Python Data Structures

Includes excellent, high-performance data structures as part of language.

Length of Structure

Generic ***len(x)*** returns length of x:

- # chars in string
- # items in list
- # items in dictionary
- # items in a set

Lists

Like JS arrays:

- Mutable, ordered sequence
- ***O(n)*** to search, add, delete
  - Except when at end: ***O(1)***

Making Lists

```
alpha = ['a', 'b', 'c']
```

Can use constructor function, ***list()***

This will make list from iterating over argument:

```
letters = list("apple")   # ['a', 'p', 'p', 'l', 'e']
```

Membership

Can check for membership with ***in***:

```
if "taco" in foods:
    print("Yum!")

if "cheese" not in foods:
    print("Oh no!")
```

Retrieving By Index

Can retrieve/mutate item with `[n]`:

```
print(fav_foods[0])

fav_foods[0] = "taco"
```

```
fav_foods[-1]   # last item

fav_foods[-3]   # third from end
```

Slicing

Can retrieve list from list:

`lst[start:stop:step]`

- ***start***: Index to begin retrieval *(default start)*
- ***stop***: Index to end retrieval before *(default: end)*
- ***step***: Number to step *(default: 1)*

```
alpha = ['a', 'b', 'c', 'd', 'e']

alpha[2:]         # ['c', 'd', 'e']
alpha[2:4]        # ['c', 'd']
alpha[:3]         # ['a', 'b', 'c']
alpha[::2]        # ['a', 'c', 'e']
alpha[3:0:-1]     # ['d', 'c', 'b']
alpha[::-2]       # ['e', 'c', 'a']
```

Splicing

Can assign a list to a splice:

```
alpha = ['a', 'b', 'c', 'd', 'e']

alpha[2:] = ['y', 'z']
print(alpha)            # ['a', 'b', 'y', 'z']

alpha[1:3] = []
print(alpha)            # ['a', 'z']
```

Core API

`l.append(x)`	Add ***x*** to end of of list
`l.copy()`	Return shallow copy of list ***l***
`l.count(x)`	Return # times ***x*** appears in ***l***

`l.extend(l2)`	Add items of ***l2*** to ***l***
`l.index(x)`	Return (0-based) index of ***x*** in ***l***
`l.insert(i, x)`	Insert ***x*** at position ***i***
`l.pop(i)`	Remove & return item at ***i*** (default last)
`l.reverse()`	Reverse list (change in place)
`l.sort()`	Sort list in place

Differences From JS Arrays

Can't add new item with `[]`:

```
alpha = ['a', 'b', 'c']
alpha[3] == 'd'           # error!
```

```
alpha.append('d')         # ok!
```

Functions that mutate list return ***None***, not data:

JavaScript

```
let ltrs = ["c", "a", "b"];
ltrs.sort(); // sorts in-place; returns l
```

Python

```
ltrs = ["c", "a", "b"]
ltrs.sort() # sorts in-place; returns None
```

Strings

Immutable sequence of characters (like JS)

Making Strings

```
msg = "Hello!"
also = 'Oh hi!'

long_msg = """This can continue on for several
lines of text"""

greet = f"Hi, {fname} {lname}"

email = f"""Dear {user},
You owe us ${owed}. Please remit."""
```

```
nums = [1, 2, 3]
```

```
str(nums)      # "[1, 2, 3]"
```

Membership / Substrings

- Can use ***in*** for membership (`"e" in "apple"`)
- Can slice to retrieve substring (`"apple"[1:3] == "pp"`)
  - Cannot splice; strings are immutable!
- Can iterate over, get letter-by-letter:

```
for letter in word:
    print(letter)
```

Core API

`s.count(t)`	Returns # times ***t*** occurs in ***s***
`s.endswith(t)`	Does ***s*** end with string ***t***?
`s.find(t)`	Index of first occurence of ***t*** in ***s*** (-1 for failure)
`s.isdigit()`	Is ***s*** entirely made up of digits?
`s.join(seq)`	Make new string of ***seq*** joined by ***s*** (`"|".join(nums)`)
`s.lower()`	Return lowercased copy of ***s***
`s.replace(old,new,count)`	Replace ***count*** (default: all) occurrences of ***t*** in ***s***
`s.split(sep)`	Return list of items made from splitting ***s*** on ***sep***
`s.splitlines()`	Split ***s*** at newlines
`s.startswith(t)`	Does ***s*** start with ***t***?
`s.strip()`	Remove whitespace at start/end of ***s***

Dictionaries

Mutable, ordered mapping of keys → values

O(1) runtime for adding, retrieving, deleting items

(like JS object or ***Map***)

Making Dictionaries

```
fruit_colors = {
    "apple": "red",
```

```
    "berry": "blue",
    "cherry": "red",
}
```

- Values can be *any type*
- Keys can be any **immutable type**

```
my_dict = {
    "ok": "yes",
    42: "all good",
    [1,2]: 2
} # ERR: not immutable
```

Membership & Retrieval

- ***in*** checks for membership of key (`"apple" in fruit_colors`)
- `[]` retrieves item by key (`fruit_colors['apple']`)
  - Cannot use dot notation, though (no `fruit_colors.apple`)
  - Failure to find is *error* (can say `.get(x, default)`)

Looping over Dictionaries

```
ages = {"Whiskey": 6, "Fluffy": 3, "Ezra": 7}

for name in ages.keys():
    print(name)

for age in ages.values():
    print(age)

for name_and_age in ages.items():
    print(name_and_age)
```

Can unpack ***name_and_age*** while looping:

```
for (name, age) in ages.items():
    print(name, "is", age)
```

JS calls this same idea “destructuring”.

Core API

`d.copy()`	Return new copy of ***d***
`d.get(x, default)`	Retrieve value of ***x*** (return optional ***default*** if missing)

`d.items()`	Return iterable of (key, value) pairs
`d.keys()`	Return iterable of keys
`d.values()`	Return iterable of values

Sets

Unordered, unique collection of items, like JS ***Set***

O(1) runtime for adding, retrieving, deleting

Making Sets

Use `{}`, but with only keys, not `key: value`

```
colors = {"red", "blue", "green"}
```

Can use constructor function to make set from iterable:

```
set(pet_list)    # {"Whiskey", "Fluffy", "Ezra"}

set("apple")     # {"a", "p", "l", "e"}
```

Any immutable thing can be put in a set

Membership

Use ***in*** for membership check:

```
"red" in colors
```

Core API

`s.add(x)`	Add item ***x*** to ***s***
`s.copy()`	Make new copy of ***s***
`s.pop()`	Remove & return arbitrary item from ***s***
`s.remove(x)`	Remove ***x*** from ***s***

Set Operations

```
moods = {"happy", "sad", "grumpy"}

dwarfs = {"happy", "grumpy", "doc"}
```

```
moods | dwarfs    # union: {"happy", "sad", "grumpy", "doc"}

moods & dwarfs    # intersection: {"happy", "grumpy"}

moods - dwarfs    # difference: {"sad"}
dwarfs - moods    # difference: {"doc"}

moods ^ dwarfs    # symmetric difference: {"sad", "doc"}
```

(These are so awesome!)

Tuples

Immutable, ordered sequence

(like a list, but immutable)

Making Tuples

```
t1 = (1, 2, 3)

t2 = ()           # empty tuple

t3 = (1,)         # one-item tuple: note trailing comma
```

Can use constructor function to make tuple from iterable:

```
ids = [1, 12, 44]

t_of_ids = tuple(ids)
```

What Are These Good For?

Slightly smaller, faster than lists

Since they're immutable, they can be used as dict keys or put into sets

Comprehensions

Python has ***filter()*** and ***map()***, like JS

But *comprehensions* are even more flexible

Filtering Into List

Instead of this:

```
evens = []

for num in nums:
    if num % 2 == 0:
        evens.append(num)
```

You can say this:

```
evens = [num for num in nums if num % 2 == 0]
```

Mapping Into List

Instead of this:

```
doubled = []

for num in nums:
    doubled.append(num * 2)
```

You can say this:

```
doubled = [num * 2 for num in nums]
```

Can combine this mapping and filtering:

```
doubled_evens = [n * 2 for n in nums if n % 2 == 0]
```

Super Flexible

Can make lists via comprehensions from *any kind of iterable*:

```
vowels = {"a", "e", "i", "o", "u"}
word = "apple"

vowel_list = [ltr for ltr in word if ltr in vowels]
```

Can make “dictionary comprehensions” and “set comprehensions”:

```
evens_to_doubled = {n: n * 2 for n in nums if n % 2 == 0}

a_words = {w for w in words if w.startswith("a")}
```