

DISCUSSION 05


Sequences, Data Abstraction, Trees

Mingxiao Wei

mingxiaowei@berkeley.edu

Sep 29, 2022

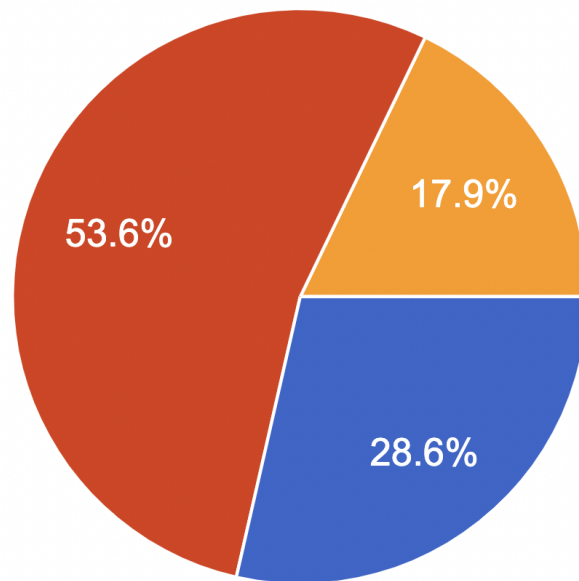
LOGISTICS

- CATS 
 - Try out the game here: cats.cs61a.org
 - The whole project due tomorrow 09/30
 - Submit everything by today 09/29 for one extra credit
- NO homework due this week - homework 04 due next Thu 10/06

HIGHLIGHT FROM LAST TIME 🙄🙄

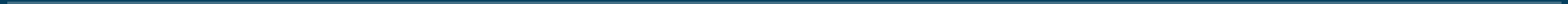
Morning bird or night owl or ...?

28 responses



- morning bird 🐣
- night owl 🦉
- constantly awake 🍷

SEQUENCES



SEQUENCES - INTRO

- Sequences are ordered collections of values that support element-selection and have length.
 - Lists, strings, tuples, even ranges, are all sequences
- Sequences support the following operations:
 - `s[0]` - element selection / indexing
 - `len(s)` - length of `s`
 - `s[i:j:k]` - slicing
 - `max(s)` / `min(s)` - largest / smallest item from `s`
 - `x in s` / `x not in s` - checking whether or not `x` is in `s`

SEQUENCES - EXAMPLES

```
>>> x = 'Hello there Oski!'
>>> len(x) # space and punctuation counts!
17
>>> x[6:]
'there Oski!'
>>> x[::-1]
'!iksO ereht olleH'
>>> 'Oski' in x and 'here' in x # look for a substring in x
True
>>> 'oski' not in x # case sensitive
True
```

SEQUENCES - EXAMPLES

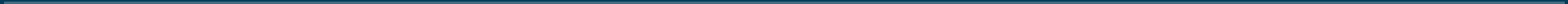
- To iterate through a sequence, we can either iterate over the items in the sequence directly or iterate through its indices

```
>>> x = 'I am not Oski.'
>>> vowel_count = 0
>>> for i in range(len(x)):
...     if x[i] in 'aeiou':
...         vowel_count += 1
...
>>> vowel_count
5
```

```
>>> for i in x[-5:-1]:
...     print(i)
...
O
s
k
i
```

WORKSHEET Q1, 2

DATA ABSTRACTION



DATA ABSTRACTION - INTRO

- Data abstraction allows us to treat code as "objects", so that we don't have to worry about the actual implementation (e.g., how information is stored and calculated) - we just need to know what it does.
- The nature of abstraction allows whoever uses them to assume that the functions have been written correctly and work as described.
- A data abstraction consists of two types of functions:
 - **Constructors**: functions that build and return the abstract data type
 - **Selectors**: functions that retrieve information from the data type.

RATIONAL NUMBERS REVISIT

- Constructor: `rational(n, d)` returns a rational number $x = \frac{n}{d}$ using some underlying representation, which we, as users, do not need to know anything of
- Selectors:
 - `numer(x)` returns the numerator of x
 - `denom(x)` returns the denominator of x
- Arithmetic Operations:

```
def mul_rational(x, y):  
    return rational(numer(x) * numer(y), \  
                    denom(x) * denom(y))
```

- Note how we can manipulate rational numbers using their constructor/selectors without knowing their implementation
- Side note: the `\` is used to indicate that the expression continues on the next line

RATIONAL NUMBERS REVISIT

There are many ways to implement the rational number data abstraction. Below are two examples:

```
def rational(n, d):  
    return [n, d]  
def numer(x):  
    return x[0]  
def denom(x):  
    return x[1]
```

```
def rational(n, d):  
    return {'n': n, 'd': d}  
def numer(x):  
    return x['n']  
def denom(x):  
    return x['d']
```

No matter which one we use, the rational number data abstraction has the same, correct behavior from the users' end.

In other words, as programmers, we can design the underlying implementation for the data abstraction however we want as long as it behaves correctly according to its documentation.

DON'T BREAK THE ABSTRACTION BARRIER!

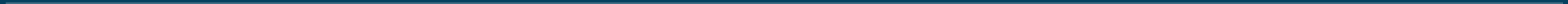
Parts of the program that...	Treat rationals as...	Using...
Use rational numbers to perform computation	whole data values	<code>add_rational, mul_rational</code> <code>rationals_are_equal, print_rational</code>
Create rationals or implement rational operations	numerators and denominators	<code>rational, numer, denom</code>
Implement selectors and constructor for rationals	two-element lists	list literals and element selection
<i>Implementation of lists</i>		

Source: [lecture 13 slides](#)

DON'T BREAK THE ABSTRACTION BARRIER!

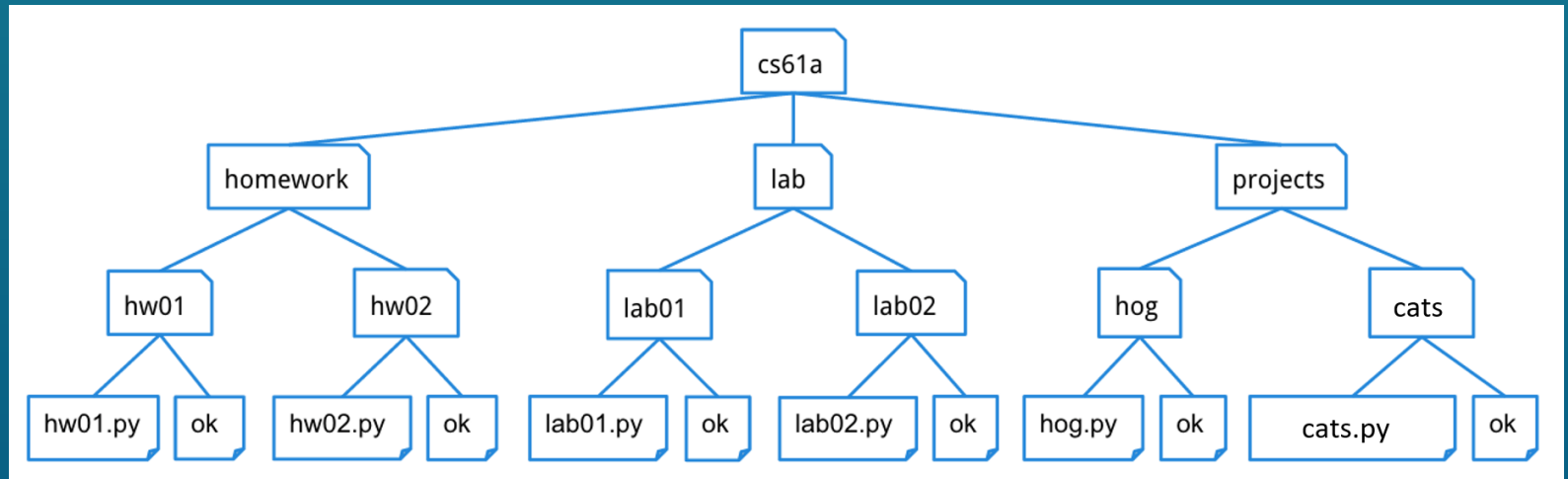
- Why?
 - Breaking the abstraction barrier is essentially repeating some function's job
 - Once the underlying implementation changes, the code may not work anymore
- Examples of violating the abstraction barriers:
 - `mul_rational([1, 2], [3, 4])` - should be `mul_rational(rational(1, 2), rational(1, 2))`
 - ```
def divide_rational(x, y):
 return [x[0] * y[1], x[1] * y[0]]
should be:
def divide_rational(x, y):
 return rational(denom(x) * numer(y), \
 numer(x) * denom(y))
```
- Takeaway: do not repeat any work that an existing function could do!

# TREES



# TREES - INTRO

- A tree is a data structure that represents a hierarchy of information.
  - For example, a file system:



As you can see, unlike trees in nature, the tree abstract data type is drawn with the root at the top and the leaves at the bottom.



# TREES - TERMINOLOGY

---

- **node**: any location within the tree (e.g., root node, leaf nodes, etc.)
- **root**: the node at the top of the tree
- **label**: the value in a node
- **branches**: a list of trees directly under the tree's root
- **leaf**: a tree with zero branches
- **parent node**: a node that has at least one branch.
- **child node**: a node that has a parent. A child node can only have one parent.
- **depth**: the number of edges between the root to the node. The root has depth 0.
- **height**: the depth of the lowest (furthest from the root) leaf.

# TREES - DATA ABSTRACTION

---

- Constructor:
  - `tree(label, branches=[ ])`: creates and returns a tree object with the given `label` value at its root node and list of `branches`. Notice that the second argument to this constructor, `branches`, is optional - if you want to make a tree with no branches, leave this argument empty
- Selector:
  - `label(tree)`: returns the value in the root node of `tree`.
  - `branches(tree)`: returns the list of branches of `tree`.
- Convenience function:
  - `is_leaf(tree)`: returns `True` if `tree`'s list of branches is empty, and `False` otherwise.

# TREES - DATA ABSTRACTION IMPLEMENTATION

---

```
def tree(label, branches=[]):
 return [label] + list(branches)

def label(tree):
 return tree[0]

def branches(tree):
 return tree[1:]

def is_leaf(tree):
 return not branches(tree)
```

# WORKING WITH DATA ABSTRACTION

---

- There's no mutation in data abstraction (i.e., we cannot mutate an existing object) - to "update" an object, rather than resetting the attributes directly, we need to create a new object with the updated attributes

```
t0 = tree(61, [tree('a'), tree('b')])
```

```
update the label of t0 to be 100
```

```
t1 = tree(100, branches(t0))
```

```
add one more branch to t0
```

```
t2 = tree(label(t0), branches(t0) + [tree('c')])
```

# WORKSHEET Q3-Q6

---

# ATTENDANCE! 🤠

---

[go.cs61a.org/mingxiao-att](https://go.cs61a.org/mingxiao-att)

- The attendance form and slides are both linked on our [section website](#)!
- Once again, please do remember to fill out the form by midnight today!!