Dictionaries +
More Lists
Class outline:

- Dictionaries
- List diagrams
- Slicing
- Built-ins for iterables
- Recursive exercises
Dictionaries
A **dict** is a mapping of key-value pairs

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

Dictionaries support similar operations as lists/strings:

```python
>>> len(states)
4

>>> "CA" in states
True

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

A `dict` is a mapping of key-value pairs

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

Dictionaries support similar operations as lists/strings:

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

>>> "CA" in states
True

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

A **dict** is a mapping of key-value pairs

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

Dictionaries support similar operations as lists/strings:

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

>>> "CA" in states
True

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

A **dict** is a mapping of key-value pairs

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

Dictionaries support similar operations as lists/strings:

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

```python
>>> "CA" in states
True
```

```python
>>> "ZZ" in states
False
```
Dictionary access

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

Ways to access a value by key:

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

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

>>> words["pavo"]

>>> words.get("pavo", "")
```
Dictionary access

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

Ways to access a value by key:

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

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

>>> words["pavo"]

>>> words.get("pavo", "")
```
Dictionary access

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

Ways to access a value by key:

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

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

>>> words["pavo"]

>>> words.get("pavo", ")
```
Dictionary access

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

Ways to access a value by key:

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

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

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

```python
>>> words.get("pavo", "")
```

Dictionary access

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

Ways to access a value by key:

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

```python
>>> first_word = "aguа"
>>> words[first_word]
'water'
```

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

```python
>>> words.get("pavo", "")
''
```
Dictionary rules

- All keys in a dictionary are distinct (there can only be one value per key)
- A key **cannot** be a list or dictionary (or any other mutable type)
- 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])
```

What will be the order of items?
Dictionary iteration

```python
insects = {"spiders": 8, "centipedes": 100, "bees": 6}
for name in 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.
Dictionary comprehensions

General syntax:

```python
{key: value for <name> in <iter exp>}
```

Example:

```python
{x: x*x for x in range(3,6)}
```
Exercise: Prune

```python
def prune(d, keys):
    """Return a copy of D which only contains key/value pairs
    whose keys are also in KEYS.
    >>> prune({"a": 1, "b": 2, "c": 3, "d": 4}, ["a", "b", "c"])
    {"a": 1, "b": 2, "c": 3}
    """
```
def prune(d, keys):
    """Return a copy of D which only contains key/value pairs whose keys are also in KEYS."
    >>> prune({"a": 1, "b": 2, "c": 3, "d": 4}, ["a", "b", "c"])
    {'a': 1, 'b': 2, 'c': 3}
    """
    return {k: d[k] for k in keys}
Exercise: Index

```python
def index(keys, values, match):
    """Return a dictionary from keys k to a list of values v for which match(k, v) is a true value.

    >>> index([7, 9, 11], range(30, 50), lambda k, v: v % k == 0)
    {7: [35, 42, 49], 9: [36, 45], 11: [33, 44]}  
    """
```
Exercise: Index (solution)

def index(keys, values, match):
    """Return a dictionary from keys k to a list of values v for which
    match(k, v) is a true value.
    """

    >>> index([7, 9, 11], range(30, 50), lambda k, v: v % k == 0)
    {7: [35, 42, 49], 9: [36, 45], 11: [33, 44]}  
    """
    return {k: [v for v in values if match(k, v)] for k in keys}
## Nested data

Many useful way to combine lists and dicts:

| Lists of lists | [ [1, 2], [3, 4] ] |
| Dicts of dicts | {"name": "Brazilian Breads", "location": {"lat": 37.8, "lng": -122}} |
| Dicts of lists | {"heights": [89, 97], "ages": [6, 8]} |
| Lists of dicts | [{"title": "Ponyo", "year": 2009}, {"title": "Totoro", "year": 1993}] |
Slicing
Slicing syntax

Slicing a list creates a new list with a subsequence of the original list.

```
letters = ["A", "B", "C", "D", "E", "F"]
    #  0  1  2  3  4  5

sublist1 = letters[1:]
sublist2 = letters[1:4]
```

Slicing also works for strings.

```
compound_word = "cortaúñas"

word1 = compound_word[:5]
word2 = compound_word[5:]
```

Negatives indices and steps can also be specified.
Slicing syntax

Slicing a list creates a new list with a subsequence of the original list.

```
letters = ["A", "B", "C", "D", "E", "F"]
    #  0  1  2  3  4  5

sublist1 = letters[1:]    # ['B', 'C', 'D', 'E', 'F']
sublist2 = letters[1:4]
```

Slicing also works for strings.

```
compound_word = "cortaúñas"

word1 = compound_word[:5]
word2 = compound_word[5:]
```

Negatives indices and steps can also be specified.
Slicing syntax

Slicing a list creates a new list with a subsequence of the original list.

```python
letters = ["A", "B", "C", "D", "E", "F"]
          #  0  1  2  3  4  5
sublist1 = letters[1:]   # ['B', 'C', 'D', 'E', 'F']
sublist2 = letters[1:4]  # ['B', 'C', 'D']
```

Slicing also works for strings.

```python
compound_word = "cortaúñas"
word1 = compound_word[:5]
word2 = compound_word[5:]
```

Negatives indices and steps can also be specified.
Slicing syntax

Slicing a list creates a new list with a subsequence of the original list.

```
letters = ["A", "B", "C", "D", "E", "F"]
    #  0  1  2  3  4  5
sublist1 = letters[1:]    # ["B", "C", "D", "E", "F"]
```

Slicing also works for strings.

```
compound_word = "cortaúñas"

word1 = compound_word[:5]    # "corta"
word2 = compound_word[5:]
```

Negatives indices and steps can also be specified.
Slicing syntax

Slicing a list creates a new list with a subsequence of the original list.

```python
letters = ['A', 'B', 'C', 'D', 'E', 'F']
    # 0 1  2  3  4  5
sublist1 = letters[1:]    # ['B', 'C', 'D', 'E', 'F']
sublist2 = letters[1:4]    # ['B', 'C', 'D']
```

Slicing also works for strings.

```python
compound_word = "cortaúñas"

word1 = compound_word[:5]    # "corta"
word2 = compound_word[5:]    # "úñas"
```

Negatives indices and steps can also be specified.
Copying whole lists

Slicing a whole list copies a list:

```python
listA = [2, 3]
listB = listA

listC = listA[:]
listA[0] = 4
listB[1] = 5
```

`list()` creates a new list containing existing elements from any iterable:

```python
listA = [2, 3]
listB = listA

listC = list(listA)
listA[0] = 4
listB[1] = 5
```

Try both in PythonTutor.

Python3 provides more ways in the `copy` module.
Built-in functions for iterables
Functions that process iterables

The following built-in functions work for lists, strings, dicts, and any other **iterable** data type.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>sum(iterable, start)</code></td>
<td>Returns the sum of values in <strong>iterable</strong>, initializing sum to <strong>start</strong></td>
</tr>
<tr>
<td><code>all(iterable)</code></td>
<td>Return <strong>True</strong> if all elements of <strong>iterable</strong> are true (or if <strong>iterable</strong> is empty)</td>
</tr>
<tr>
<td><code>any(iterable)</code></td>
<td>Return <strong>True</strong> if any element of <strong>iterable</strong> is true. Return <strong>False</strong> if <strong>iterable</strong> is empty.</td>
</tr>
<tr>
<td><code>max(iterable, key=None)</code></td>
<td>Return the max value in <strong>iterable</strong></td>
</tr>
<tr>
<td><code>min(iterable, key=None)</code></td>
<td>Return the min value in <strong>iterable</strong></td>
</tr>
</tbody>
</table>
Examples with sum/any/all

```
sum([73, 89, 74, 95], 0)  # 331

all([True, True, True, True])
any([False, False, False, True])

all([x < 5 for x in range(5)])

perfect_square = lambda x: x == round(x ** 0.5) ** 2
any([perfect_square(x) for x in range(50, 60)])
```
Examples with sum/any/all

```python
sum([73, 89, 74, 95], 0)  # 331

all([True, True, True, True])  # True
any([False, False, False, True])

all([x < 5 for x in range(5)])

perfect_square = lambda x: x == round(x ** 0.5) ** 2
any([perfect_square(x) for x in range(50, 60)])
```

19
Examples with sum/any/all

```
sum([73, 89, 74, 95], 0)  # 331
```

```
all([True, True, True, True])  # True
any([False, False, False, True])  # True
```

```
all([x < 5 for x in range(5)])
```

```
perfect_square = lambda x: x == round(x ** 0.5) ** 2
any([perfect_square(x) for x in range(50, 60)])
```
Examples with sum/any/all

```
sum([73, 89, 74, 95], 0)  # 331

all([True, True, True, True])  # True
any([False, False, False, True])  # True

all([x < 5 for x in range(5)])  # True

perfect_square = lambda x: x == round(x ** 0.5) ** 2
any([perfect_square(x) for x in range(50, 60)])
```

Examples with sum/any/all

```
sum([73, 89, 74, 95], 0)  # 331

all([True, True, True, True])  # True
any([False, False, False, True])  # True

all([x < 5 for x in range(5)])  # True

perfect_square = lambda x: x == round(x ** 0.5) ** 2
any([perfect_square(x) for x in range(50, 60)])  # False
```
Examples with max/min

```python
max([73, 89, 74, 95])  # 95
max(['C+', 'B+', 'C', 'A'])
max(range(10))
```
Examples with max/min

max([73, 89, 74, 95])  # 95
max(['C+', 'B+', 'C', 'A'])  # C+
max(range(10))
Examples with max/min

```python
max([73, 89, 74, 95])  # 95
max(['C+', 'B+', 'C', 'A'])  # C+
max(range(10))  # 9
```
Examples with max/min

max([73, 89, 74, 95])  # 95
max(["C+", "B+", "C", "A"])  # C+
max(range(10))  # 9

A key function can decide how to compare each value:

coords = [ [37, -144], [-22, -115], [56, -163] ]
max(coords, key=lambda coord: coord[0])
min(coords, key=lambda coord: coord[0])

gymnasts = [ ["Brittany", 9.15, 9.4, 9.3, 9.2],
              ["Lea", 9, 8.8, 9.1, 9.5],
              ["Maya", 9.2, 8.7, 9.2, 8.8] ]
min(gymnasts, key=lambda scores: min(scores[1:]))
max(gymnasts, key=lambda scores: sum(scores[1:]), 0))
Examples with max/min

\[
\text{max([73, 89, 74, 95])} \quad \# \quad 95
\]
\[
\text{max(["C+", "B+", "C", "A"])} \quad \# \quad \text{C+}
\]
\[
\text{max(range(10))} \quad \# \quad 9
\]

A key function can decide how to compare each value:

\[
\text{coords = [ [37, -144], [-22, -115], [56, -163] ]}
\]
\[
\text{max(coords, key=\texttt{lambda coord: coord[0]})} \quad \# \quad \text{[56, -163]}
\]
\[
\text{min(coords, key=\texttt{lambda coord: coord[0]})}
\]

\[
\text{gymnasts = [ ["Brittany", 9.15, 9.4, 9.3, 9.2],
\quad ["Lea", 9, 8.8, 9.1, 9.5],
\quad ["Maya", 9.2, 8.7, 9.2, 8.8] ]}
\]
\[
\text{min(gymnasts, key=\texttt{lambda scores: min(scores[1:])})}
\]
\[
\text{max(gymnasts, key=\texttt{lambda scores: sum(scores[1:], 0))}
\]
Examples with max/min

```
max([73, 89, 74, 95])  # 95
max(['C+', 'B+', 'C', 'A'])  # C+
max(range(10))  # 9
```

A key function can decide how to compare each value:

```
coords = [ [37, -144], [-22, -115], [56, -163] ]
max(coords, key=lambda coord: coord[0])  # [56, -163]
min(coords, key=lambda coord: coord[0])  # [-22, -115]
```

```
gymnasts = [ ['Brittany', 9.15, 9.4, 9.3, 9.2],
             ['Lea', 9, 8.8, 9.1, 9.5],
             ['Maya', 9.2, 8.7, 9.2, 8.8] ]
min(gymnasts, key=lambda scores: min(scores[1:]))
max(gymnasts, key=lambda scores: sum(scores[1:], 0))
```
Examples with max/min

max([73, 89, 74, 95])  # 95
max(["C+", "B+", "C", "A"])  # C+
max(range(10))  # 9

A key function can decide how to compare each value:

cords = [ [37, -144], [-22, -115], [56, -163] ]
max(cords, key=lambda coord: coord[0])  # [56, -163]
min(cords, key=lambda coord: coord[0])  # [-22, -115]

gymnasts = [ ["Brittany", 9.15, 9.4, 9.3, 9.2],
            ["Lea", 9, 8.8, 9.1, 9.5],
            ["Maya", 9.2, 8.7, 9.2, 8.8] ]
min(gymnasts, key=lambda scores: min(scores[1:]))  # ["Maya", ..
max(gymnasts, key=lambda scores: sum(scores[1:], 0))
Examples with max/min

```python
max([73, 89, 74, 95])  # 95
max(["C+", "B+", "C", "A"])  # C+
max(range(10))  # 9
```

A key function can decide how to compare each value:

```python
coords = [ [37, -144], [-22, -115], [56, -163] ]
max(coords, key=lambda coord: coord[0])  # [56, -163]
min(coords, key=lambda coord: coord[0])  # [-22, -115]
```

```python
gymnasts = [ ["Brittany", 9.15, 9.4, 9.3, 9.2],
            ["Lea", 9, 8.8, 9.1, 9.5],
            ["Maya", 9.2, 8.7, 9.2, 8.8] ]
min(gymnasts, key=lambda scores: min(scores[1:]))  # ["Maya", ..
max(gymnasts, key=lambda scores: sum(scores[1:], 0))  # ["Brittany"
```
Recursion exercises
Helper functions

If a recursive function needs to keep track of more state than the arguments of the original function, you may need a helper function.

```python
def fUnKyCaSe(text):
    """
    >>> fUnKyCaSe("wats up")
    'wAtS Up'
    """

def toggle_case(letter, should_up_case):
    return letter.upper() if should_up_case else letter.lower()

def up_down(text, should_up_case):
    if len(text) == 1:
        return toggle_case(text, should_up_case)
    else:
        return toggle_case(text[0], should_up_case) + \
            up_down(text[1:], not should_up_case)

return up_down(text, False)
```
Recursively sum a list

Let's code this up recursively:

```python
def sum_nums(nums):
    '''Returns the sum of the numbers in NUMS.
    >>> sum_nums([6, 24, 1984])
    2014
    >>> sum_nums([-32, 0, 32])
    0
    '''
```

Docstrings typically would not specify whether an approach was recursive or iterative, since that is an implementation detail.

However, we'll make it clear in assignments and exam questions.
Recursively sum a list (solution)

```python
def sum_nums(nums):
    """Returns the sum of the numbers in NUMS."
    >>> sum_nums([6, 24, 1984])
    2014
    >>> sum_nums([-32, 0, 32])
    0
    """
    if nums == []:
        return 0
    else:
        return nums[0] + sum_nums(nums[1:])
```

When recursively processing lists, the base case is often the empty list and the recursive case is often all-but-the-first items.
Recursively reversing a string

def reverse(s):
    """Returns a string with the letters of S in the inverse order.
    >>> reverse('ward')
    'draw'
    """

Breaking it down into subproblems:

reverse("ward") =
reverse("ard") =
reverse("rd") =
reverse("d") =
Recursively reversing a string

```python
def reverse(s):
    """Returns a string with the letters of S in the inverse order.
    >>> reverse('ward')
    'draw'
    """
```

Breaking it down into subproblems:

```python
reverse("ward") = reverse("ard") + "w"
reverse("ard") = reverse("rd") + "a"
reverse("rd") = reverse("d") + "r"
reverse("d") = 
```
Recursively reversing a string

def reverse(s):
    """Returns a string with the letters of S in the inverse order.
    >>> reverse('ward')
    'draw'
    """

Breaking it down into subproblems:

reverse("ward") = reverse("ard") + "w"
reverse("ard") = reverse("rd") + "a"
reverse("rd") = reverse("d") + "r"
reverse("d") = "d"
def reverse(s):
    """Returns a string with the letters of S in the inverse order.
    >>> reverse('ward')
    'draw'
    """
    if len(s) == 1:
        return s
    else:
        return reverse(s[1:]) + s[0]

When recursively processing strings, the base case is typically an empty string or single-character string, and the recursive case is often all-but-the-first characters.
Recursively reversing a string (visual)
Exercise: Reversing a number

def reverse(n):
    """Returns N with the digits reversed.
    >>> reverse_digits(123)
    321
    """

See walkthrough video here
## Recursion on different data types

<table>
<thead>
<tr>
<th>Data type</th>
<th>Base case condition</th>
<th>Current item</th>
<th>Recursive case argument</th>
</tr>
</thead>
<tbody>
<tr>
<td>Numbers</td>
<td>== 0</td>
<td>n</td>
<td>n - 1</td>
</tr>
<tr>
<td></td>
<td>== 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Numbers</td>
<td>== 0 &lt; 10</td>
<td>n % 10</td>
<td>n // 10</td>
</tr>
<tr>
<td>(Digits)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Lists</td>
<td>== []</td>
<td>L[0]</td>
<td>L[1:]</td>
</tr>
<tr>
<td></td>
<td>len(L) == 0</td>
<td>L[-1]</td>
<td>L[::1]</td>
</tr>
<tr>
<td>Strings</td>
<td>== ''</td>
<td>S[0]</td>
<td>S[1:]</td>
</tr>
<tr>
<td></td>
<td>len(S) == 1</td>
<td></td>
<td>S[::1]</td>
</tr>
</tbody>
</table>
List diagrams
Lists in environment diagrams

Lists are represented as a row of index-labeled adjacent boxes, one per element.

pair = [1, 2]

Try in PythonTutor.
Nested lists in environment diagrams

Each box either contains a primitive value or points to a compound value.

```
matrix = [[1, 2, 0, 4], [0, 1, 3, -1], [0, 0, 1, 8]]
```
Nested lists in environment diagrams

A very nested list:

```python
worst_list = [ [1, 2], [], [ [3, False, None], [4, lambda: 5]]]
```

View in PythonTutor
Python Project of The Day!
Sea Level Rise

Sea Level Rise, by Douwe Osinga: Visualize sea levels and population density on interactive maps.

Technologies used: Python (notebook) with PIL/numpy/Rasterio, HTML/CSS/JS with PanZoom (Github repository)