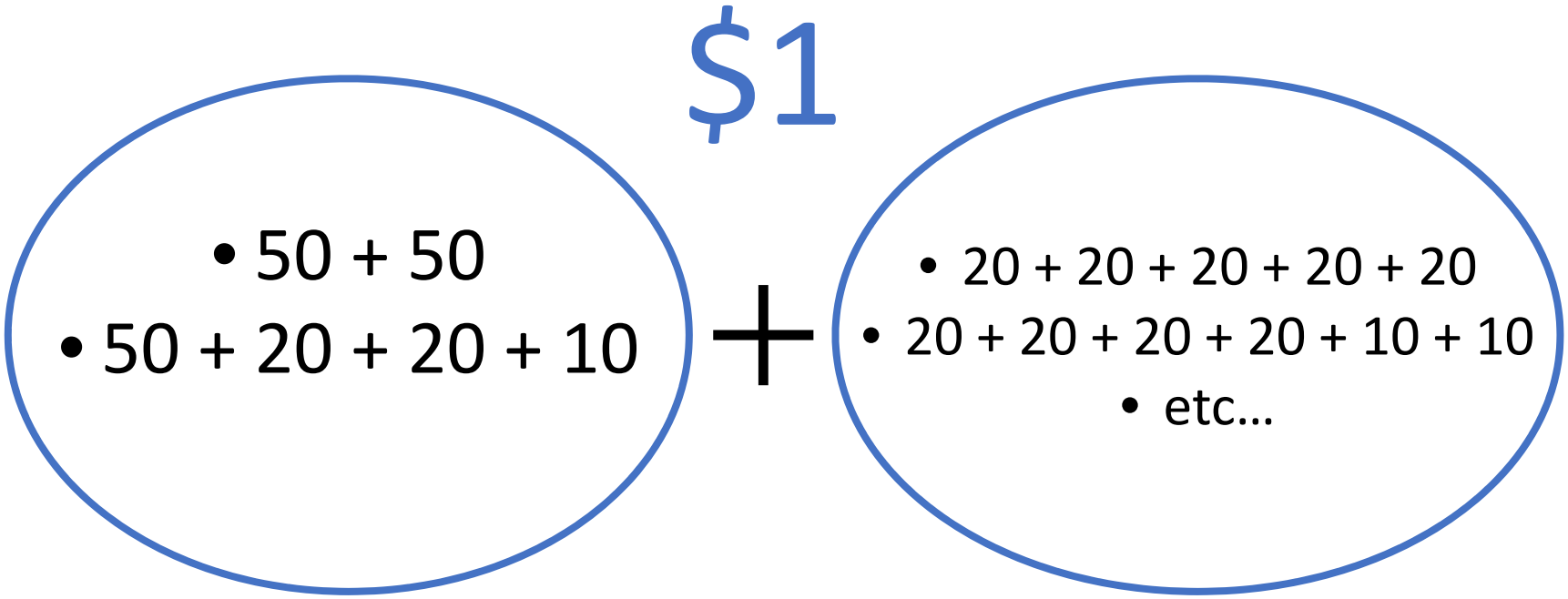
- amount: amt
  - The amount in cents, e.g. \$1

- types-of-coins: $\{d_1, d_2, \dots, d_k\}$
  - e.g. $\{50¢, 20¢, 10¢, 5¢, 1¢\}$

How many ways can we make the change for *amt*?

Recursive Idea

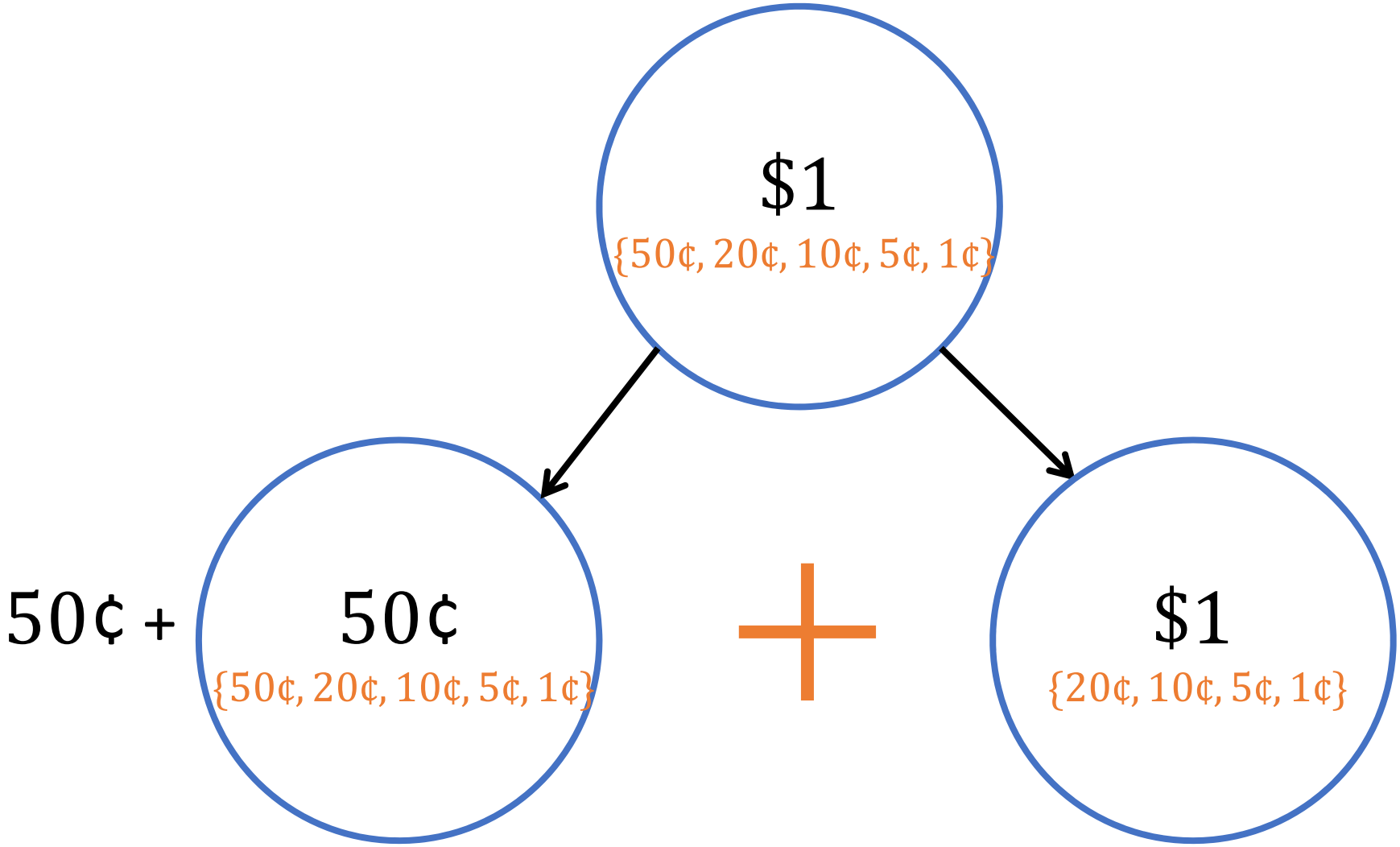
Observation: we can divide all possible combinations into two groups:



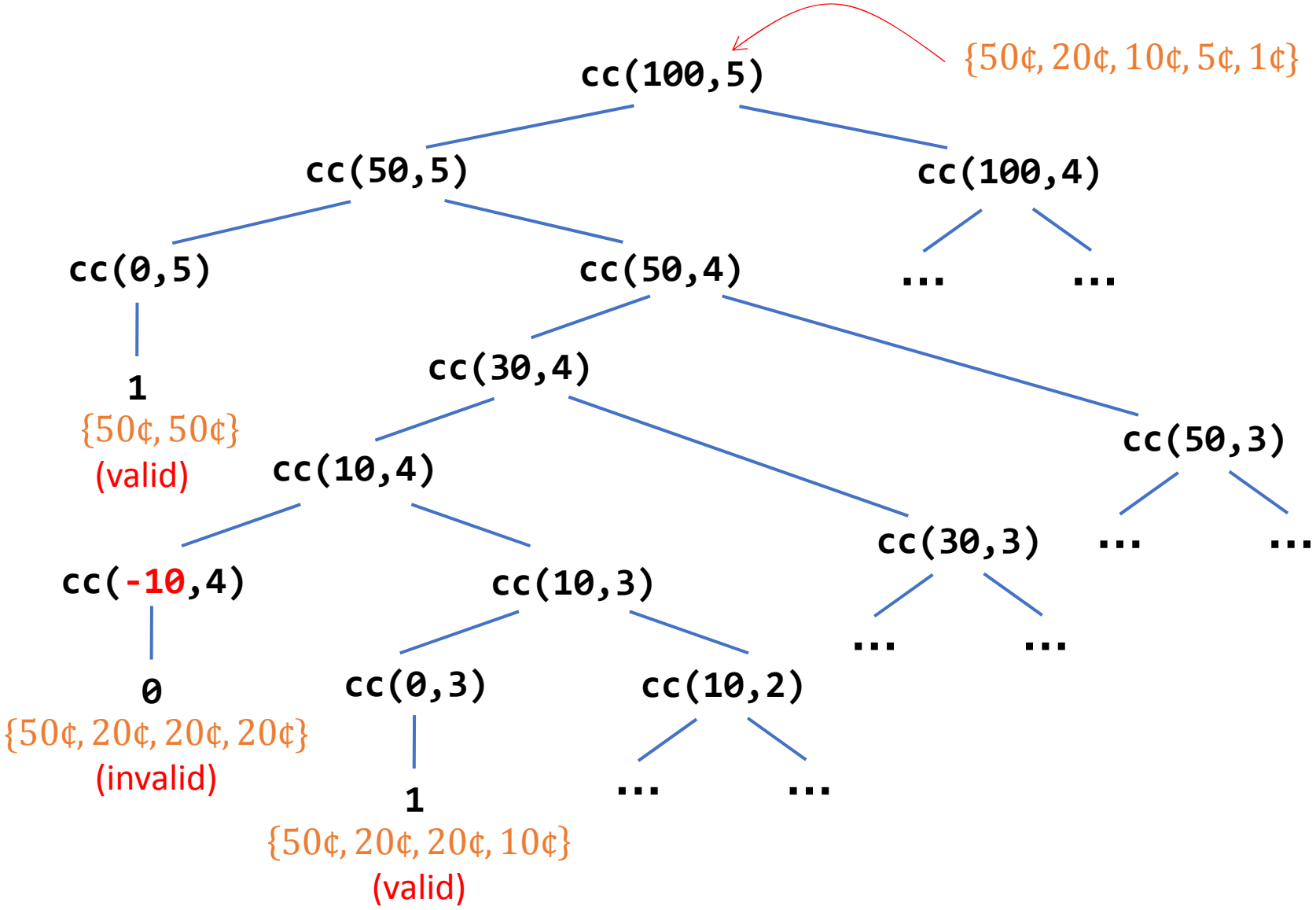
At least one 50 cent coin

No 50 cent coins

Different Combinations



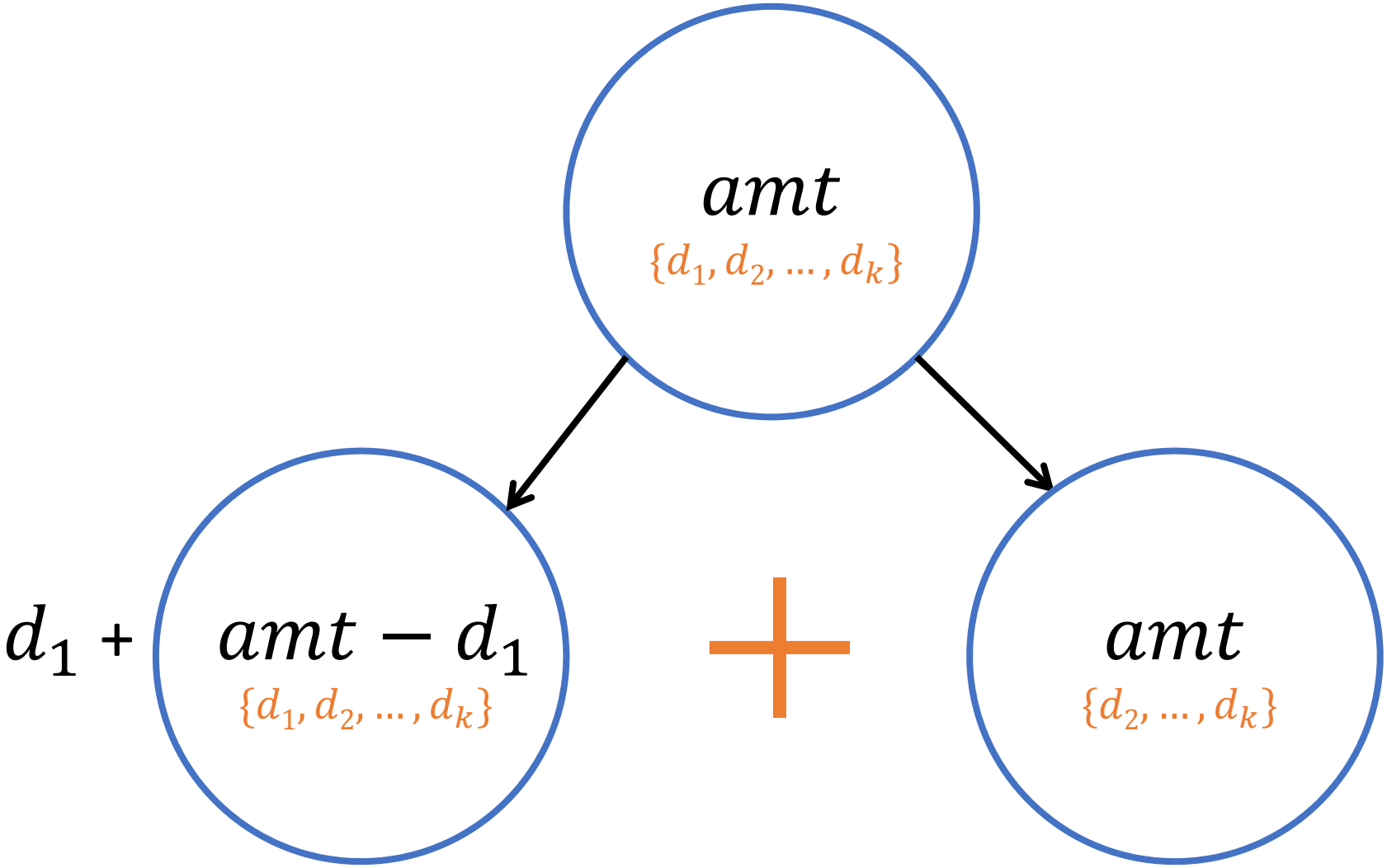
Binary Trace Tree



Base Cases

- if $amt = 0$ — 1 valid way to make change.
- if $amt < 0$ — no way to make change
- if coins = {} — no way to make change.

In general



Recursive Idea

Given a particular set of coins

$$\{d_1, d_2, \ldots, d_n\}$$

the change for an amount amt can be divided into two disjoint and complimentary sets:

1. Those that has least one d_1 coin
2. Those that do not use any d_1 coins

Python function

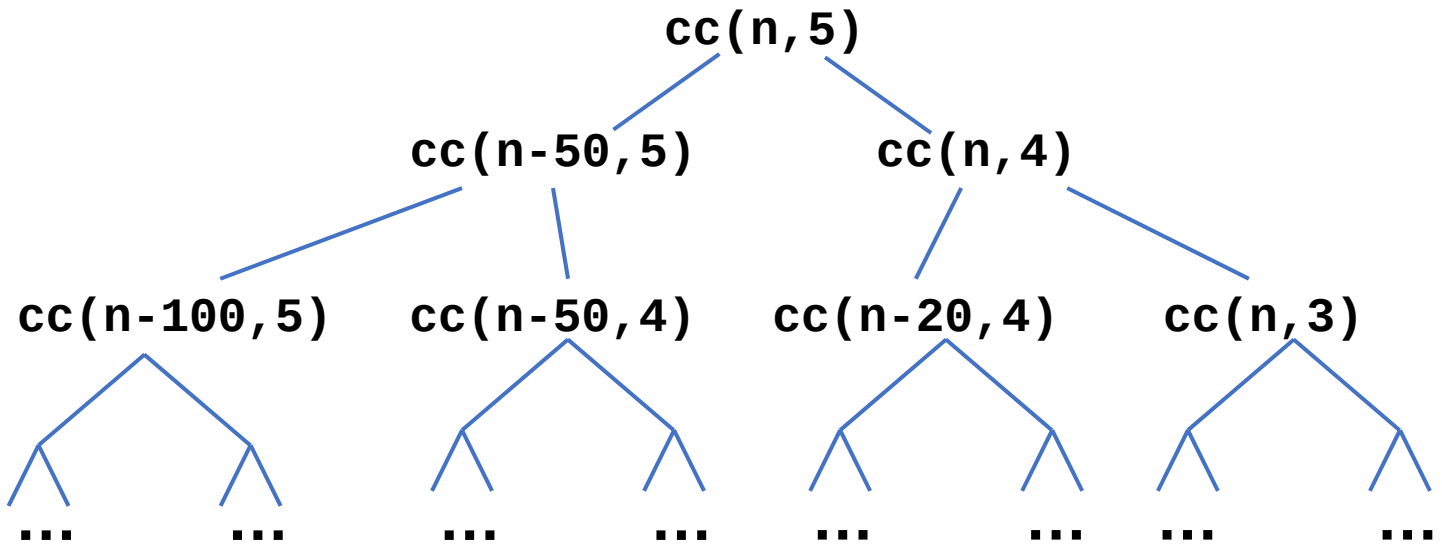
```
def cc(amt, kinds_of_coins):
    if amt == 0:
        return 1
    elif amt < 0 or kinds_of_coins == 0:
        return 0
    else:
        first = get_first_denomination(kinds_of_coins)
        return cc(amt-first, kinds_of_coins) + \
               cc(amt, kinds_of_coins-1)

# return the first available denomination
def get_first_denomination(kinds_of_coins):
    # <left as an exercise>
```

Ung 1 coin of the first kind

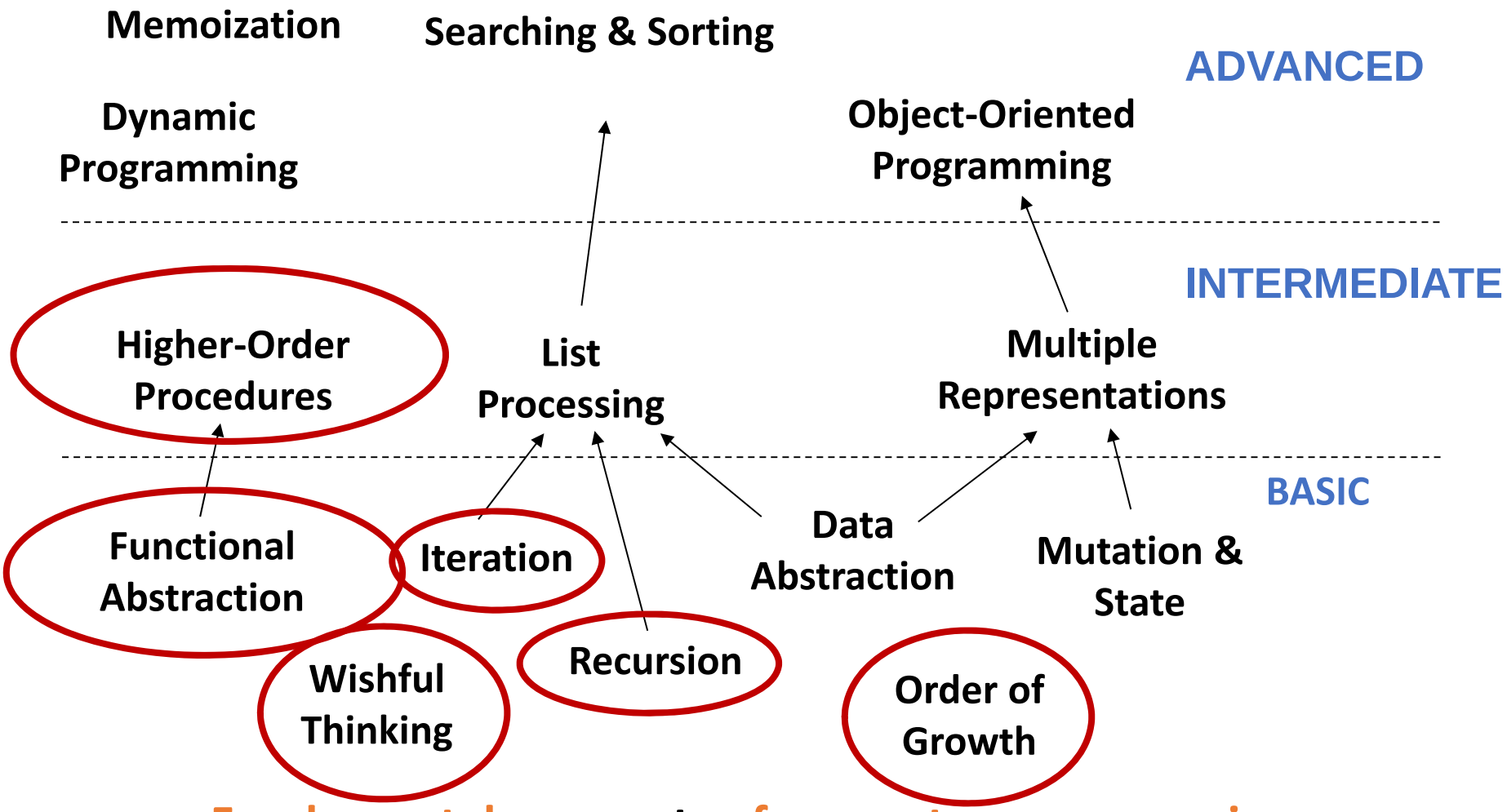
Without using first kind of coin

Order of Growth?

Binary trace tree with depth n…



CS1010S Road Map

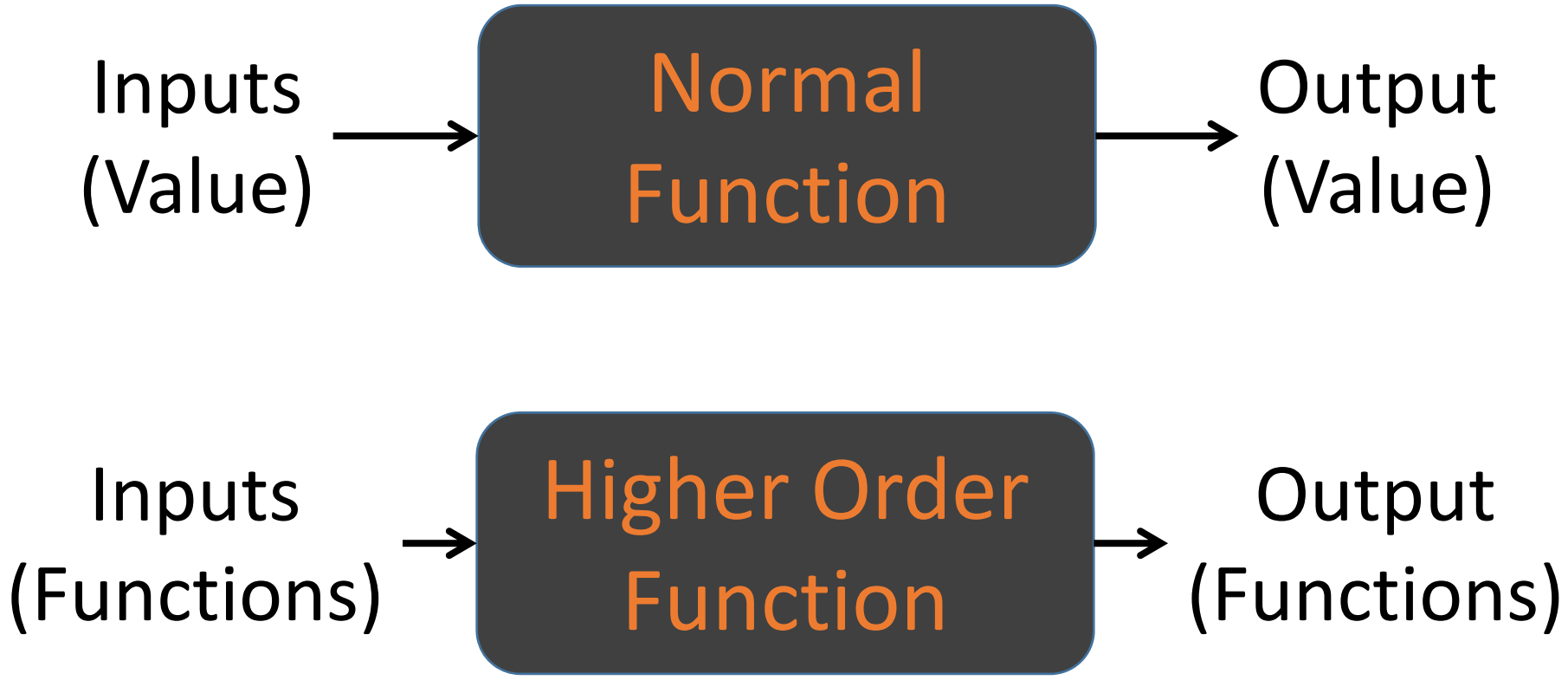


Fundamental **concepts** of computer programming

What is

Higher Order Function?

What is High Order Function?



Example

```
def f(x, y): # parameter x is a function!
    return x(y)

f(abs, -2) # return 2

f(len, 'abcd') # return length of string: 4

from math import sqrt
f(sqrt, 4) # return 2.0
```

WHY

Higher Order Function?

Consider the following recursive function to sum all integers in the range *a* to *b*

```
def sum_integers(a, b):
  if a > b:
    return 0
  else:
    return a + sum_integers(a+1, b)
```

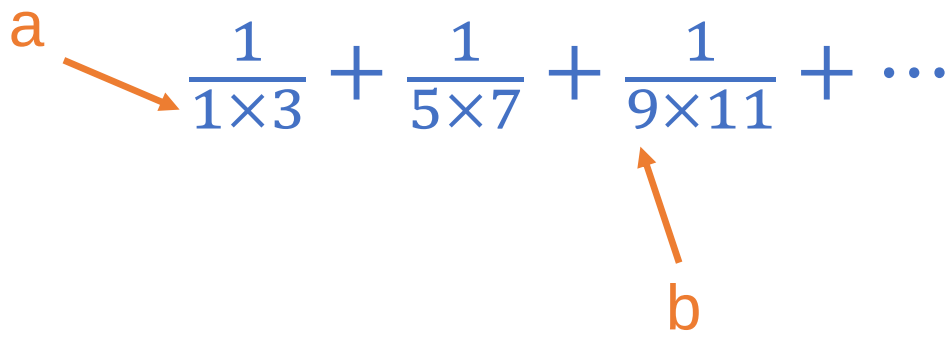
$$\sum_{n=a}^{b} n$$

Now suppose we want to sum the cubes of integers in the range *a* to *b*

$$\sum_{n=a}^{b} n^3$$

```
def sum_cubes(a, b):
  if a > b:
    return 0
  else:
    return cube(a) + sum_cubes(a+1, b)

def cube(n):
  return n*n*n
```

Finally, we want to sum the following series (which converges very slowly to $\pi/8$):




```
def pi_sum(a, b):
  if a > b:
    return 0
  else:
    return 1/(a*(a+2)) + pi_sum(a+4, b)
```

$$\sum_{a=1,a+4}^{b} \frac{1}{a*(a+2)}$$

Abstracting Common Patterns

- All three functions are very similar.

```
def sum_integers(a, b):
    if a > b:
        return 0
    else:
        return a +
               sum_integers(a + 1, b)
```

```
def pi_sum(a, b):
    if a > b:
        return 0
    else:
        return 1/(a*(a + 2)) +
               pi_sum(a + 4, b)
```

```
def sum_cubes(a, b):
    if a > b:
        return 0
    else:
        return cube(a) +
               sum_cubes(a + 1,  b)
```

$$\sum_{n=a}^{b} f(n)$$

Abstracting Common Patterns

- All three functions are very similar.

```
def sum_integers(a, b):
  if a > b:
    return 0
  else:
    return a +
           sum_integers(a + 1, b)
```

```
def pi_sum(a, b):
  if a > b:
    return 0
  else:
    return 1/(a*(a + 2)) +
           pi_sum(a + 4, b)
```

```
def sum_cubes(a, b):
  if a > b:
    return 0
  else:
    return cube(a) +
           sum_cubes(a + 1,  b)
```

```
def sum(a, b):
  if a > b:
    return 0
  else:
    return <term>(a) +
           sum(<next>(a), b)
```

Functions as Arguments

```
def sum(term, a, next, b):
    if a > b:
        return 0
    else:
        return term(a) +
               sum(term, next(a), next, b)
```

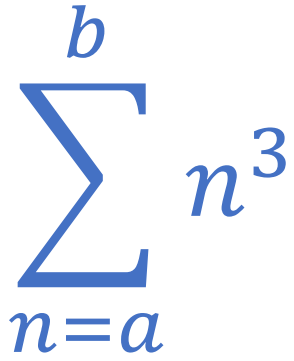
- Note that **`term`** and **`next`** are functions given by user (caller function).

Re-writing sum_integers()

```
def sum(term, a, next, b):
  if a > b:
    return 0
  else:
    return term(a) + sum(term, next(a), next, b)

def sum_integers(a, b):
    return sum(identity, a, inc, b)
def identity(x):
    return x
def inc(n):
    return n+1
```

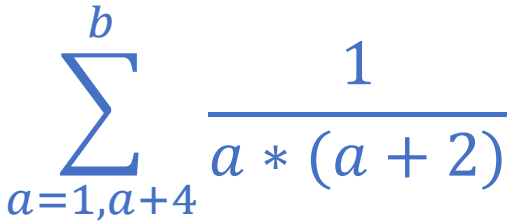
Re-writing sum_cubes()

```
def sum(term, a, next, b):
  if a > b:
    return 0
  else:
    return term(a) + sum(term, next(a), next, b)

def sum_cubes(a, b):
    return sum( cube ,   a, inc, b)
def cube(x):
    return x*x*x
def inc(n):
    return n+1
```



Re-writing pi_sum()

```
def sum(term, a, next, b):
    if a > b:
        return 0
    else:
        return term(a) + sum(term, next(a), next, b)

def pi_sum(a, b):
    return sum(fraction, a, inc, b)
def fraction(x):
    return 1/(x*(x+2))
def inc(n):
    return n+4
```



Re-writing pi_sum() again

```
def sum(term, a, next, b):
  if a > b:
    return 0
  else:
    return term(a) + sum(term, next(a), next, b)

def pi_sum(a, b):
    def fraction(x): # inner function
        return 1/(x*(x+2))
    def inc(n):
        return n+4
    return sum(fraction, a, inc, b)
```

$$\sum_{a=1,a+4}^{b} \frac{1}{a*(a+2)}$$

Anonymous Functions

- Alternatively,

```
def pi_sum(a, b):
    return sum(lambda x: 1/(x*(x+2)),
               a,
               lambda n: n+4,
               b)
```

```
def fraction(x):
    return 1/(x*(x+2))
```

```
def inc(n):
    return n+4
```

anonymous functions

Key idea

- `sum()` is a higher-order function.
  - captures a common pattern.

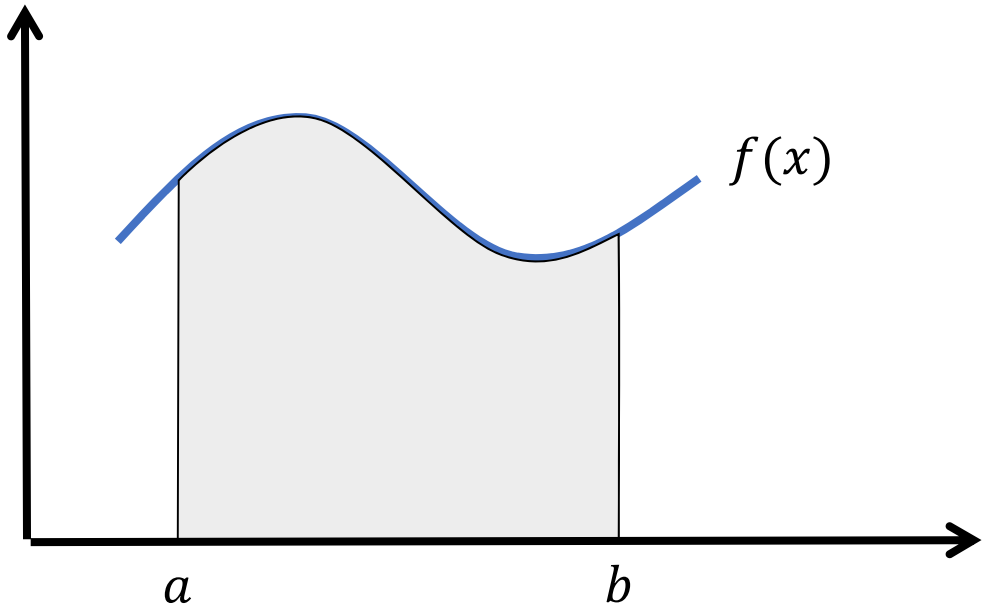
- All other functions (`sum_integers`, `sum_cubes`, `pi_sum`) are specific cases of `sum`.

Higher-order Functions

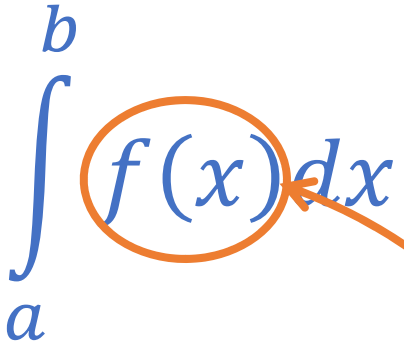
generalize common patterns by taking functions as input

Example: Integration

$$\int_a^b f(x)dx = \text{area under curve}$$



Example: Integration

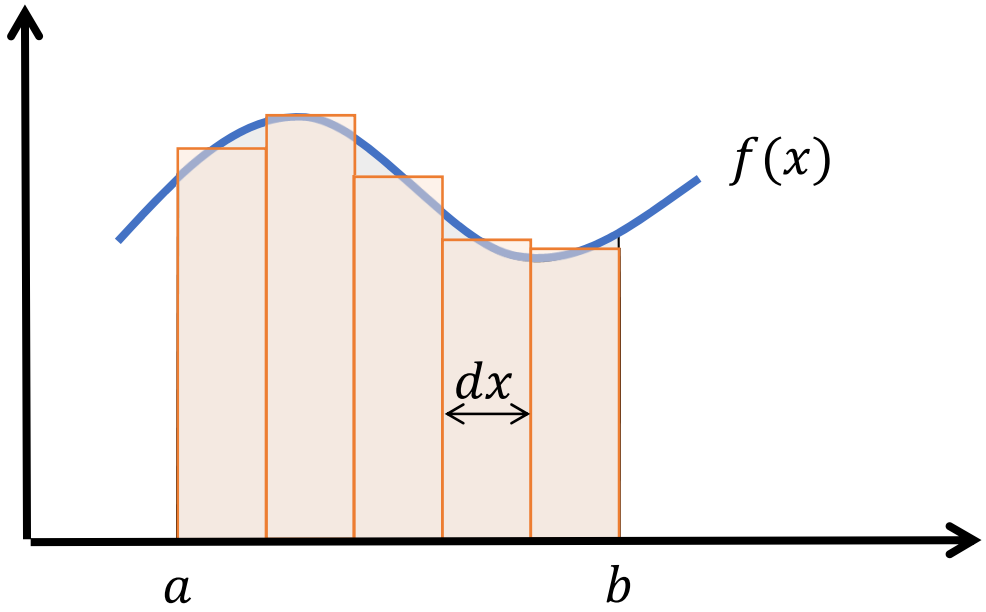


Integration is a higher order function!

- Inputs contain a function
- Output is a number (area)

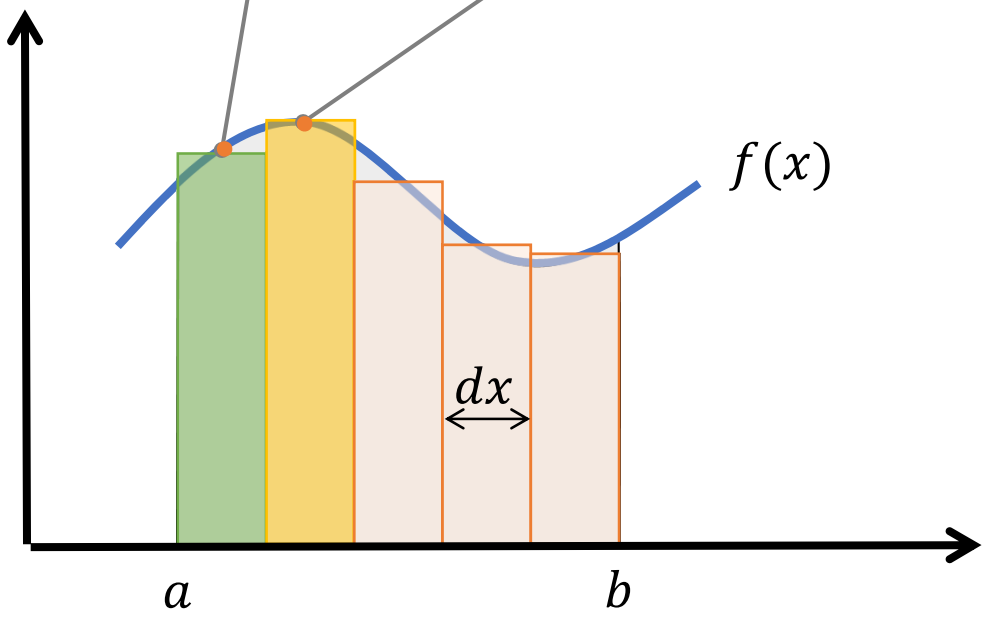
Example: Integration

$$\int_a^b f(x)dx \approx \text{sum area of rectangles}$$



Example: Integration

$$\int_a^b f(x)dx \approx \left(f\left(a + \frac{dx}{2}\right)dx + f\left(a + dx + \frac{dx}{2}\right)dx + \cdots \right)$$



$$\int_a^b f(x)dx \approx \left\{f\left(a+\frac{dx}{2}\right)+f\left(a+\frac{dx}{2}+dx\right)+f\left(a+\frac{dx}{2}+2dx\right)+\cdots\right\}dx$$

```
def sum(term, a, next, b):
  if a > b:
    return 0
  else:
    return term(a) + sum(term, next(a), next, b)

def integral(f, a, b):
    dx = 0.01
    def add_dx(x): # inner function
        return x + dx
    return sum(f, a+(dx/2), add_dx, b) * dx

>>> integral(lambda x: x**3, 0, 1) # in math: 0.25
0.24998750000000042
```

Higher-order Functions

Functions as output

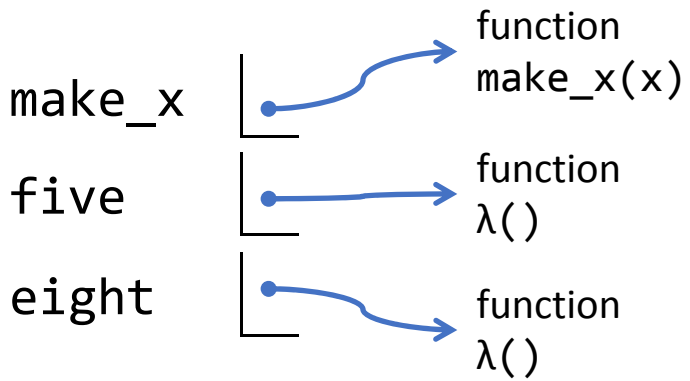
Example

```
def make_x(x):
    def f(): # inner function
        return x
    return f # return a function!

>>> five = make_x(5)
>>> five
<function make_x.<locals>.f at 0x0000000003BE0598>
>>> five()
5
>>> eight = make_x(8)
>>> eight()
8
```

Import to understand what is a function's return type: value or function?

Re-writing make_x()

```
'''
def make_x(x):
    def f(): # inner function
        return x
    return f # return a function!
'''

def make_x(x):
    return lambda : x
five = make_x(5)
five()
eight = make_x(8)
eight()
```



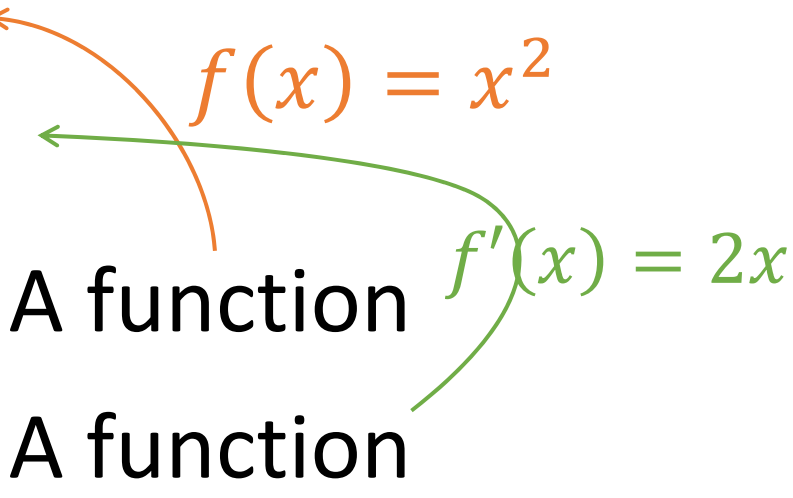
Example: Derivative

$$\frac{dy}{dx} = D(y)(x)$$

Example:

$$y = x^2$$

$$\frac{dy}{dx} = 2x$$



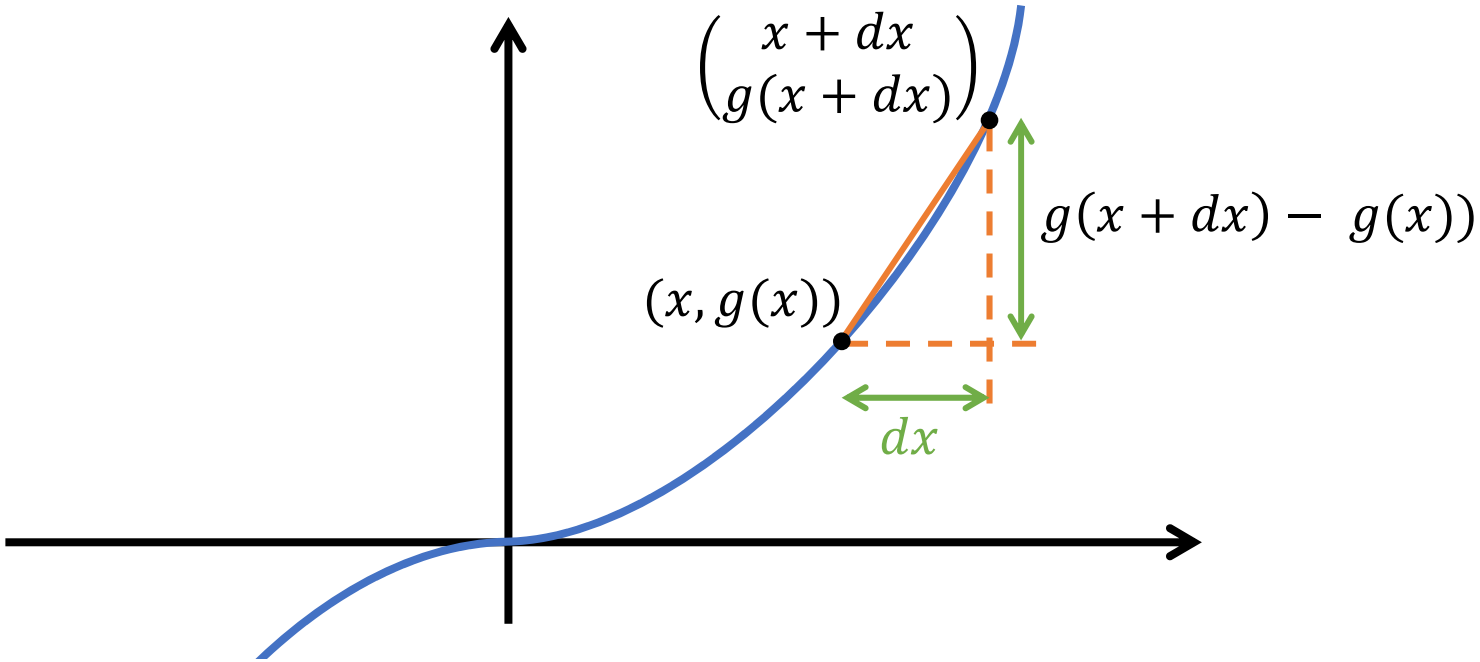
What is the input?

What is the output?

How is Derivative Computed?

In math, the derivative of $g(x)$ is

$$D(g)(x) = \lim_{dx \to 0} \frac{g(x + dx) - g(x)}{dx}$$



Derivative Function

$$D(g)(x) = \lim_{dx \to 0} \frac{g(x + dx) - g(x)}{dx}$$

Derivative takes an input function g and returns g', computed by the above formula.

```
def deriv(g): # version 1
    dx = 0.00001
    def f(x): # inner function
        return (g(x+dx)-g(x)) / dx
    return f
```

Derivative Function

$$D(g)(x) = \lim_{dx \to 0} \frac{g(x + dx) - g(x)}{dx}$$

Derivative takes an input function g and returns g', computed by the above formula.

```
def deriv(g): # version 2, lambda
    dx = 0.00001
    return lambda x: (g(x+dx)-g(x)) / dx
```

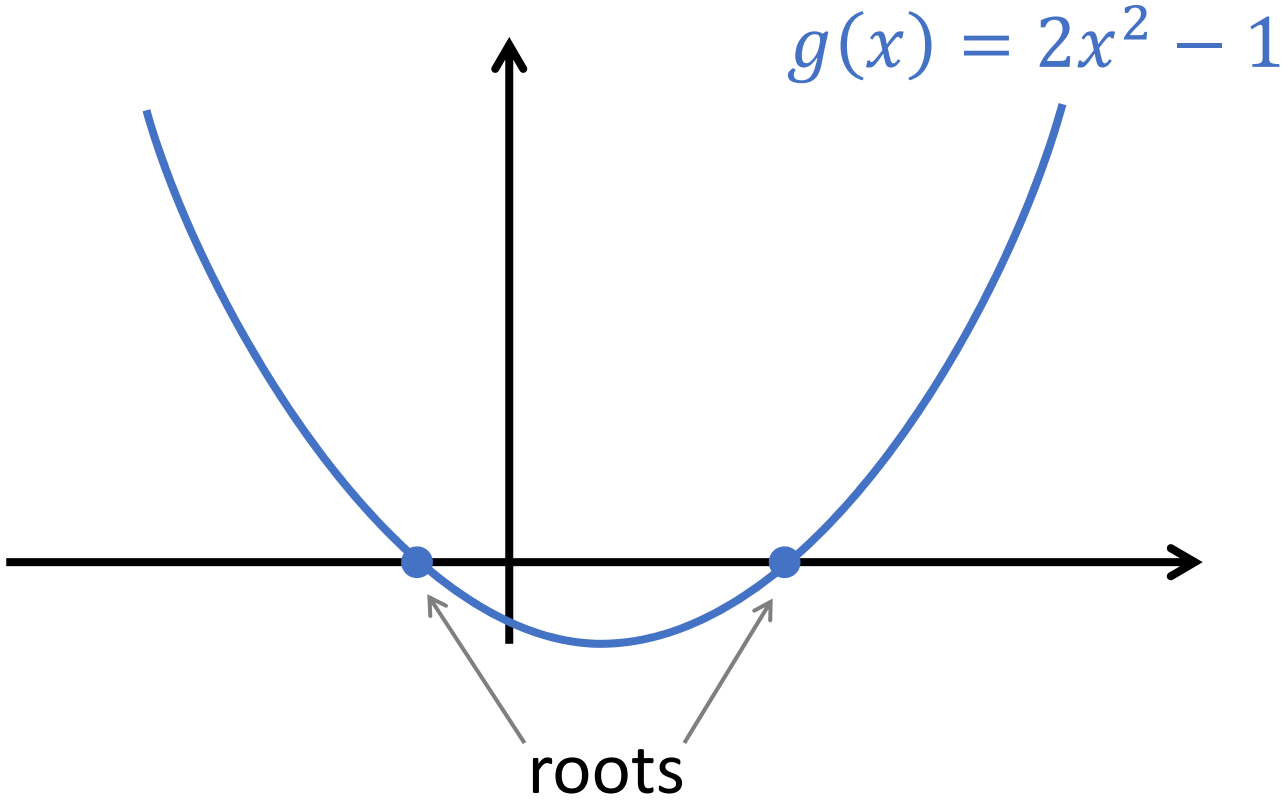
Computing Derivative

```
def deriv(g): # compute derivative of g(x)
    dx = 0.00001
    return lambda x: (g(x+dx)-g(x)) / dx

cube = lambda x: x*x*x # g(x) = x^3
d_cube = deriv(cube)
d_cube(5)  # output: 75.00014999664018

from math import sin, pi
cos = deriv(sin) # g(x) = sin(x)
cos(pi/4) # output: 0.7071032456451575
cos(pi/2) # output: -5.000000413701855e-06
          # i.e., -5.0000 × 10-6 ≈ 0
```

Another example

Example: Newton's method

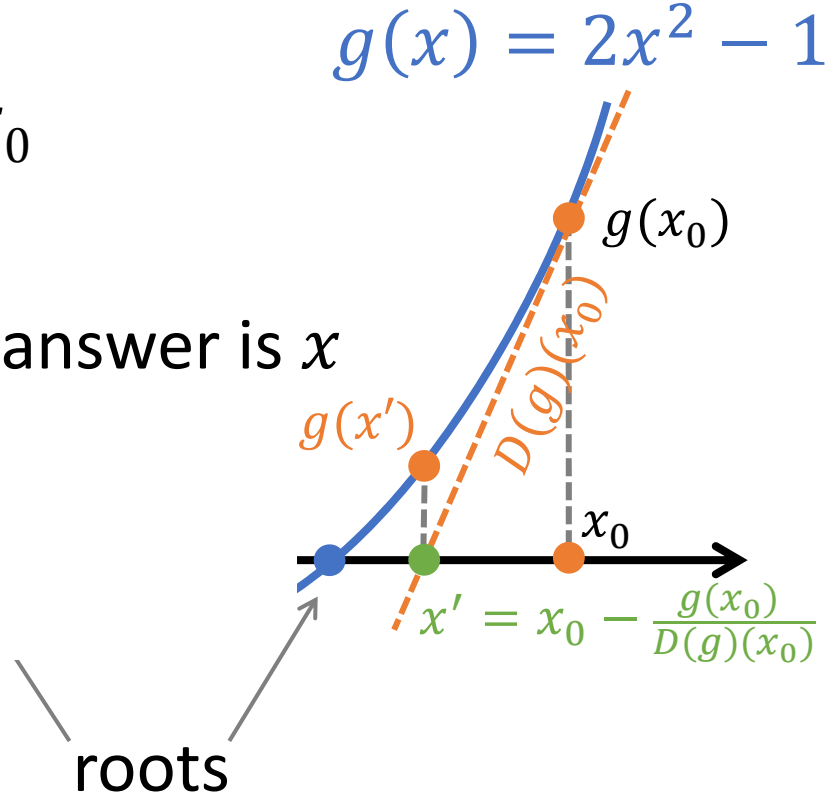
To compute root of function $g(x)$, i.e. find x such that $g(x) = 0$



Example: Newton's method

To compute root of function $g(x)$, i.e. find x such that $g(x) = 0$

1. Start with initial guess x_0
2. $x \leftarrow x_0$
3. If $g(x) \approx 0$ then stop: answer is x
4. $x \leftarrow x - \frac{g(x)}{D(g)(x)}$
5. Go to step 3



```
def newtons_method(g, first_guess):
    dg = deriv(g)

    def improve(x): # step 4
        return x - g(x)/dg(x)

    def close_to_zero(v): # step 3
        tolerance = 0.0001
        return abs(v) < tolerance

    def attempt(x):
        if close_to_zero(g(x)):
            return x
        else:
            return attempt(improve(x)) # step 5

    return attempt(first_guess) # return root
```

1. Start with initial guess x_0
2. $x \leftarrow x_0$
3. If $g(x) \approx 0$ then stop: answer is x
4. $x \leftarrow x - \frac{g(x)}{D(g)(x)}$
5. Go to step 3

Computing Square Root

- Square root of a is the number x such that:

$$x^2 = a$$

- Use Newton's method to find x:

$$g(x) \equiv x^2 - a = 0$$

Use Newton's method to find x:

$$g(x) \equiv x^2 - a = 0$$

```
def sqrt(a):
    return newtons_method(lambda x: x*x-a, a/2)
                          #initial guess is half of a

sqrt(9) # output: 3.0000153774963274

sqrt(2) # output: 1.4142156951657834
```

Higher Order Functions

Manipulate Other Functions

Another Example: `fold()`

Compute, $f(1) \oplus f(2) \oplus \cdots \oplus f(n)$ for some function f, by applying binary operator $\oplus$ $n-1$ times.

Examples: $1 + 2 + \cdots + n$

$1^2 * 2^2 * \cdots * n^2$

$\frac{1}{1\times3} + \frac{1}{5\times7} + \frac{1}{9\times11} + \cdots + \frac{1}{n\times(n+2)}$

```
def fold(op, f, n):
    if n == 1:
        return f(1)
    else:
        return op(fold(op, f, n-1), f(n))
```

Computing a^n with `fold()`

$$a^n = \overbrace{a \times a \times \cdots \times a}^{n}$$

```
def fold(op, f, n):
    if n == 1:
        return f(1)
    else:
        return op(fold(op, f, n-1), f(n))

def exp(a, n):
    op = lambda x, y: x * y
    f = lambda n: a # f(n) -> a
    return fold(op, f, n)
```

Computing a^n with `fold()`

$$a^n = \overbrace{a \times a \times \cdots \times a}^{n}$$

```
def fold(op, f, n):
    if n == 1:
        return f(1)
    else:
        return op(fold(op, f, n-1), f(n))

def exp(a, n):
    return fold(lambda x, y: x * y, #op
                lambda n: a, # f
                n)
```

Summing Digits w/ fold()

- Write a function sum_of_digits(num) that returns the sum of all the digits of num.
  - How do we express this as fold()?

```
def fold(op, f, n):
    if n == 1:
        return f(1)
    else:
        return op(fold(op, f, n-1), f(n))

def sum_of_digits(num): # 1234 -> 1+2+3+4 = 10
    return fold( lambda x, y: x + y,
                 lambda k: kth_digit(k, num),
                 count_digits(num))
```

```
def product_of_digits(num):
    return fold(lambda x, y: x * y,
                lambda k: kth_digit(k, num),
                count_digits(num))


def sum_of_square_of_digits(num):
    return fold(lambda x, y: x + y
                lambda k: kth_digit(k, num) ** 2,
                count_digits(num))

from math import sqrt
def sum_of_sqrt_of_digits(num):
    return fold(lambda x, y: x + y,
                lambda k: sqrt(kth_digit(k, num)),
                count_digits(num))
```

Last example!

Recap: Sum of Integers

```
def sum(term, a, next, b):
  if a > b:
    return 0
  else:
    return term(a) + sum(term, next(a), next, b)


def sum_integers(a, b):
  return sum(lambda x: x,
             a,
             lambda x: x+1,
             b)
```

$$\sum_{n=a}^{b} n$$

Product of Integers

```
def product(term, a, next, b):
  if a > b:
    return 1
  else:
    return term(a) * product(term, next(a), next, b)


def product_integers(a, b):
  return product(lambda x: x,
                 a,
                 lambda x: x+1,
                 b)
```

$$\prod_{n=a}^{b} n$$

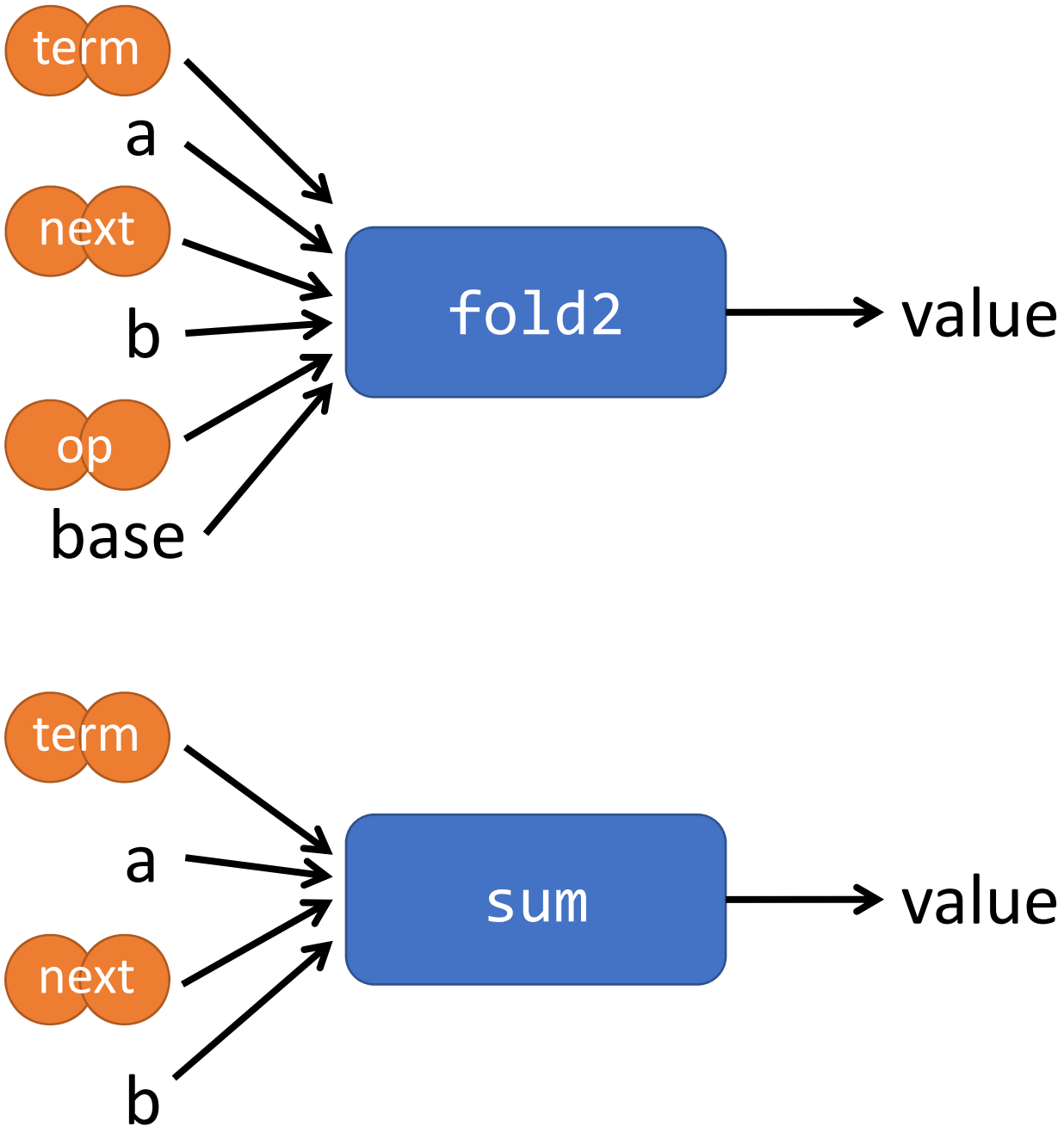
Comparison

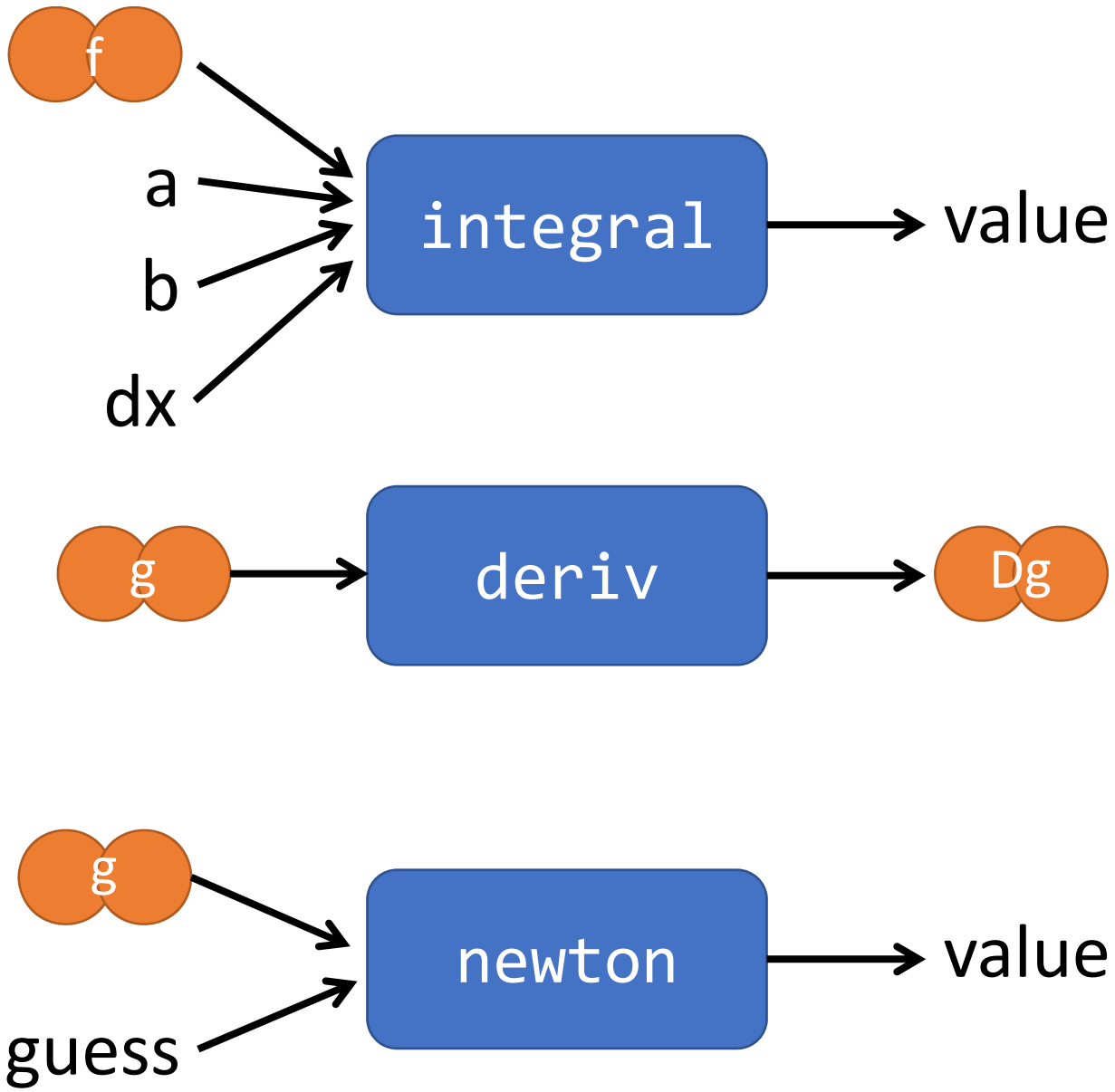
```
def product(term, a, next, b):
    if a > b:
        return 1
    else:
        return term(a) * product(term, next(a), next, b)
```

```
def sum(term, a, next, b):
    if a > b:
        return 0
    else:
        return term(a) + sum(term, next(a), next, b)
```

Higher Order Function

```
def fold2(term, a, next, b, op, base):
  if a > b:
    return base
  else:
    return op(term(a),
              fold2(term, next(a), next, b, op, base))

def sum(term, a, next, b):
  return fold2(term, a, next, b, lambda x,y: x+y, 0)
def product(term, a, next, b):
  return fold2(term, a, next, b, lambda x,y: x*y, 1)
```

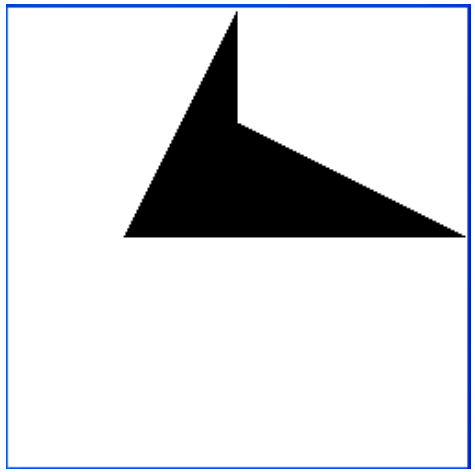

term
a
next
b
op
base
fold2
value
term
a
next
b
sum
value

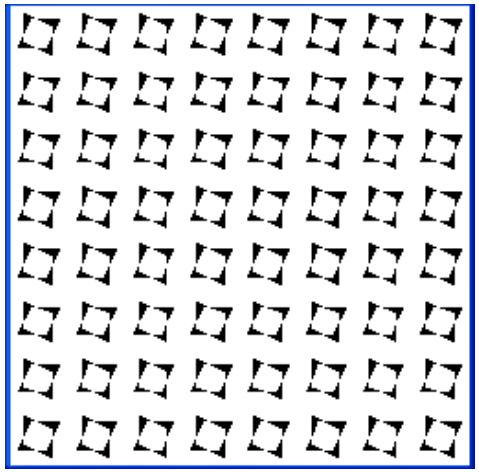

f
a
b
dx
integral
value
g
deriv
Dg
g
guess
newton
value

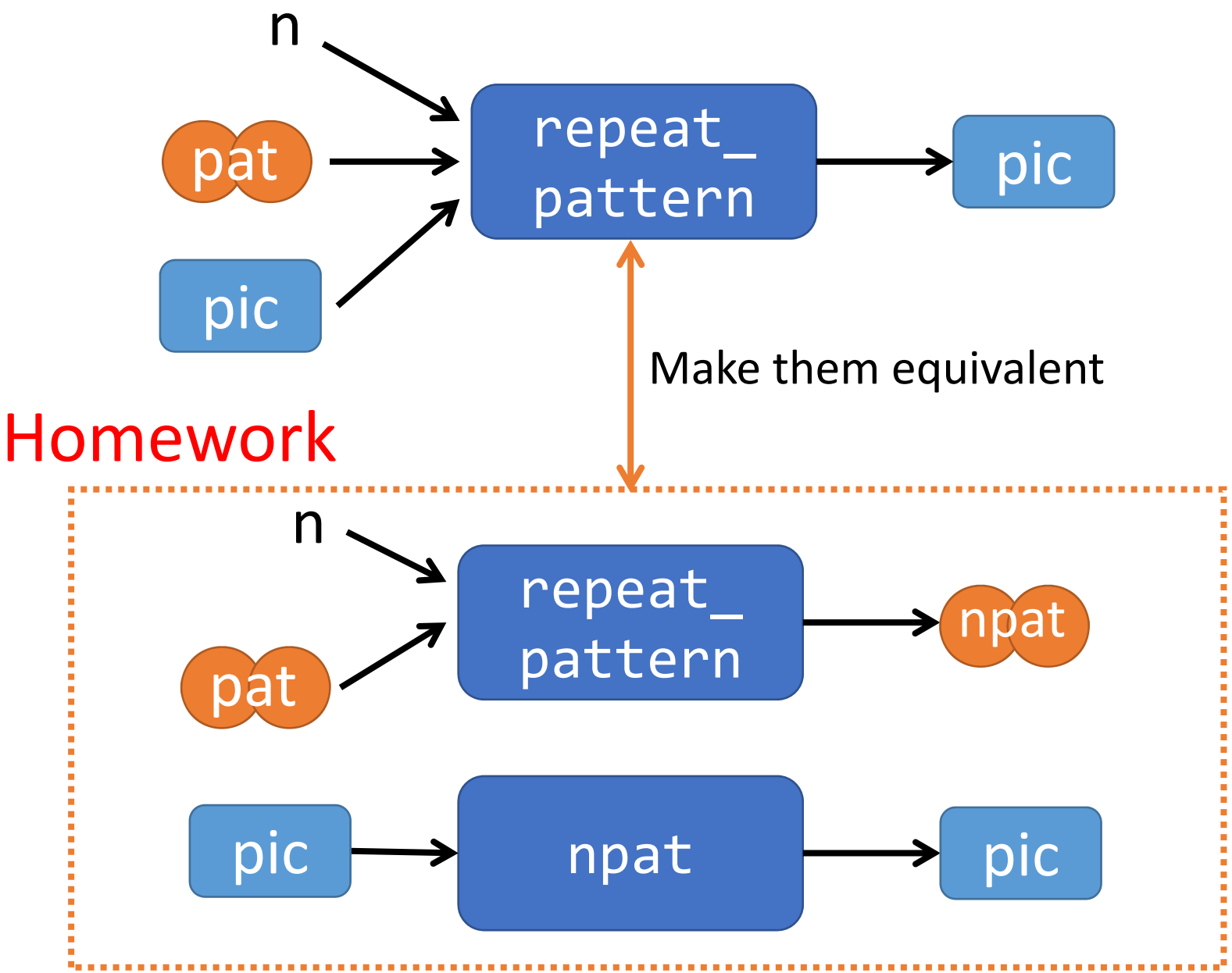
Repeating Patterns

```
def repeat_pattern(n, pat, pic):
  if n == 0:
    return pic
  else:
    return pat(repeat_pattern(n-1, pat, pic))

show(repeat_pattern(4, make_cross, nova_bb))
```

Isn't this a function also?






n
pat
pic
repeat_
pattern
pic
Make them equivalent
Homework
n
pat
repeat_
pattern
npat
pic
npat
pic

Do NOT try to remember the following functions:

- sum_integers(), sum_cube() … sum()
- integral()
- deriv()
- newtowns_method()
- product()
- fold()
- fold2()
- …

They are just examples.

You may re-write them after class.

CS1010S is NOT about memory work.

It is about UNDERSTANDING.

Summary

- Python functions are first-class objects.
  - They may be named by variables.
  - They may be passed as arguments to functions.
  - They may be returned as the results of functions.

Summary

- Higher-order functions capture common programming patterns.
- Functions can be returned as the result of functions

Required Competencies

1. Understand how to use higher-order functions to define specific functions
2. Understand how to define higher-order functions by abstracting patterns

For practice (and to check your understanding…..)

- How would you define `factorial` in terms of `product`?
- How would you define `expt` in terms of `product`?