Python Data structures

What we'll cover

- 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

Basic list operations

- Running example: `a = [3, 1, 4, 5]`
- `len(a)` — returns the number of elements in `a`: 4
- `a[idx]` — returns the `idx`-th element of `a`, using 0-based indexing
  - idx must be between `0` and `len(a)-1`
  - `a[2]` → `4`
- `a[-idx]` — returns the `idx`-th element of `a` from the right, using 1-based indexing
  - Intuitively means `len(a)-idx`, hence 1-based
  - `idx` must be between `1` and `len(a)`
  - `a[-1]` → `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 = len(a)`
  - `a[1:3]` → `[1, 4]`
  - `a[1:]` → `[1, 4, 5]`
  - `a[:3]` → `[3, 1, 4]` (equivalently `a[:-1]`)
  - `a[:]` → `[3, 1, 4, 5]` (copy)
- `a[idx_0 : idx_f : step]` — as above, but in steps of `step`
  - `a[0:3:2]` → `[3, 4]`
  - `a[-2:-5:-2]` → `[4, 3]`
  - `a[::-1]` → `[5, 4, 1, 3]`
- Support slice assignment: requires assigned array to be the same size as the slice
  - `a[0:3:2] = [0,0]` → `[0, 1, 0, 5]`

List implementation

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

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 for x in range(3))` just creates a *generator*, not a tuple
- Support all list operations for accessing, but not for altering

Tuple implementation

Operation	Description	Runtime	Note
`len(a)`	Number of elements	O(1)	
`a[idx_0:idx_f]`	Slice	O(idx_f - idx_0)	O(1) for single element
`a + b`	Concatenate tuples	O(n+m)	
`a * k`	Repeat tuple `k` times	O(kn)	
`x in a`	True if some element in `a` has value `x`	O(n)	
`min(a), max(a)`	Minimum and maximum values	O(n)	
`a.index(x)`	Position of first element with value `x`	O(n)	
`a.count(x)`	Number of elements with value `x`	O(n)	
`sorted(a)*`	Sort tuple *not* in place	O(n logn)	

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

Range

Range basics

- Built-in collection of `int` objects
- Immutable
- Its size is O(1)
- Constructors
  - `range(stop)` — from 0 to `stop-1`
  - `range(start, stop)` — from `start` to `stop-1`
  - `range(start, stop, step)` — from `start` to `stop-1` on steps of `step`
- Like tuples, support accessing operations
- Does *not* support the repetition (`*`) operator

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`)

Sets

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

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

*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

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

- collections.Counter subclasses dictionaries. Useful to count instances
  - 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

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

- `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'` → `"Hellothere"`
- String variables are concatenated with `+`
- Repeat string with `*`
- Concatenate strings from iterable with `sep.join(iterable)`
- Immutable type! No support for `+=` or `*=`

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?

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}`
  `{name!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

Other string operations

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

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

Functions

Function definitions

- Prototypical example

```
def function(x, y, z=0):
 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!