# Programming Principles in Python (CSCI 503)

## Dictionaries & Sets

Dr. David Koop

(some slides adapted from Dr. Reva Freedman)

# Sequences

- Strings `"abcde"`, Lists `[1, 2, 3, 4, 5]`, and Tuples `(1, 2, 3, 4, 5)`

- Defining a list: `my_list = [0, 1, 2, 3, 4]`
- But lists can store different types:
  - `my_list = [0, "a", 1.34]`
- Including other lists:
  - `my_list = [0, "a", 1.34, [1, 2, 3]]`
- Others are similar: tuples use parenthesis, strings are delineated by quotes (single or double)

# Sequence Operations

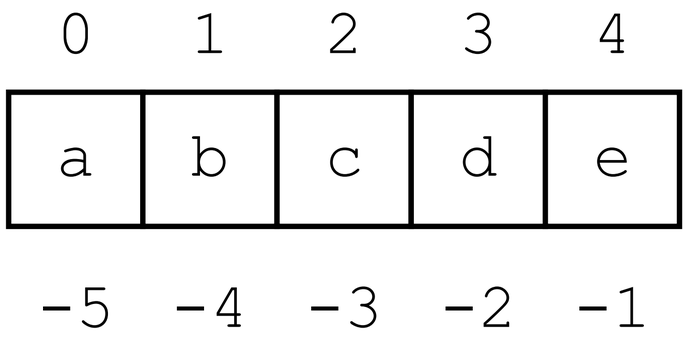
- Concatenate: `[1, 2] + [3, 4] # [1,2,3,4]`
- Repeat: `[1,2] * 3 # [1,2,1,2,1,2]`
- Length: `my_list = [1,2]; len(my_list) # 2`

- Concatenate: `(1, 2) + (3, 4) # (1,2,3,4)`
- Repeat: `(1,2) * 3 # (1,2,1,2,1,2)`
- Length: `my_tuple = (1,2); len(my_tuple) # 2`

- Concatenate: `"ab" + "cd" # "abcd"`
- Repeat: `"ab" * 3 # "ababab"`
- Length: `my_str = "ab"; len(my_str) # 2`

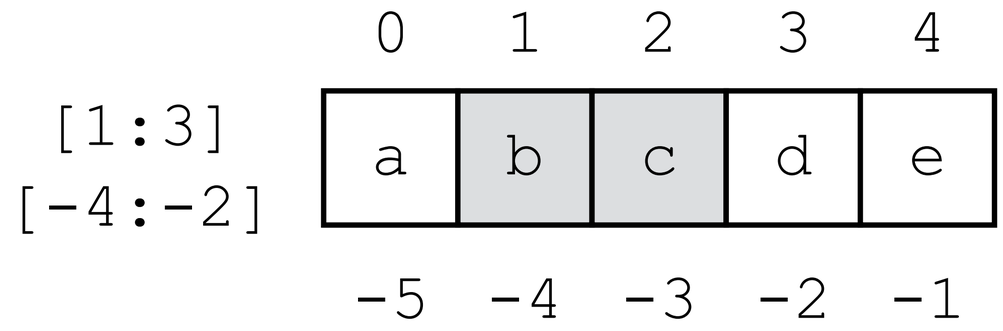
# Indexing (Positive and Negative)

- Positive indices start at zero, negative at -1
- `my_str = "abcde"; my_str[1] # "b"`
- `my_list = [1,2,3,4,5]; my_list[-3] # 3`
- `my_tuple = (1,2,3,4,5); my_tuple[-5] # 1`



# Slicing

- Positive or negative indices can be used at any step
- `my_str = "abcde"; my_str[1:3] # ["b", c"]`
- `my_list = [1,2,3,4,5]; my_list[3:-1] # [4]`
- Implicit indices
  - `my_tuple = (1,2,3,4,5); my_tuple[-2:] # (4,5)`
  - `my_tuple[:3] # (1,2,3)`



# Iteration

- `for d in sequence:`
  `    # do stuff`
- **Important**: `d` is a **data** item, **not** an **index**!
- `sequence = "abcdef"`
  `for d in sequence:`
  `    print(d, end=" ")  # a b c d e f`
- `sequence = [1,2,3,4,5]`
  `for d in sequence:`
  `    print(d, end=" ")  # 1 2 3 4 5`
- `sequence = (1,2,3,4,5)`
  `for d in sequence:`
  `    print(d, end=" ")  # 1 2 3 4 5`

# Sequence Operations

| Operator | Meaning |
|---|---|
| `<seq> + <seq>` | Concatenation |
| `<seq> * <int-expr>` | Repetition |
| `<seq>[<int-expr>]` | Indexing |
| `len(<seq>)` | Length |
| `<seq>[<int-expr?>:<int-expr?>]` | Slicing |
| `for <var> in <seq>:` | Iteration |
| `<expr> in <seq>` | Membership (Boolean) |

# Sequence Operations

| Operator | Meaning |
|---|---|
| `<seq> + <seq>` | Concatenation |
| `<seq> * <int-expr>` | Repetition |
| `<seq>[<int-expr>]` | Indexing |
| `len(<seq>)` | Length |
| `<seq>[<int-expr?>:<int-expr?>]` | Slicing |
| `for <var> in <seq>:` | Iteration |
| `<expr> in <seq>` | Membership (Boolean) |

`<int-expr?>`: may be `<int-expr>` but also can be empty

# List methods

| Method | Meaning |
|---|---|
| `<list>.append(d)` | Add element `d` to end of list. |
| `<list>.extend(s)` | Add **all** elements in `s` to end of list. |
| `<list>.insert(i, d)` | Insert `d` into list at index i. |
| `<list>.pop(i)` | Deletes `i`th element of the list and returns its value. |
| `<list>.sort()` | Sort the list. |
| `<list>.reverse()` | Reverse the list. |
| `<list>.remove(d)` | Deletes first occurrence of `d` in list. |
| `<list>.index(d)` | Returns index of first occurrence of `d`. |
| `<list>.count(d)` | Returns the number of occurrences of `d` in list. |

# List methods

| Method | Meaning | Mutate |
|---|---|---|
| `<list>.append(d)` | Add element `d` to end of list. | |
| `<list>.extend(s)` | Add **all** elements in `s` to end of list. | |
| `<list>.insert(i, d)` | Insert `d` into list at index i. | |
| `<list>.pop(i)` | Deletes `i`th element of the list and returns its value. | |
| `<list>.sort()` | Sort the list. | |
| `<list>.reverse()` | Reverse the list. | |
| `<list>.remove(d)` | Deletes first occurrence of `d` in list. | |
| `<list>.index(d)` | Returns index of first occurrence of `d`. | |
| `<list>.count(d)` | Returns the number of occurrences of `d` in list. | |

# Updating collections

- There are three ways to deal with operations that update collections:
  - Returns an **updated copy** of the collection
  - Updates the collection **in place**
  - Updates the collection in place **and returns it**
- `list.sort` and `list.reverse` work **in place** and **don't return** it
- `sorted` and `reversed` return an **updated copy**
  - these also work for immutable sequences like strings and tuples

# Assignment 3

- Coming soon…

# enumerate

- Often you **do not** need the index when iterating through a sequence
- If you need an index while looping through a sequence, use `enumerate`
- ```
  for i, d in enumerate(my_list):
      print("index:", i, "element:", d)
  ```
- Each time through the loop, it yields **two** items, the **index** `i` & the **element** `d`
- `i, d` is actually a **tuple**
- Automatically **unpacked** above, can manually do this, but don't!
- ```
  for t in enumerate(my_list):
      i = t[0]
      d = t[1]
      print("index:", i, "element:", d)
  ```

# enumerate

- Often you **do not** need the index when iterating through a sequence
- If you need an index while looping through a sequence, use `enumerate`
- 
```
for i, d in enumerate(my_list):
    print("index:", i, "element:", d)
```
- Each time through the loop, it yields **two** items, the **index** `i` & the **element** `d`
- `i, d` is actually a **tuple**
- Automatically **unpacked** above, can manually do this, but don't!
- 
```
for t in enumerate(my_list):
    i = t[0]
    d = t[1]
    print("index:", i, "element:", d)
```

# Tuples

- Tuples are immutable sequences
- We've actually seen tuples a few times already
  - Simultaneous Assignment
  - Returning Multiple Values from a Function
- Python allows us to omit parentheses when it's clear
  - `b, a = a, b` `# same as (b, a) = (a, b)`
  - `t1 = a, b` `# don't normally do this`
  - `c, d = f(2, 5, 8)` `# same as (c, d) = f(2, 5, 8)`
  - `t2 = f(2, 5, 8)` `# don't normally do this`

# Packing and Unpacking

- ```
  def f(a, b):
      if a > 3:
          return a, b-a # tuple packing
      return a+b, b # tuple packing
  ```
- `c, d = f(4, 3) # tuple unpacking`

- Make sure to unpack the correct number of variables!
- `c, d = a+b, a-b, 2*a # ValueError: too many values to unpack`
- Sometimes, check return value before unpacking:

```
- retval = f(42)
  if retval is not None:
      c, d = retval
```
```

# Packing and Unpacking

- ```
  def f(a, b):
      if a > 3:
          return a, b-a # tuple packing
      return a+b, b # tuple packing
  ```
- `c, d = f(4, 3) # tuple unpacking`

```
t = (a, b-a)
return t
```

- Make sure to unpack the correct number of variables!
- `c, d = a+b, a-b, 2*a # ValueError: too many values to unpack`
- Sometimes, check return value before unpacking:

```
- retval = f(42)
  if retval is not None:
      c, d = retval
```
```

# Packing and Unpacking

- ```
  def f(a, b):
      if a > 3:
          return a, b-a # tuple packing
      return a+b, b # tuple packing
  ```
- ```
  c, d = f(4, 3) # tuple unpacking
  ```

```
t = (a, b-a)
return t
```

```
t = f(4, 3)
(c, d) = t
```

- Make sure to unpack the correct number of variables!
- `c, d = a+b, a-b, 2*a # ValueError: too many values to unpack`
- Sometimes, check return value before unpacking:

```
- retval = f(42)
  if retval is not None:
      c, d = retval
```

# Unpacking other sequences

- You can unpack other sequences, too
  - `a, b = 'ab'`
  - `a, b = ['a', 'b']`
- Why is list unpacking rare?

# Other sequence methods

- `my_list = [7, 2, 1, 12]`
- Math methods:
  - `max(my_list) # 12`
  - `min(my_list) # 1`
  - `sum(my_list) # 22`
- `zip`: combine two sequences into a single sequence of tuples
  - ```
    zip_list = list(zip(my_list, "abcd"))
    zip_list # [(1, 'a'), (2, 'b'), (7, 'c'), (12, 'd')]
    ```
  - Use this instead of using indices to count through both
    ```

# Dictionaries

# Dictionary

- AKA associative array or map
- Collection of key-value pairs
  - Keys must be unique
  - Values need not be unique
- Syntax:
  - Curly brackets {} delineate start and end
  - Colons separate keys from values, commas separate pairs
  - `d = {'DeKalb': 783, 'Kane': 134, 'Cook': 1274, 'Will': 546}`
- No type constraints
  - `d = {'abc': 25, 12: 'abc', ('Kane', 'IL'): 123.54}`

# Dictionary Examples

| Keys | Key type | Values | Value type |
|---|---|---|---|
| Country names | `str` | Internet country | `str` |
| Decimal numbers | `int` | Roman numerals | `str` |
| States | `str` | Agricultural | `list of str` |
| Hospital patients | `str` | Vital signs | `tuple of floats` |
| Baseball players | `str` | Batting averages | `float` |
| Metric | `str` | Abbreviations | `str` |
| Inventory codes | `str` | Quantity in stock | `int` |

[Deitel & Deitel]

# Collections

- A dictionary is **not** a sequence
- Sequences are **ordered**
- Conceptually, dictionaries need no order
- A dictionary is a **collection**
- Sequences are also collections
- All collections have length (`len`), membership (`in`), and iteration (loop over values)
- Length for dictionaries counts number of key-value **pairs**
  - Pass dictionary to the `len` function
  - ```
    d = {'abc': 25, 12: 'abc', ('Kane', 'IL'): 123.54}
    len(d) # 3
    ```
    ```

# Mutability

- Dictionaries are **mutable**, key-value pairs can be added, removed, updated
- (Each key must be immutable)
- Accessing elements parallels lists but with different "indices" possible
- Index → Key
- `d = {'DeKalb': 783, 'Kane': 134, 'Cook': 1274, 'Will': 546}`
- `d['Winnebago'] = 1023 # add a new key-value pair`
- `d['Kane'] = 342 # update an existing key-value pair`
- `d.pop('Will') # remove an existing key-value pair`
- `del d['Winnebago'] # remove an existing key-value pair`

# Key Restrictions

- Many types can be keys… including tuples
  - `d = {'abc': 25, 12: 'abc', ('Kane', 'IL'): 123.54}`
- …but the type must be immutable—lists cannot be keys
  - ~~`d = {['Kane', 'IL']: 2348.35, [1, 2, 3]: "apple"}`~~
- Why?

# Key Restrictions

- Many types can be keys… including tuples
  - `d = {'abc': 25, 12: 'abc', ('Kane', 'IL'): 123.54}`
- …but the type must be **immutable*** — lists cannot be keys
  - ~~`d = {['Kane', 'IL']: 2348.35, [1, 2, 3]: "apple"}`~~
- *technically, the type must be hashable, but having a mutable and still hashable type almost always causes problems
- Why?
  - Dictionaries are fast in Python because are implemented as hash tables
  - No matter how long the key, python hashes it stores values by hash
  - Given a key to lookup, Python hashes it and finds the value quickly (O(1))
  - If the key can mutate, the hash will not match the key!

# Principle

- Be careful using floats for keys
- Why?

# Principle

- Be careful using floats for keys
- ```
  a = 0.123456
  b = 0.567890

  values = [a, b, (a / b) * b, (b / a) * a]
  found = {}
  for d in values:
      found[d] = True
  len(found) # 3 !!!
  found.keys() # [0.123456, 0.56789, 0.12345599999999998]
  ```
  ```

# Accessing Values by Key

- To get a value, we start with a key
- Things work as expected
  - `d['Kane'] + d['Cook']`
- If a value does not exist, get KeyError
  - `d['Boone'] > 12 # KeyError`

# Membership

- The membership operator (`in`) applies to **keys**
  - `'Boone' in d # False`
  - `'Cook' in d # True`
- To check the negation (if a key doesn't exist), use `not in`
  - `'Boone' not in d # True`
  - `not 'Boone' in d # True (equivalent but less readable)`
- Membership testing is much **faster** than for a list
- Checking and accessing at once
  - `d.get('Boone')    # no error, evaluates to None`
  - `d.get('Boone', 0) # no error, evaluates to 0 (default)`

# Updating multiple key-value pairs

- Update adds or replaces key-value pairs
- Update from another dictionary:
  - `d.update({'Winnebago': 1023, 'Kane': 324})`
- Update from a list of key-value tuples
  - `d.update([('Winnebago', 1023), ('Kane', 324)])`
- Update from keyword arguments
  - `d.update(Winnebago=1023, Kane=324)`
  - Only works for strings!
- Syntax for update also works for constructing a **new** dictionary
  - `d = dict([('Winnebago', 1023), ('Kane', 324)])`
  - `d = dict(Winnebago=1023, Kane=324)`

# Dictionary Methods

| Method | Meaning |
|---|---|
| `<dict>.clear()` | Remove all key-value pairs |
| `<dict>.update(other)` | Updates the dictionary with values from `other` |
| `<dict>.pop(k, d=None)` | Removes the pair with key `k` and returns value or default `d` if no key |
| `<dict>.get(k, d=None)` | Returns the value for the key `k` or default `d` if no key |
| `<dict>.items()` | Returns iterable view over all pairs as (key, value) tuples |
| `<dict>.keys()` | Returns iterable view over all keys |
| `<dict>.values()` | Returns iterable view over all values |

# Dictionary Methods

| Method | Meaning | Mutate |
|---|---|---|
| `<dict>.clear()` | Remove all key-value pairs | |
| `<dict>.update(other)` | Updates the dictionary with values from `other` | |
| `<dict>.pop(k, d=None)` | Removes the pair with key `k` and returns value or default `d` if no key | |
| `<dict>.get(k, d=None)` | Returns the value for the key `k` or default `d` if no key | |
| `<dict>.items()` | Returns iterable view over all pairs as (key, value) tuples | |
| `<dict>.keys()` | Returns iterable view over all keys | |
| `<dict>.values()` | Returns iterable view over all values | |

# Iteration

- Even though dictionaries are not sequences, we can still iterate through them
- Principle: Don't depend on order
- ```
  for k in d:
      print(k, end=" ")
  ```
- This only iterates through the **keys**!
- We could get the values:
- ```
  for k in d:
      print('key:', k, 'value:', d[k], end=" ")
  ```
- …but this is kind of like counting through a sequence (not pythonic)

# Dictionary Views

- `for k in d.keys():  # iterate through keys`
  `    print('key:', k)`
- `for v in d.values():  # iterate through values`
  `    print('value:', v)`
- `for k, v in d.items(): # iterate through key-value pairs`
  `    print('key:', k, 'value:', v)`
- `keys()` is superfluous but is a bit clearer
- `items()` is the enumerate-like method

# Exercise: Count Letters

- Write code to take a string and return the count the number of each letter that occurs in a dictionary
- count_letters('illinois') # returns {'i': 3, 'l': 2, 'n': 1, 'o': 1, 's': 1}

# Exercise: Count Letters

```
def count_letters(s):
    d = {}
    for c in s:
        if c not in d:
            d[c] = 1
        else:
            d[c] += 1
    return d
count_letters('illinois')
```

# Exercise: Count Letters

```
def count_letters(s):
    d = {}
    for c in s:
        d[c] = d.get(c, 0) + 1
    return d
count_letters('illinois')
```

# Exercise: Count Letters (using collections)

# Exercise: Count Letters (using collections)

```
from collections import defaultdict
def count_letters(s):
    d = defaultdict(int)
    for c in s:
        d[c] += 1
    return d
count_letters('illinois')
```

# Exercise: Count Letters (using collections)

- ```
  from collections import defaultdict
  def count_letters(s):
      d = defaultdict(int)
      for c in s:
          d[c] += 1
      return d
  count_letters('illinois')
  ```

- ```
  from collections import Counter
  def count_letters(s):
      return Counter(s)
  count_letters('illinois')
  ```

# Sorting

- Order doesn't really mean anything in a dictionary
- There is no .sort or .reverse method
- We can iterate through items in sorted order using sorted
- ```
  d = count_letters('illinois')
  for k, v in sorted(d.items()):
      print(k, ':', v)
  ```
- reversed also works on dictionary views
- sorted and reversed work on any iterable (thus all collections)
  ```

# Sets

# Sets

- Sets are dictionaries but without the values
- Same curly braces, no pairs
- `s = {'DeKalb', 'Kane', 'Cook', 'Will'}`
- Only one instance of a value is in a set—sets **eliminate duplicates**
- Adding multiple instances of the same value to a set doesn't do anything
- ```
  s = {'DeKalb', 'DeKalb', 'DeKalb', 'Kane', 'Cook', 'Will'}
      # {'Cook', 'DeKalb', 'Kane', 'Will'}
  ```
- Watch out for the empty set
  - `s = {} # not a set!`
  - `s = set() # an empty set`
  ```

# Sets are Mutable Collections

- Sets are **mutable** like dictionaries: we can add, replace, and delete
- Again, no type constraints
  - s = {12, 'DeKalb', 22.34}
- Like a dictionary, a set is a **collection** but not a sequence
- Q: What three things can we do for any collection?

# Collection Operations on Sets

- `s = {'DeKalb', 'Kane', 'Cook', 'Will'}`
- Length
  - `len(s) # 4`
- Membership: fast just like dictionaries
  - `'Kane' in s # True`
  - `'Winnebago' not in s # True`
- Iteration
  - 
```
for county in s:
    print(county)
```