Dictionaries, Matrices, and Trees
Today will cover...

• Announcements
• Dictionary type
• Matrix abstraction
• Tree abstraction
• Tree processing
### Review: Layers of abstraction

<table>
<thead>
<tr>
<th>Primitive Representation</th>
<th>1 2 3 True False</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data abstraction</td>
<td>make_rat() numer() denom()</td>
</tr>
<tr>
<td></td>
<td>add_rat() mul_rat()</td>
</tr>
<tr>
<td></td>
<td>print_rat() equal_rat()</td>
</tr>
<tr>
<td>User program</td>
<td>exact_harmonic_number()</td>
</tr>
</tbody>
</table>

Each layer only uses the layer above it.
<table>
<thead>
<tr>
<th>Type</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integers</td>
<td>0 -1 0xFF 0b1101</td>
</tr>
<tr>
<td>Booleans</td>
<td>True False</td>
</tr>
<tr>
<td>Functions</td>
<td>def f(x)... lambda x: ...</td>
</tr>
<tr>
<td>Strings</td>
<td>&quot;pear&quot;&quot;I say, &quot;hello!&quot;&quot;</td>
</tr>
<tr>
<td>Tuples</td>
<td>(1, 10) (&quot;Oh&quot;, &quot;hi&quot;, 11)</td>
</tr>
<tr>
<td>Ranges</td>
<td>range(11) range(1, 6)</td>
</tr>
<tr>
<td>Lists</td>
<td>[] [&quot;apples&quot;, &quot;bananas&quot;] [x**3 for x in range(2)]</td>
</tr>
</tbody>
</table>
A dict is a mutable mapping of key-value pairs

```python
states = {
    "CA": "California",
    "DE": "Delaware",
    "NY": "New York",
    "TX": "Texas",
    "WY": "Wyoming"
}
```

Queries:

```python
>>> len(states)

>>> "CA" in states

>>> "ZZ" in states
```
Dictionaries

A dict is a mutable mapping of key-value pairs

```python
states = {
    "CA": "California",
    "DE": "Delaware",
    "NY": "New York",
    "TX": "Texas",
    "WY": "Wyoming"
}
```

Queries:

```python
>>> len(states)
5

>>> "CA" in states
True

>>> "ZZ" in states
False
```
Dictionaries

A dict is a mutable mapping of key-value pairs

```python
states = {
    "CA": "California",
    "DE": "Delaware",
    "NY": "New York",
    "TX": "Texas",
    "WY": "Wyoming"
}
```

Queries:

```python
>>> len(states)
5
>>> "CA" in states
True
>>> "ZZ" in states
```
Dictionaries

A dict is a mutable mapping of key-value pairs

```python
states = {
    "CA": "California",
    "DE": "Delaware",
    "NY": "New York",
    "TX": "Texas",
    "WY": "Wyoming"
}

Queries:

>>> len(states)
5

>>> "CA" in states
True

>>> "ZZ" in states
False
```
Dictionary selection

```python
words = {
    "más": "more",
    "otro": "other",
    "agua": "water"
}

Select a value:

```python
>>> words["otro"]

```python
>>> first_word = "agua"
>>> words[first_word]

```python
>>> words["pavo"]

```python
>>> words.get("pavo", "🤔")
```
Dictionary selection

```python
words = {
    "más": "more",
    "otro": "other",
    "agua": "water"
}

Select a value:

```python
>>> words["otro"]
'other'

```python
>>> first_word = "agua"
>>> words[first_word]

```python
>>> words["pavo"]

```python
>>> words.get("pavo", "🤔")
```
Dictionary selection

```python
dict = {
    "más": "more",
    "otro": "other",
    "agua": "water"
}
```

Select a value:

```python
>>> dict["otro"]
'other'
```

```python
>>> first_word = "agua"
>>> dict[first_word]
'water'
```

```python
>>> dict["pavo"]
```

```python
>>> dict.get("pavo", "🤔")
'thinking'
```
Dictionary selection

```python
words = {
    "más": "more",
    "otro": "other",
    "agua": "water"
}

Select a value:

```text
>>> words["otro"]
'other'
```text

```text
>>> first_word = "agua"
>>> words[first_word]
'water'
```text

```text
>>> words["pavo"]
KeyError: pavo
```text

```text
>>> words.get("pavo", "🤔")
'🤔'
```text
Dictionary selection

```python
words = {
    "más": "more",
    "otro": "other",
    "agua": "water"
}

Select a value:

```text
>>> words["otro"]
'other'

```text
>>> first_word = "agua"
>>> words[first_word]
'water'

```text
>>> words["pavo"]
KeyError: pavo

```text
>>> words.get("pavo", "🤔")
🤔
```
Dictionary mutation

Create an empty dict:

```python
users = {}
```

Add values:

```python
users["profpamela"] = "b3stp@ssEvErDontHackMe"
```

Change values:

```python
users["profpamela"] += "itsLongerSoItsMoreSecure!!"
```

```python
>>> users["profpamela"]
```

```python
b3stp@ssEvErDontHackMeitsLongerSoItsMoreSecure!!
```
Dictionary mutation

Create an empty dict:

```python
users = {}
```

Add values:

```python
users["profpamela"] = "b3stp@ssEvErDontHackMe"
```

Change values:

```python
users["profpamela"] += "itsLongerSoItsMoreSecure!!"
```

```python
>>> users["profpamela"]
'b3stp@ssEvErDontHackMeitsLongerSoItsMoreSecure!!'
```
Dictionary rules

- A key **cannot** be a list or dictionary (or any mutable type)
- All keys in a dictionary are distinct (there can only be one value per key)
- The values can be any type, however!

```python
spiders = {
    "smeringopus": {
        "name": "Pale Daddy Long-leg",
        "length": 7
    },
    "holocnemus pluchei": {
        "name": "Marbled cellar spider",
        "length": (5, 7)
    }
}
```
Dictionary iteration

```python
insects = {"spiders": 8, "centipedes": 100, "bees": 6}
for name in insects:
    print(insects[name])
```

...is the same as:

```python
for name in list(insects):
    print(insects[name])
```

What will be the order of items?
Dictionary iteration

```python
insects = {"spiders": 8, "centipedes": 100, "bees": 6}
for name in insects:
    print(insects[name])
```

...is the same as:

```python
for name in list(insects):
    print(insects[name])
```

What will be the order of items?

```
8 100 6
```

Keys are iterated over in the order they are first added.
## Nested data

<table>
<thead>
<tr>
<th>Type</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lists of lists</td>
<td><code>[ [1, 2], [3, 4] ]</code></td>
</tr>
<tr>
<td>Lists of tuples</td>
<td><code>[ (1, 2), (3, 4) ]</code></td>
</tr>
<tr>
<td>Tuples of tuples</td>
<td><code>( (1, 2), (3, 4) )</code></td>
</tr>
<tr>
<td>Dicts of tuples</td>
<td><code>{ &quot;x&quot;: (1, 2), &quot;y&quot;: (3, 4) }</code></td>
</tr>
<tr>
<td>Dicts of lists</td>
<td><code>{ &quot;heights&quot;: [89, 97], &quot;ages&quot;: [6, 8] }</code></td>
</tr>
</tbody>
</table>

...what else?!
### Nested data

<table>
<thead>
<tr>
<th>Type</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lists of lists</td>
<td><code>[ [1, 2], [3, 4] ]</code></td>
</tr>
<tr>
<td>Lists of tuples</td>
<td><code>[ (1, 2), (3, 4) ]</code></td>
</tr>
<tr>
<td>Tuples of tuples</td>
<td><code>( (1, 2), (3, 4) )</code></td>
</tr>
<tr>
<td>Dicts of tuples</td>
<td><code>{ &quot;x&quot;: (1, 2), &quot;y&quot;: (3, 4) }</code></td>
</tr>
<tr>
<td>Dicts of lists</td>
<td><code>{ &quot;heights&quot;: [89, 97], &quot;ages&quot;: [6, 8] }</code></td>
</tr>
</tbody>
</table>

...what else?! Dicts of dicts, Lists of dicts, etc.
Next up: more abstractions
Matrices

Consider a matrix (two-dimensional table) like this:

\[
\begin{array}{cccc}
1 & 2 & 0 & 4 \\
0 & 1 & 3 & -1 \\
0 & 0 & 1 & 8 \\
\end{array}
\]

That matrix has three **rows** and four **columns**, with integer values in each location.
Matrices: Data abstraction

We want this constructor, selector, and mutator:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>matrix(rows, cols)</td>
<td>Returns a ROWS x COLS matrix with all values set to 0</td>
</tr>
<tr>
<td>value(matrix, row, col)</td>
<td>Returns value of MATRIX at (ROW, COL)</td>
</tr>
<tr>
<td>set_value(matrix, row, col, val)</td>
<td>Sets value of MATRIX at (ROW, COL) to VAL</td>
</tr>
</tbody>
</table>

How could we implement? Answer the poll!
Matrices: Implementation A

A list of lists, row-major order:

\[
\begin{bmatrix}
[1,2,0,4],
[0,1,3,-1],
[0,0,1,8]
\end{bmatrix}
\]

def matrix(rows, cols):
    return [ [0 for col in range(cols)] for row in range(rows) ]

def value(matrix, row, col):
    return matrix[row][col]

def set_value(matrix, row, col, val):
    matrix[row][col] = val

m = matrix(3, 4)
set_value(m, 0, 0, 1)
set_value(m, 0, 1, 2)
set_value(m, 0, 3, 4)
Matrices: Implementation B

A list of lists, column-major order:

\[
\begin{bmatrix}
[1,0,0],&[2,1,0],&[0,3,1],&[4,-1,8]
\end{bmatrix}
\]

def matrix(rows, cols):
    return [ [ 0 for row in range(rows)] for col in range(cols) ]

def value(matrix, row, col):
    return matrix[col][row]

def set_value(matrix, row, col, val):
    matrix[col][row] = val

m = matrix(3, 4)
set_value(m, 0, 0, 1)
set_value(m, 0, 1, 2)
set_value(m, 0, 3, 4)
Matrices: Implementation C

A tuple of lists, row-major order:

( [1,0,0], [2,1,0], [0,3,1], [4, -1,8] )

def matrix(rows, cols):
    return tuple([0 for col in range(cols)] for row in range(rows))

def value(matrix, row, col):
    return matrix[row][col]

def set_value(matrix, row, col, val):
    matrix[row][col] = val

m = matrix(3, 4)
set_value(m, 0, 0, 1)
set_value(m, 0, 1, 2)
set_value(m, 0, 3, 4)
Matrices: Implementation D

A list of tuples?

\[
\begin{bmatrix}
(1,2,0,4), & (0,1,3,-1), & (0,0,1,8)
\end{bmatrix}
\]

def matrix(rows, cols):
    return [ tuple(0 for col in range(cols))
              for row in range(rows) ]

def value(matrix, row, col):
    return matrix[row][col]

def set_value(matrix, row, col, val):
    matrix[row][col] = val

m = matrix(3, 4)
set_value(m, 0, 0, 1)
set_value(m, 0, 1, 2)
set_value(m, 0, 3, 4)
Matrices: Implementation D

A list of tuples?

\[
\begin{pmatrix}
(1,2,0,4), & (0,1,3,-1), & (0,0,1,8) \\
\end{pmatrix}
\]

def matrix(rows, cols):
    return [ tuple(0 for col in range(cols)) for row in range(rows) ]

def value(matrix, row, col):
    return matrix[row][col]

def set_value(matrix, row, col, val):
    matrix[row][col] = val

m = matrix(3, 4)
set_value(m, 0, 0, 1)
set_value(m, 0, 1, 2)
set_value(m, 0, 3, 4)
Matrices: Implementation D2

A list of tuples?

\[
\begin{bmatrix}
(1,2,0,4), & (0,1,3,-1), & (0,0,1,8)
\end{bmatrix}
\]

def matrix(rows, cols):
    return [ tuple(0 for col in range(cols))
             for row in range(rows) ]

def value(matrix, row, col):
    return matrix[row][col]

def set_value(matrix, row, col, val):
    matrix[row] = matrix[row][:col] + (val,) + matrix[row][col+1:]

m = matrix(3, 4)
set_value(m, 0, 0, 1)
set_value(m, 0, 1, 2)
set_value(m, 0, 3, 4)
Designing an implementation

Which implementation was your favorite?
Answer the poll!

When might you use a tuple?
When might you use a list?
When might you use a dict?
Each oval is a **node**
Top node is the **root**
Each node is itself the root of another tree (called a **subtree**); the nodes immediately below are its **children**
Nodes without children are **leaves**; others are **inner nodes**
Each node generally has a **label**
**Trees: Data abstraction**

We want this constructor and selectors:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>tree(label, children)</td>
<td>Returns a tree with given LABEL at its root, whose children are CHILDREN</td>
</tr>
<tr>
<td>label(tree)</td>
<td>Returns the label of root node of TREE</td>
</tr>
<tr>
<td>children(tree)</td>
<td>Returns the children of TREE (each a tree).</td>
</tr>
<tr>
<td>is_leaf(tree)</td>
<td>Returns true if TREE is a leaf node.</td>
</tr>
</tbody>
</table>

How could we implement? Answer the poll!
Trees: Implementation A

A list of label + list for each tree/subtree:

```
[20, [12, [9, [7], [2]], [3]], [8, [4], [4]]]
```

def tree(label, children=[]):
    return [label] + children

def label(tree):
    return tree[0]

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

def is_leaf(tree):
    return len(children(tree)) == 0

t = tree(20, [tree(12,
            [tree(9,
                [tree(7), tree(2)]),
                tree(3))],
            tree(8,
                [tree(4), tree(4)]))]
Trees: Implementation B

A number-list tuple for each tree/subtree:

\[(20, [(12, [[9, [[7, []], [2, []]]], [3, []]]), (8, [[4, []], [4, []]]))])\]

def tree(label, children=[]):
    return (label, children)

def label(tree):
    return tree[0]

def children(tree):
    return tree[1]

t = tree(20, [tree(12,
    [tree(9,
        [tree(7), tree(2)]),
        tree(3))],
    tree(8,
        [tree(4), tree(4)]))]}
A dictionary for each tree/subtree:

{{{1':20,'c':{{1':12,'c':{{1':9,'c':[]},{1':7,'c':[]},{1':2,'c':[]}}},{1':3,'c':[]}}},

{{{1':8,'c':{{1':4,'c':[]},{1':4,'c':[]}}}}}}

```python
def tree(label, children=[]):
    return {"l": label, "c": children}
def label(tree):
    return tree["l"]
def children(tree):
    return tree["c"]

```

t = tree(20, [tree(12,
    [tree(9, 
        [tree(7), tree(2)]),
    tree(3))],
    tree(8,
    [tree(4), tree(4)]))
```
Tree processing

A tree is a recursive structure.

Each tree has:

- A label
- 0 or more children, each a tree

Recursive structure implies recursive algorithm!
Tree processing: Counting leaves

```python
def count_leaves(t):
    """Returns the number of leaf nodes in T."""
    if
    else:
```

What's the base case? What's the recursive call?
def count_leaves(t):
    """Returns the number of leaf nodes in T."""
    if is_leaf(t):
        return 1
    else:
        return 0

What's the base case? What's the recursive call?
def count_leaves(t):
    """Returns the number of leaf nodes in T.""
    if is_leaf(t):
        return 1
    else:

What's the base case? What's the recursive call?
def count_leaves(t):
    
    if is_leaf(t):
        return 1
    else:
        children_leaves = 0
        for c in children(t):
            children_leaves += count_leaves(c)
        return children_leaves

What's the base case? What's the recursive call?
Tree processing: Counting leaves

The `sum()` function sums up the items of an iterable.

```python
>>> sum([1, 1, 1, 1])
4
```
Tree processing: Counting leaves

The `sum()` function sums up the items of an iterable.

```python
>>> sum([1, 1, 1, 1])
4
```

That leads to this shorter function:

```python
def count_leaves(t):
    """Returns the number of leaf nodes in T."""
    if is_leaf(t):
        return 1
    else:
        return sum([count_leaves(c) for c in children(t)])
```
Creating trees

A function that creates a tree from another tree is also often recursive.
def double(t):
    """Returns a tree identical to T, but with all labels doubled.""
    if
    else:

What's the base case? What's the recursive call?
Creating trees: Doubling labels

```python
def double(t):
    """Returns a tree identical to T, but with all labels doubled.""
    if is_leaf(t):
        # What's the base case?
    else:
        # What's the recursive call?
```

What's the base case? What's the recursive call?
Creating trees: Doubling labels

```
def double(t):
    """Returns a tree identical to T, but with all labels doubled."""
    if is_leaf(t):
        return tree(label(t) * 2)
    else:
```

What's the base case? What's the recursive call?
def `double(t)`:
    """Returns a tree identical to T, but with all labels doubled.""
    if `is_leaf(t)`:
        return tree(label(t) * 2)
    else:
        return tree(label(t) * 2,
                     [double(c) for c in children(t)])

What's the base case? What's the recursive call?
Creating trees: Doubling labels

Longer...

def double(t):
    """Returns a tree identical to T, but with all labels doubled."""
    if is_leaf(t):
        return tree(label(t) * 2)
    else:
        doubled_children = []
        for c in children(t):
            doubled_children.append(double(c))
        return tree(label(t) * 2, doubled_children)

Shorter!

def double(t):
    """Returns the number of leaf nodes in T."""
    return tree(label(t) * 2, [double(c) for c in children(t)])
Challenge: List of leaves

Try this on your own:

```python
def list_of_leaves(t):
    """Return a list containing the leaf labels of T."""

    >>> leaves(t)  # Using the t from the slides
    [7, 2, 3, 4, 4]
    """
    if ______:
        return ______
    else:
        ______
        return ______
```

Hint: If you sum a list of lists, you get a list containing the elements of those lists. The sum function takes a second argument, the starting value of the sum.
## Tree: Layers of abstraction

<table>
<thead>
<tr>
<th>Primitive Representation</th>
<th>1 2 3 True False</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(..,..) [..,..] {..}</td>
</tr>
</tbody>
</table>

### Data abstraction
- tree()
- children()
- label()
- is_leaf()

### User program
- double(t)
- count_leaves(t)

Each layer only uses the layer above it.
Trees, trees, everywhere!
Directory structures

- cs61a
  - homework
    - hw01
      - hw01.py ok
    - hw02
      - hw02.py ok
  - lab
    - lab01
      - lab01.py ok
    - lab02
      - lab02.py ok
  - projects
    - hog
      - hog.py ok
    - cats
      - cats.py ok
Parse trees

For natural languages...

\[
S \rightarrow NP \rightarrow D \rightarrow N \rightarrow V \rightarrow NP \\
\rightarrow A \rightarrow \text{mouse} \rightarrow \text{eats} \rightarrow a \rightarrow \text{cat}.
\]

Key: \( S \) = Sentence, \( NP \) = Noun phrase, \( D \) = Determiner, \( N \) = Noun, \( V \) = Verb, \( VP \) = Verb Phrase
Parse trees

For programming languages, too...

Key: E = expression