Data structures
and functions

Instructor: Jorge Mendez
Logistics

• Everyone should be signed up for Piazza by now
  • Otherwise, please sign yourself up at
    [https://www.piazza.com/upenn/spring2020/cis192202](https://www.piazza.com/upenn/spring2020/cis192202)

• Remember that slides, examples, and homeworks will be available at the course website: [https://www.cis.upenn.edu/~cis192/jorge/](https://www.cis.upenn.edu/~cis192/jorge/)

• First homework will come out this Friday and will be due one week after that.
What we’ll cover today

• Lists
• Tuples
• Range
• Sets
• Dictionaries
• Strings
• Functions
Lists
List basics

• Built-in collection of objects
• Mutable: its elements *can* be modified
• Arbitrarily typed elements (even differently typed)
  • Usually elements are of the same type
• Typically implemented as dynamic arrays (like `ArrayList` in Java)
• Constructors:
  • `[a, b, c]` — comma separated values, potentially empty
  • `list(iterable)` — e.g., `list(range(5))`
  • `[x for x in iterable]` — list comprehension, more later
List comprehension

- Create lists with simple computations
- “… brackets containing an expression followed by a for clause, then zero or more for or if clauses”
  - [https://docs.python.org/3/tutorial/datastructures.html#list-comprehensions](https://docs.python.org/3/tutorial/datastructures.html#list-comprehensions)
Live example
Basic list operations

- Running example: \( a = [3, 1, 4, 5] \)
- \( \text{len}(a) \) — returns the number of elements in \( a \): 4
- \( a[\text{idx}] \) — returns the \( \text{idx} \)-th element of \( a \), using 0-based indexing
  - \( \text{idx} \) must be between 0 and \( \text{len}(a) - 1 \)
  - \( a[2] \rightarrow 4 \)
- \( a[-\text{idx}] \) — returns the \( \text{idx} \)-th element of \( a \) from the right, using 1-based indexing
  - Intuitively means \( \text{len}(a) - \text{idx} \), hence 1-based
  - \( \text{idx} \) must be between 1 and \( \text{len}(a) \)
  - \( a[-1] \rightarrow 5 \)
List slicing

- Running example: \( a = [3, 1, 4, 5] \)
- Slices are **copies** of (not necessarily contiguous) subarrays
  - \( a[idx_0 : idx_f] \) — returns a slice of \( a \) from \( idx_0 \) (inclusive) to \( idx_f \) (non-inclusive). Both indices are optional.
    - Defaults: \( idx_0 = 0, idx_f = \text{len}(a) \)
    - \( a[1:3] \rightarrow [1, 4] \)
    - \( a[1:] \rightarrow [1, 4, 5] \)
    - \( a[:3] \rightarrow [3, 1, 4] \) (equivalently \( a[\ldots-1] \))
    - \( a[:3] \rightarrow [3, 1, 4, 5] \) (copy)
  - \( a[idx_0 : idx_f : \text{step}] \) — as above, but in steps of \( \text{step} \)
    - \( a[0:3:2] \rightarrow [3, 4] \)
    - \( a[-2:-5:-2] \rightarrow [4,3] \)
    - \( a[::-1] \rightarrow [5, 4, 1, 3] \)
- Support slice assignment: requires assigned array to be the same size as the slice
  - \( a[0:3:2] = [0,0] \rightarrow [0, 1, 0, 5] \)
## List implementation

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
<th>Runtime</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>len(a)</td>
<td>Number of elements</td>
<td>O(1)</td>
<td></td>
</tr>
<tr>
<td>a[idx_0:idx_f]</td>
<td>Slice</td>
<td>O(idx_f - idx_0)</td>
<td>O(1) for single element</td>
</tr>
<tr>
<td>a + b</td>
<td>Concatenate lists</td>
<td>O(n+m)</td>
<td>+= for concatenate update</td>
</tr>
<tr>
<td>a * k</td>
<td>Repeat list k times</td>
<td>O(kn)</td>
<td>*= for repeat update</td>
</tr>
<tr>
<td>x in a</td>
<td>True if some element in a has value x</td>
<td>O(n)</td>
<td></td>
</tr>
<tr>
<td>min(a), max(a)</td>
<td>Minimum and maximum values</td>
<td>O(n)</td>
<td></td>
</tr>
<tr>
<td>a.index(x)</td>
<td>Position of first element with value x</td>
<td>O(n)</td>
<td>Linear search</td>
</tr>
<tr>
<td>a.count(x)</td>
<td>Number of elements with value x</td>
<td>O(n)</td>
<td></td>
</tr>
<tr>
<td>del a[idx_0:idx_f]</td>
<td>Remove slice from a</td>
<td>O(n)</td>
<td>O(n) even for single element</td>
</tr>
<tr>
<td>a.sort()</td>
<td>Sort list in place</td>
<td>O(n logn)</td>
<td></td>
</tr>
<tr>
<td>a.pop(-idx)</td>
<td>Delete the idx-th element</td>
<td>O(idx)</td>
<td>O(1) for last element (default)</td>
</tr>
<tr>
<td>a.append(x)</td>
<td>Add element with value x at the end</td>
<td>O(1)</td>
<td></td>
</tr>
</tbody>
</table>
Live example
Lists as stacks

• init — `a = []`
• isEmpty — `len(a) == 0`
• isFull — N/A
• push — `a.append(x)`
• pop — `a.pop()`
• All operations are O(1)

• For FIFO queue, use `collections.deque`
  • [https://docs.python.org/3/library/collections.html#collections.deque](https://docs.python.org/3/library/collections.html#collections.deque)
Tuples
Tuple basics

• Built-in collection of objects
• Immutable: its elements are fixed after creation
• Arbitrarily typed elements (even differently typed)
  • Common to have multiple types in a tuple
  • E.g., a number and associated name (‘a’, 1)
• Typically implemented as a static array (Array in Java)
• Constructors:
  • () — empty tuple
  • (a,) — singleton tuple
  • (a, b, c) or a, b, c — multiple elements
  • tuple(iterable) — created from iterable, order preserved
Notes on tuples

• They are faster than lists
  • Use them if you won’t need to modify it at runtime!

• There is no tuple comprehension
  • \((x \text{ for } x \text{ in } \text{range}(3))\) just creates a generator, not a tuple

• Support all list operations for accessing, but not for altering
## Tuple implementation

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
<th>Runtime</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>len(a)</code></td>
<td>Number of elements</td>
<td>$O(1)$</td>
<td></td>
</tr>
<tr>
<td><code>a[idx_0:idx_f]</code></td>
<td>Slice</td>
<td>$O(idx_f - idx_0)$</td>
<td>$O(1)$ for single element</td>
</tr>
<tr>
<td><code>a + b</code></td>
<td>Concatenate tuples</td>
<td>$O(n+m)$</td>
<td></td>
</tr>
<tr>
<td><code>a * k</code></td>
<td>Repeat tuple $k$ times</td>
<td>$O(kn)$</td>
<td></td>
</tr>
<tr>
<td><code>x in a</code></td>
<td>True if some element in <code>a</code> has value $x$</td>
<td>$O(n)$</td>
<td></td>
</tr>
<tr>
<td><code>min(a), max(a)</code></td>
<td>Minimum and maximum values</td>
<td>$O(n)$</td>
<td>Linear search</td>
</tr>
<tr>
<td><code>a.index(x)</code></td>
<td>Position of first element with value $x$</td>
<td>$O(n)$</td>
<td></td>
</tr>
<tr>
<td><code>a.count(x)</code></td>
<td>Number of elements with value $x$</td>
<td>$O(n)$</td>
<td></td>
</tr>
<tr>
<td><code>sorted(a)</code>*</td>
<td>Sort tuple not in place</td>
<td>$O(n \log n)$</td>
<td></td>
</tr>
</tbody>
</table>

*Returns sorted list. Compare to lists’ `a.sort()` in place method.
Range
Range basics

- Built-in collection of \texttt{int} objects
- Immutable
- Its size is \(O(1)\)

**Constructors**
- \texttt{range(stop)} — from 0 to \(\text{stop}-1\)
- \texttt{range(start, stop)} — from \(\text{start}\) to \(\text{stop}-1\)
- \texttt{range(start, stop, step)} — from \(\text{start}\) to \(\text{stop}-1\) on steps of \(\text{step}\)

- Like tuples, support accessing operations
  - Does \textit{not} support the repetition (\(*\)) operator
## Range implementation

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
<th>Runtime</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>len(a)</code></td>
<td>Number of elements</td>
<td>O(1)</td>
<td></td>
</tr>
<tr>
<td><code>a[idx_0:idx_f]</code></td>
<td>Slice</td>
<td>O(idx_f - idx_0)</td>
<td>O(1) for single element</td>
</tr>
<tr>
<td><code>a + b</code></td>
<td>Concatenate tuples</td>
<td>O(n+m)</td>
<td></td>
</tr>
<tr>
<td><code>x in a</code></td>
<td>True if some element in a has value x</td>
<td>O(n)</td>
<td>Linear search</td>
</tr>
<tr>
<td><code>min(a), max(a)</code></td>
<td>Minimum and maximum values</td>
<td>O(n)</td>
<td></td>
</tr>
<tr>
<td><code>a.index(x)</code></td>
<td>Position of first element with value x</td>
<td>O(n)</td>
<td></td>
</tr>
<tr>
<td><code>a.count(x)</code></td>
<td>Number of elements with value x</td>
<td>O(n)</td>
<td></td>
</tr>
<tr>
<td><code>sorted(a)</code></td>
<td>Sort range not in place</td>
<td>O(n logn)</td>
<td></td>
</tr>
</tbody>
</table>

*Returns sorted list. Compare to lists’ `a.sort()` in place method*
Sequences

- Types we’ve seen (list, tuple, range) are sequence types
  - We will see another sequence type, string, later today
- Support for common indexing and slicing operations
- Sequence comparison is done in lexicographical order
  - Compare first element, if equal move to second, and so on
  - If an element is itself a sequence, compare recursively
  - Sequences must be of the same type (e.g., two lists)
- You can implement your own sequence class
  - More on this later in the course!
Live example
Looping through sequences

• Use sequence as iterable if all you need is the elements
• If you need index and value, use `enumerate()`
  • returns `(idx, element)` tuples
• If you need to iterate over two or more sequences in parallel, use `zip()`
  • returns `(seq1[idx], seq2[idx], ..., seqn[idx])` tuples

https://docs.python.org/3/tutorial/datastructures.html#looping-techniques
Live example
Set basics

• Mutable collection of objects with no repeated elements
• No ordering imposed
• Implemented as a hash set
• Supports only hashable types, but may contain multiple types
  • Mutable types are not hashable
• Constructors
  • `set()` — an empty set (cannot use `{}`, reserved for dictionaries)
  • `set(iterable)` — non-repeated elements, order not preserved
  • `{a,b,c}` — set from elements, ignore repeated
  • `{x for x in iterable}` — set comprehension, just like lists
## Set implementation

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
<th>Runtime</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>len(a)</code></td>
<td>Number of elements</td>
<td>O(1)</td>
<td></td>
</tr>
<tr>
<td><code>x in a</code></td>
<td>True if some element in <code>a</code> has value <code>x</code></td>
<td>O(1)</td>
<td></td>
</tr>
<tr>
<td><code>a.add(x)</code></td>
<td>Insert element if not repeated</td>
<td>O(1)</td>
<td>Hashing</td>
</tr>
<tr>
<td><code>a.remove(x)</code></td>
<td>Remove element, error if not present</td>
<td>O(1)</td>
<td></td>
</tr>
<tr>
<td><code>a.discard(x)</code></td>
<td>Remove element if present</td>
<td>O(1)</td>
<td></td>
</tr>
<tr>
<td><code>a &lt;= b</code></td>
<td>True if <code>a</code> is improper subset of <code>b</code></td>
<td>O(n)</td>
<td>&lt; for proper</td>
</tr>
<tr>
<td><code>a &gt;= b</code></td>
<td>True if <code>a</code> is improper superset of <code>b</code></td>
<td>O(m)</td>
<td>&gt; for proper</td>
</tr>
<tr>
<td><code>a.isdisjoint(b)</code></td>
<td>True if no element of <code>a</code> is in <code>b</code></td>
<td>O(min(n,m))</td>
<td></td>
</tr>
<tr>
<td>`a</td>
<td>b`</td>
<td>Union of sets</td>
<td>O(n+m)</td>
</tr>
<tr>
<td><code>a &amp; b</code></td>
<td>Intersection of sets</td>
<td>O(min(n,m))</td>
<td>&amp; for intersection update</td>
</tr>
<tr>
<td><code>a - b</code></td>
<td>Difference of sets</td>
<td>O(n)</td>
<td>-= for difference update</td>
</tr>
<tr>
<td><code>a ^ b</code></td>
<td>Symmetric difference of sets</td>
<td>O(n)</td>
<td>^= for sym. diff. update</td>
</tr>
<tr>
<td><code>min(a), max(a)</code></td>
<td>Minimum and maximum values</td>
<td>O(n)</td>
<td>Linear search</td>
</tr>
<tr>
<td><code>sorted(a)*</code></td>
<td>Sort set <em>not</em> in place</td>
<td>O(n log n)</td>
<td></td>
</tr>
</tbody>
</table>

*Returns sorted list. Compare to lists’ `a.sort()` in place method*
Dictionaries
Dictionary basics

• Mutable collection of key-value pairs with no repeated elements
• As of Python 3.7, insertion order is preserved
• Implemented as a hash table
• Keys may only be hashable types, values are arbitrary types

Constructors
• `dict(), {}` — empty dictionary
• `dict(iterable)` — sequence of k-v pairs
  • E.g., `dict([(‘a’,1),(‘b’,2)])`
• `{k1: v1, k2: v2, k3: v3}` — key value pairs
• `{k : v for k,v in iterable}` — dictionary comprehension
  • E.g., `{x : x**2 for x in range(5)}`
## Dictionary implementation

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
<th>Runtime</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>len(a)</code></td>
<td>Number of elements</td>
<td>O(1)</td>
<td></td>
</tr>
<tr>
<td><code>key in a</code></td>
<td>True if some element in a has key <code>key</code></td>
<td>O(1)</td>
<td>Hashing</td>
</tr>
<tr>
<td><code>a[key] = val</code></td>
<td>Insert <code>key-val</code> pair</td>
<td>O(1)</td>
<td>Hashing, overwrite if present</td>
</tr>
<tr>
<td><code>del a[key]</code></td>
<td>Remove element, error if not present</td>
<td>O(1)</td>
<td>Hashing</td>
</tr>
<tr>
<td><code>a[key]</code></td>
<td>Return <code>val</code>, error if <code>key</code> not present</td>
<td>O(1)</td>
<td>Hashing</td>
</tr>
<tr>
<td><code>a.update(b)</code></td>
<td>Update with key-val from <code>b</code></td>
<td>O(m)</td>
<td>Overwrite if present</td>
</tr>
<tr>
<td><code>a.keys()</code></td>
<td>Return view of keys</td>
<td>O(1)</td>
<td>Supports set operations</td>
</tr>
<tr>
<td><code>a.values()</code></td>
<td>Return view of values</td>
<td>O(1)</td>
<td></td>
</tr>
<tr>
<td><code>a.items()</code></td>
<td>Return view of <code>(key, val)</code> pairs</td>
<td>O(1)</td>
<td></td>
</tr>
</tbody>
</table>

- `collections.Counter` subclasses dictionaries. Useful to count instances
  - [https://docs.python.org/3/library/collections.html#collections.Counter](https://docs.python.org/3/library/collections.html#collections.Counter)
Looping through dictionaries

• If you only need the keys, use dictionary as iterable
• If you need both keys and values, use `items()`

https://docs.python.org/3/tutorial/datastructures.html#looping-techniques
Live example
Use case: trees

- Dictionaries can be used to create tree data structures
- `create_tree()` — `root = {}`
- `add_child(node, key) — node[key] = {}`
- Example:
  - `{2:
    {7:
      {2: {},
       6:
       {5: {},
        11: {}},
      5:
      {9:
       {4: {}}}}})`
Strings
String basics

• Python’s built-in text sequence type
• Stored as Unicode
• Immutable sequence: supports sequence accessing operations
• Comparisons are lexicographical
String constructors

• ‘We can use “double” quotes’ — single quotations
• “We can use ‘single’ quotes” — double quotations
• ‘‘Triple’’ or “"Triple"” — triple quotes
  • Whitespace (tabs, newlines) are maintained
  • E.g., ‘‘Can look formatted
        like this’’ → “Can look formatted\n\n like this”
• str(obj) — uses obj’s __str__() method
  • Does not parse iterables with characters or strings
  • E.g., str([‘a’, ‘b’, ‘c’]) → “[‘a’, ‘b’, ‘c’]”
  • More on this later in the course
• str() or ‘’ or “” — empty string
String concatenation

- String literals are concatenated with whitespaces
  - ‘Hello’ ‘there’ → "Hello there"
- String variables are concatenated with +
- Repeat string with *
- Concatenate strings from iterable with sep.join(iterable)
- Immutable type! No support for += or *=
Live example
String indexing

• Strings can be indexed like sequences
• Slicing works as with sequences too
• Indexing and slicing return substrings
  • E.g., ‘Python’ [1:3] → ‘yt’
• There is no special character type
  • Characters are just strings of length 1
String matching

- `b in a` — whether `b` is a substring of `a`
- `a.find(b, idx_0, idx_f)` — position of first occurrence of `b` in `a`
  - If `b` is not in `a`, returns `-1`
  - `a.index()` has the same notation but raises error if `b` not in `a`
  - `a.rfind()` or `a.rindex()` return the position of last occurrence
- `a.count(b, idx_0, idx_f)` — occurrences of `b` in `a`
- `a.startswith(prefix, idx_0, idx_f)` — whether `a` begins with `prefix`
  - `prefix` may be a tuple of strings
  - `a.endswith()` for suffixes
- `idx_0` and `idx_f` are optional, and interpreted in slice notation
  - Why is this better than simply slicing?
Live example
String formatting

• `a.format(x, y, name=z, ...)`

• `a` must contain *replacement fields*
  • Expressions surrounded by `{}`
  • To include `{, }` in `a`, escape them as double `{{, }}`

• `{idx!conversion:format_spec}`
  • `idx` — number of the argument (optional, but must be consistent)
  • `name` — may use this for named arguments (possibly mixed with `idx`)
  • If neither is provided, go in order
  • `conversion` — optional, `!s` (str()), `!a` (ascii()) or `!r` (repr())
  • `format_spec` — optional, details to follow
String `format_spec`

- From the docs ([https://docs.python.org/3/library/string.html#format-specification-mini-language](https://docs.python.org/3/library/string.html#format-specification-mini-language))

  `format_spec ::= [[fill][align][sign][#][0][width][grouping_option][.precision][type]`

  - `fill ::= <any character>`
  - `align ::= "<" | ">" | ":" | "^"`
  - `sign ::= "+" | "-" | " "`
  - `width ::= digit+`
  - `grouping_option ::= "_" | ","`
  - `precision ::= digit+`
  - `type ::= "b" | "c" | "d" | "e" | "E" | "f" | "F" | "g" | "G" | "n" | "o" | "s" | "x" | "X" | "%"`
Live example
### Other string operations

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
<th>Note</th>
<th>Related</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>len(a)</code></td>
<td>Number of characters in <code>a</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>a.capitalize()</code></td>
<td>First character capitalized, rest lowercased</td>
<td></td>
<td><code>lower()</code>, <code>swapcase()</code>, <code>title()</code></td>
</tr>
<tr>
<td><code>a.expandtabs(t)</code></td>
<td>Replace tabs with spaces to fill tabs of size <code>t</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>a.isalnum()</code></td>
<td>True if all characters are alphanumeric</td>
<td></td>
<td><code>isalpha()</code>, <code>isdecimal()</code>, <code>isnumeric()</code></td>
</tr>
<tr>
<td><code>a.islower()</code></td>
<td>True if all characters are lowercase</td>
<td></td>
<td><code>istitle()</code>, <code>isupper()</code></td>
</tr>
<tr>
<td><code>a.isspace()</code></td>
<td>True if all characters are whitespace</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>a.partition(sep)</code></td>
<td>Split string into prefix, <code>sep</code>, and suffix</td>
<td>If <code>sep</code> is not found, return <code>a</code> and two empty strings</td>
<td><code>rpartition()</code></td>
</tr>
<tr>
<td><code>a.replace(old, new)</code></td>
<td>Replace all occurrences of <code>old</code> by <code>new</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>a.split(sep, maxsplit)</code></td>
<td>Return list of words with <code>sep</code> as delimiter. Default: whitespace</td>
<td>At most <code>maxsplit</code> splits (optional), leftmost</td>
<td><code>rsplit()</code></td>
</tr>
<tr>
<td><code>a.splitlines()</code></td>
<td>Return list of lines, removing line breaks</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>a.strip()</code></td>
<td>Remove leading and trailing spaces</td>
<td>Optionally: pass string to remove specific characters</td>
<td><code>rstrip()</code>, <code>lstrip()</code></td>
</tr>
</tbody>
</table>

Note: all functions for modifying the string return a modified copy
Functions
Function definitions

• Prototypical example

```python
def function(x, y, z=0):
    '''
    x and y are required
    z is optional and takes value 0 by default
    '''
    print(x, y, z)
```

`function(1, 2)` → 1, 2, 0

• Only optional parameters may come after the first optional parameter

• Default parameters are evaluated at the time of definition
  • This is crucial if you want to use a mutable type!
Live example
Decorators

• Syntactic sugar for function wrappers

• Decorators must be functions that return functions
  • Will be used to wrap around the function itself

• Example

```python
@f1
@f2
def function():
    pass
function = f1(f2(function))
```

• Commonly used for declaring static methods with `@staticmethod` or with user-defined decorators
Annotations

• Parameters may have an annotation following
  • param: annotation

• Functions may also have annotations
  • def function() -> annotation:

• Annotations could be any valid Python expression

• Typical use: type hints

  ```python
  def sum_two_numbers(a: int, b: int) -> int:
      return a + b
  ```

• Does not affect function semantics
  • I.e., type hints are not enforced by Python
  • Useful for type analysis tools

• Access the annotations in function.__annotations__
Takeaways

• We saw a variety of Python objects
  • Sequence objects, with common operators
    • Lists, tuples, range
  • Sets and dictionaries for fast membership testing
  • Strings, which have lots of string methods

• We saw how to define functions
  • Function definitions can handle optional parameters
    • Careful with mutable objects!
  • Decorators give us syntactic sugar for wrappers
  • Annotations are potentially useful for type analysis