http://www.cs.cornell.edu/courses/cs1110/2022sp

Final Exam Review

CS 1110

Introduction to Computing Using Python

[E. Andersen, A. Bracy, D. Gries, L. Lee, S. Marschner, C. Van Loan, W. White]

Announcements

- No post-lecture office hours today
- Study Guide is published
- Extra review sessions happening
- Final Exam is Sunday, May 15

Where and When is your Exam?

- Check on Canvas
  - Final Exam Date & Time Assignments
    - Pretty much everyone is taking it in Barton
    - Only a few exceptions
  - Extended Time Exam Accommodations
- Closed Notes & Book, Reference Sheet
- Bring your Cornell ID

Expressions

An expression **represents** something

- Python ***evaluates it*** (turns it into a value)
- Similar to a calculator

Examples:

- 2.3
- (3 * 7 + 2) * 0.1

Types

Type: set of values & operations on them

Meaning of operations depends on type

Type **float:**
- Values: real numbers
- Ops: +, -, *, /,//, **,%

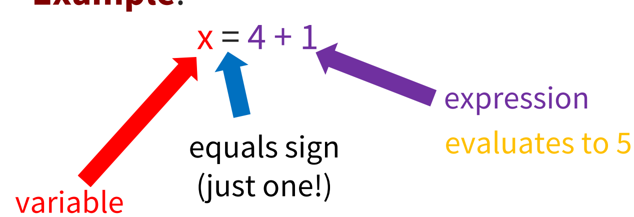
Type **int:**
- Values: integers
- Ops: +, -, *, //, %, **

Type **bool:**
- Values: `True`, `False`
- Ops: `not`, `and`, `or`

Type **str:**
- Values: strings
  - Double quotes: `"abc"`
  - Single quotes: `'abc'`
- Ops: + (concatenation)

Variable Assignment

Example:



An *assignment statement:*
- takes an *expression*
- evaluates it, and
- stores the *value* in a *variable*

In More Detail: Variables

- A **variable**
  - is a **named** memory location (**box**)
  - contains a **value** (in the box)
- Examples:

The type belongs to the *value*, not to the *variable*.

x [5] Variable **x**, with value 5 (of type **int**)

area [20.1] Variable **area**, w/ value 20.1 (of type **float**)

8

Expressions vs. Statements

Expression	Statement
• **Represents** something ▪ Python *evaluates it* ▪ End result is a value	• **Does** something ▪ Python *executes it* ▪ Need not result in a value
• Examples: ▪ 2.3 ▪ (3+5)/4 ▪ x == 5	• Examples: ▪ x = 2 + 1 ▪ x = 5

Look so similar
but they are not!

9

Executing an Assignment Statement

The command: `x = 3.0*x+1.0`

"Executing the command":

1. Evaluate right hand side `3.0*x+1.0`

2. Store the value in the variable x's box

- Requires both evaluate AND store steps
- Critical mental model for learning Python

10

Function Calls

- Function calls have the form:

  `best_function_ever(x,y,…)`

  function name — argument(s)

- Arguments
  - Separated by commas
  - Can be any expression

A function might have 0, 1, … or many arguments

11

Modules: Libraries vs. Scripts

Library	**Script**
• Provides functions, variables	• Behaves like an application
• **`import`** it into Python shell, don't include ".py"	• At command line prompt, Tell python to run the file (use full filename, including ".py")
• Within Python shell you have access to the functions and variables of the imported module	• After running the app you're back at the command line

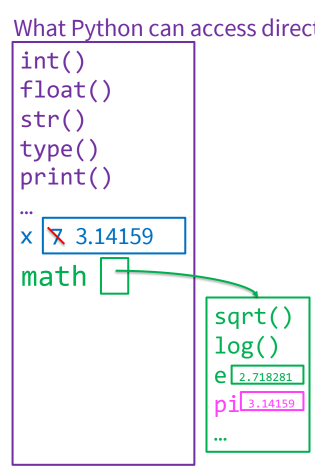
Files look the same.
Difference is how you use them.

12

Visualizing functions & variables

Running Example:
1. Built-in functions
2. Define a new variable
3. Import a module
4. Use a module variable

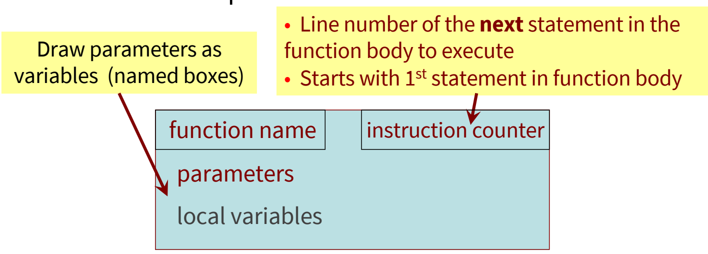
```
C:\> python
>>> x = 7
>>> import math
>>> x = math.pi
```



13

Understanding How Functions Work

- We draw pictures to show what is in memory
- **Call Frame:** representation of function call



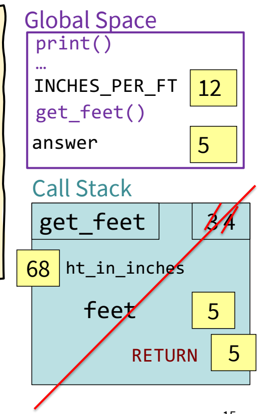
Not just a pretty picture!
The information in this picture depicts *exactly* what is stored in memory on your computer.

Function Access to Global Space

```
# height3.py

1 INCHES_PER_FT = 12
2 def get_feet(ht_in_inches):
3     feet = ht_in_inches // INCHES_PER_FT
4     return feet

5 answer = get_feet(68)
6 print(answer)
```

Python has just executed line 6.

```
C:\> python height3.py
5
```



A Precondition Is a Contract

- If precondition is met, **the function will work!**
- If precondition is **not** met… **no guarantees!**

Representative Tests

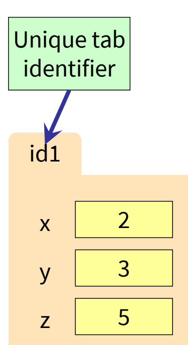
- Cannot test all inputs
  - "Infinite" possibilities
- Limit ourselves to tests that are **representative**
  - Each test is a significantly different input
  - Every possible input is similar to one chosen
- An art, not a science
  - If easy, never have bugs
  - Learn with much practice

Representative Tests for vowel_count(w)

- Word with just one vowel
  - For each possible vowel!
- Word with multiple vowels
  - Of the same vowel
  - Of different vowels
- Word with only vowels
- Word with no vowels

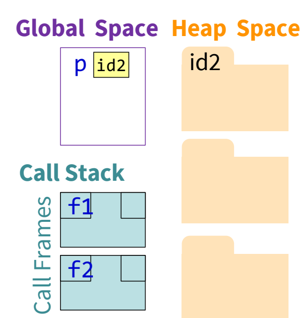
Objects: Organizing Data in Folders

- An object is like a manila folder
- It contains other variables
  - Variables are called attributes
  - These values can change
- It has an ID that identifies it
  - Unique number assigned by Python (just like a NetID for a Cornellian)
  - Cannot ever change
  - Has no meaning; only identifies



Storage in Python

- **Global Space**
  - What you "start with"
  - Stores global variables
  - Lasts until you quit Python
- **Heap Space**
  - Where "folders" are stored
  - Have to access indirectly
- **Call Stack** (with Frames)
  - Parameters
  - Other variables local to function
  - Lasts until function returns



Methods: a special kind of function

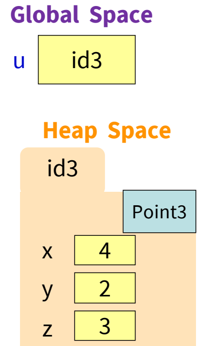
Methods are:

- Defined for specific classes
- Called using objects of that class

variable.method(*arguments* **)**

Example:

```
>>> import shapes
>>> u = shapes.Point3(4,2,3)
>>> u.greet()
"Hi! I am a 3-dimensional point located at (4,2,3)"
>>>
```

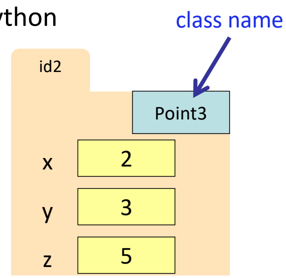


Built-in Types vs. Classes

Built-in types	Classes
Built-into Python	Provided by modules
Refer to instances as *values*	Refer to instances as *objects*
Instantiate with simple assignment statement	Instantiate with assignment statement with a *constructor*
Can ignore the folders	Must represent with folders

Classes are user-defined Types

Defining new classes = adding new types to Python

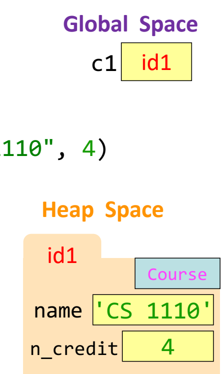


Example Classes

- Point3
- Rect
- Freq (A3), for word frequencies
- Doll (class, lab)
- Song, Mix (A4)

Evaluating a Constructor Expression

1. Constructor creates a new object (folder) of the class Course on the Heap
   - Folder is initially empty
   - Has id
2. Constructor calls `__init__ (self, "CS 1110", 4)`
   - `self` = identifier (*"Fill this folder!"*)
   - Other args come from the constructor call
   - commands in `__init__` populate folder
   - `__init__` has no return value! (*"I filled it!"*)
3. Constructor returns the id
4. LHS variable created, id is value in the box

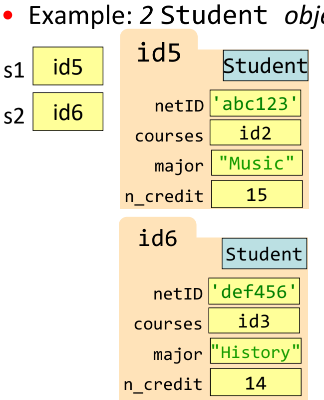



```
c1 = Course("CS 1110", 4)
```


Classes Have Folders Too

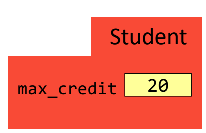
Object Folders

- Separate for each *instance*
- Example: *2* `Student` *objects*



Class Folders

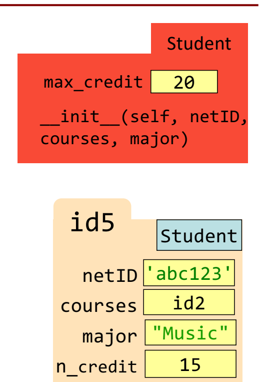
- Data common to **all** instances



- Not just data!
- *Everything* common to all instances goes here!

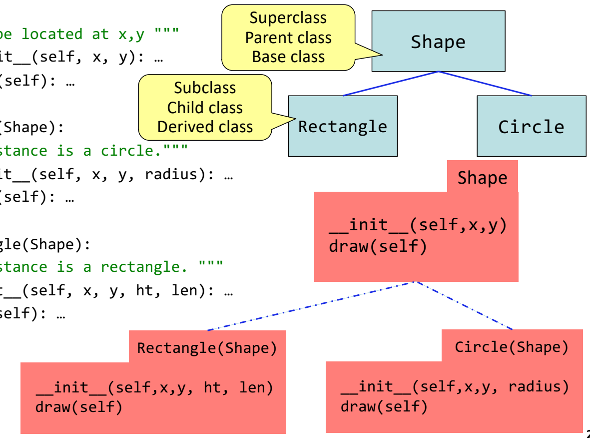
Object Methods

- Attributes live in object folder
- Class Attributes live in class folder
- Methods live in class folder



Defining a Subclass

```
class Shape:
    """A shape located at x,y """
    def __init__(self, x, y): …
    def draw(self): …

class Circle(Shape):
    """An instance is a circle."""
    def __init__(self, x, y, radius): …
    def draw(self): …

class Rectangle(Shape):
   """An in stance is a rectangle. """
   def __init__(self, x, y, ht, len): …
   def draw(self): …
```



__init__: write new one, access parent's

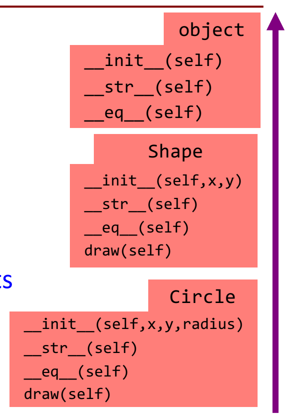
```
class Shape:
  """A shape @ location x,y """
  def __init__(self, x, y):
      self.x = x
      self.y = y

class Circle(Shape):
  """Instance is Circle @ x,y w/size radius"""
  def __init__(self, x, y, radius):
      super().__init__(x,y)
      self.radius = radius
```

- Want to use the original version of the method?
  - New method = original+more
  - Don't repeat code from the original
- Call old method explicitly

Understanding Method Overriding

```
c1 = Circle(1,2,4.0)
print(str(c1))
```

- Which __str__ do we use?
  - Start at bottom class folder
  - Find first method with name
  - Use that definition
- Each subclass automatically inherits methods of parent.
- New method definitions **override** those of parent.

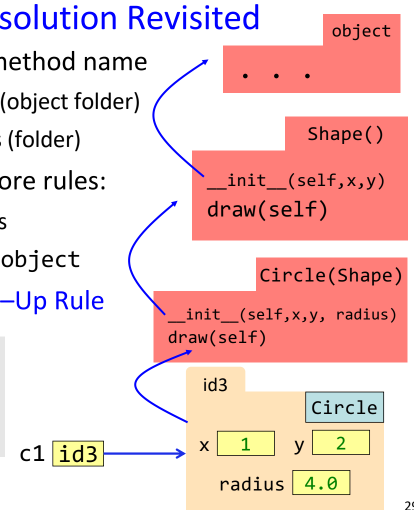


Name Resolution Revisited

- To look up attribute/method name
  1. Look first in instance (object folder)
  2. Then look in the class (folder)
- Subclasses add two more rules:
  3. Look in the superclass
  4. Repeat 3. until reach object

Often called the Bottom–Up Rule

```
c1 = Circle(1,2,4.0)
r = c1.radius
c1.draw()
```



Operator Overloading: Equality

Implement __eq__ to check for equivalence of two Fractions instead

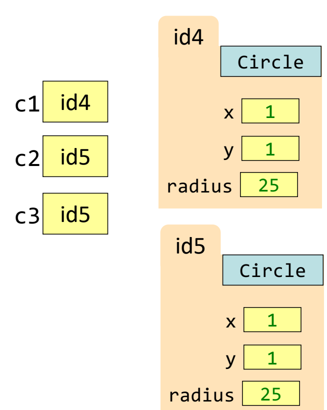
```
class Fraction():
    """Instance attributes:
        numerator:    top       [int]
        denominator: bottom [int > 0]"""

    def __eq__(self,q):
      """Returns: True if self, q equal,
        False if not, or q not a Fraction"""
        if type(q) != Fraction:
           return False
        left = self.numerator*q.denominator
        right = self.denominator*q.numerator
        return left == right
```


eq vs. is

`==` compares equality

`is` compares identity

```
c1 = Circle(1, 1, 25)
c2 = Circle(1, 1, 25)
c3 = c2
```




```
c1 == c2 → ? True
c1 is c2 → ? False
c2 == c3 → ? True
c2 is c3 → ? True
```

The `isinstance` Function

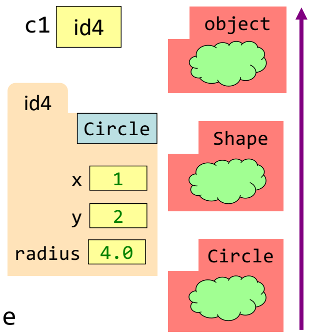
`isinstance(<obj>,<class>)`

- True if <obj>'s class is same as or a subclass of <class>
- False otherwise

Example:

`c1 = Circle(1,2,4.0)`

- `isinstance(c1,Circle)` is True
- `isinstance(c1,Shape)` is True
- `isinstance(c1,object)` is True
- `isinstance(c1,str)` is False
- Generally preferable to `type`
  - Works with base types too!



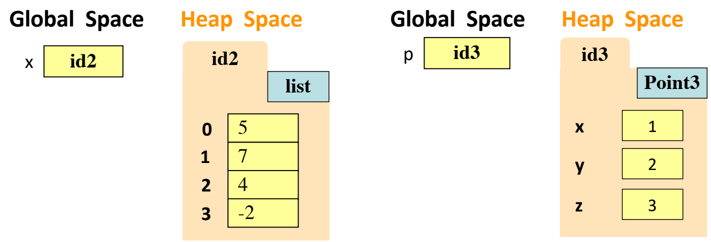
Lists: objects with special "string-like" syntax

List

- Attributes are indexed
  - Example: x[2]

Objects

- Attributes are named
  - Example: p.x



Sequences: Lists of Values

String

- s = 'abc d'

0	1	2	3	4
a	b	c		d

- Put characters in quotes
  - Use \' for quote character
- Access characters with []
  - s[0] is 'a'
  - s[5] causes an error
  - s[0:2] is 'ab' (excludes c)
  - s[2:] is 'c d'
- len(s) → 5, length of string

List

- x = [5, 6, 5, 9, 15, 23]

0	1	2	3	4	5
5	6	5	9	15	23

- Put values inside [ ]
  - Separate by commas
- Access **values** with []
  - x[0] is 5
  - x[6] causes an error
  - x[0:2] is [5, 6] (excludes 2nd 5)
  - x[3:] is [9, 15, 23]
- len(x) → 6, length of list

Sequence is a name we give to both

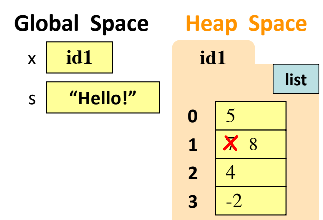
List is *mutable*; strings are not

- **Format**:

  <var>[<index>] = <value>
  - Reassign at index
  - Affects folder contents
  - Variable is unchanged

- Strings cannot do this
  - Strings are **immutable**

```
x = [5, 7,4,-2]
x[1] = 8
s = "Hello!"
s[0] = 'J'
TypeError: 'str' object does not
support item assignment
```



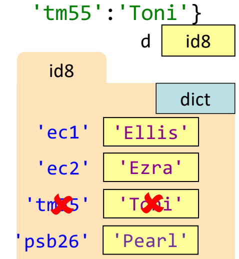
Things that Work for All Sequences

s = 'slithy'		x = [5, 6, 9, 6, 15, 5]
s.index('s') → 0	methods	x.index(5) → 0
s.count('t') → 1		x.count(6) → 2
len(s) → 6	built-in fns	len(x) → 6
s[4] → "h"	slicing	x[4] → 15
s[1:3] → "li"		x[1:3] → [6, 9]
s[3:] → "thy"		x[3:] → [6, 15, 5]
s[–2] → "h"		x[–2] → 15
s + ' toves' → "slithy toves"	operators	x + [1, 2] → [5, 6, 9, 6, 15, 5, 1, 2]
s * 2 → "slithyslithy"		x * 2 → [5, 6, 9, 6, 15, 5, 5, 6, 9, 6, 15, 5]
't' in s → True		15 in x → True

Dictionaries are mutable

1. Can reassign values
   - `d['ec1'] = 'Ellis'`
2. Can add new keys
   - `d['psb26'] = 'Pearl'`
3. Can delete keys
   - `del d['tm55']`

```
d = {'ec1':'Ezra',
     'ec2':'Ezra',
     'tm55':'Toni'}
```

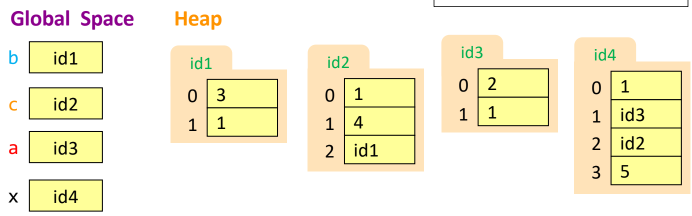


Deleting key deletes both key and value

Nested Lists

- Lists can hold any objects
- Lists are objects
- Therefore lists can hold other lists!

```
b = [3, 1]
c = [1, 4, b]
a = [2, 1]
x = [1, a, c, 5]
```



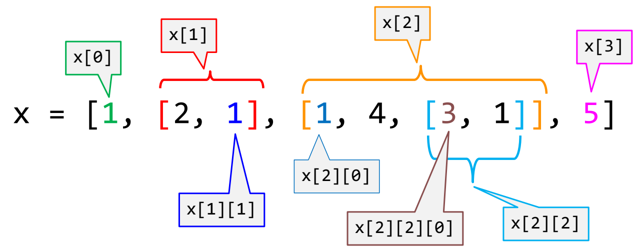
This is drawing accurate, but a little hard to reason about…

Nested Lists

Conceptually, you can visualize nested lists like this:

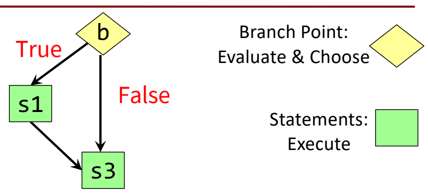
```
b = [3, 1]
c = [1, 4, b]
a = [2, 1]
x = [1, a, c, 5]
```

```
x = [1, [2, 1], [1, 4, [3, 1]], 5]
```

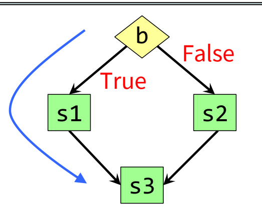


Conditionals: "Control Flow" Statements

```
if b:
    s1    # statement
s3        # statement
```




```
if b:
    s1
else:
    s2
s3
```



Flow
Program only takes one path during an execution (something will not be executed!)

Conditionals: If-Elif-Else-Statements (2)

Format

```
if <Boolean expression>:
    <statement>
    …
elif <Boolean expression>:
    <statement>
    …
…
else:
    <statement>
    …
```

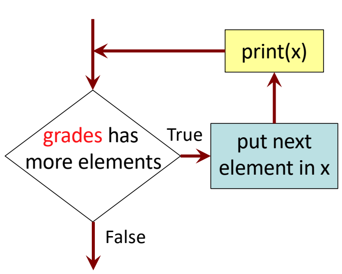
Notes on Use

- No limit on number of `elif`
  - Must be between `if`, `else`
- `else` is optional
  - `if-elif` by itself is fine
- Booleans checked in order
  - Once Python finds a true <Boolean-expression>, skips over all the others
  - `else` means **all** <Boolean-expression> are false

For Loops: Processing Sequences

```
for x in grades:
    print(x)
```

- **loop sequence:** grades
- **loop variable:** x
- **loop body:** print(x)



To execute the for-loop:

1) Check if there is a "next" element of loop sequence
2) If so:
   - assign next sequence element to loop variable
   - Execute all of the body
   - Go back to 1)
3) If not, terminate execution

For Loop with labels

```
def num_zeroes(the_list):
    """Returns: the number of zeroes in the_list
    Precondition: the_list is a list"""

  count = 0
  for x in the_list:
      if x == 0:
          count = count + 1
  return count
```

Accumulator variable: `count = 0`
Loop sequence: `the_list`
Loop variable: `x`
Loop body: `if x == 0: count = count + 1`

Modifying the Contents of a List

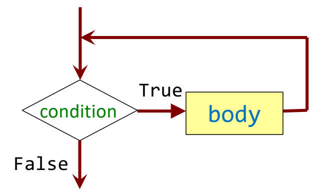
```python
def add_bonus(grades):
    """Adds 1 to every element in a list of grades
    (either floats or ints)"""
    size = len(grades)
    for k in range(size):
        grades[k] = grades[k]+1

lab_scores = [8,9,10,5,9,10]
print("Initial grades are: "+str(lab_scores))
add_bonus(lab_scores)
print("With bonus, grades are: "+str(lab_scores))
```

*If you need to modify the list, you **need to use range** to get the indices.*

Watch this in the python tutor!

Beyond Sequences: The `while`-loop

```
while <condition >:
    statement 1
    …              } body
    statement n
```



Relationship to `for`-loop

- Broader notion of "keep working until done"
- Must explicitly ensure condition becomes false
- *You* explicitly manage what changes per iteration

Recursion

Recursive Function:

A function that calls *itself*

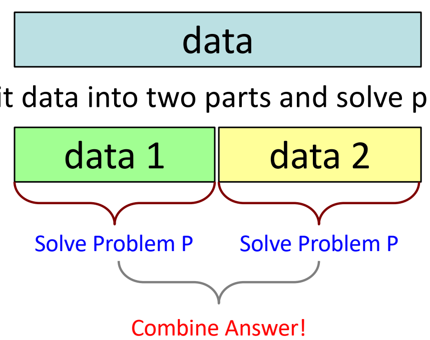
Two parts to every recursive function:

1. A simple case: can be solved easily
2. A complex case: can be made simpler (and simpler, and simpler… until it looks like the simple case)

Recursion is great for Divide and Conquer

Goal: Solve problem P on a piece of data

Idea: Split data into two parts and solve problem



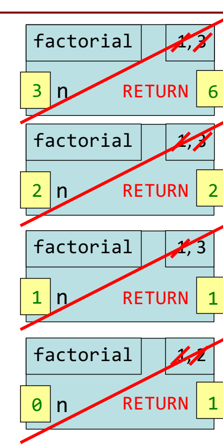
Three Steps for Divide and Conquer

1. Decide what to do on "small" data
   - Some data cannot be broken up
   - Have to compute this answer directly
2. Decide how to break up your data
   - Both "halves" should be smaller than whole
   - Often no wrong way to do this (next lecture)
3. Decide how to combine your answers
   - Assume the smaller answers are correct
   - Combine them to give the aggregate answer

Recursive Call Frames (all calls complete!)

```python
def factorial(n):
    """Returns: factorial of n.
    Precondition: n ≥ 0 an int"""
    if n == 0:
        return 1
    return n*factorial(n-1)
```

factorial(3)



Search Algorithms

Recall from last lecture:

- Searching for data is a common task
  - Linear search: on the order of n
    - input doubles? → work **doubles**!
  - Binary search: on the order of log2 n
    - input doubles? → work **increases by just 1 unit**!
    - BUT data needs to be sorted…

- **Sorting** data now suddenly interesting…

Sorting Algorithms

- Sorting data is a common task
  - Insertion sort: on the order of n^2
    - input doubles? → work **quadruples**! (yikes)

  - Merge sort: on the order of $n \cdot \log_2(n)$
    - input doubles? → work increases by a bit more than double

For fun, check out the visualizations:
https://www.youtube.com/watch?v=xxcpvCGrCBc
https://www.youtube.com/watch?v=ZRPoEKHXTJg