Data Structures & Object-Oriented Programming

April 11, 2023

Announcements

- Assignment 0 due Thursday
- Today is the last day to form groups, register for a section in Canvas
- Meet in your ***section room*** this Thursday

  Arpit — Econ 206

  Chase — 380-381T

  Will — Encina West 208
- Assignment 1 — which will be in groups! — goes out Thursday

  preview on the next slide...

Dear Data

Dear Data

Learning Goals

After today, students will be able to:

- Differentiate between the following data structures and describe their properties and methods: lists, tuples, sets, and dictionaries.
- Decide which of the built-in data structures is appropriate for a given task.
- Design and implement custom Python objects (classes) for Python programs to augment Python’s object functionalities.

Agenda

- Built-in data structures
  - Lists, Tuples, Sets, Dictionaries
  - Patterns for working with collections
  - Comprehensions
- Classes
  - High-level overview
  - Magic methods
- Demo: axess.py

Built-in Data Structures

First, a summary

	mutable?	ordered?	iterable?	check inclusion	delimiters
list	✔	✔	over the entries	`O(n)`	`[ ]`
tuple	✘	✔	over the entries	`O(n)`	`( )`
set	✔	✘	over the entries	`O(1)`	`{ }`
dictionary	✔	✘	over the keys	`O(1)` for the keys	`{ }`

Lists

```
to_remember = ['car keys', 'grading', 'the alamo', 42]
```

Lists are...

- **mutable** — they can be changed after they're created

```
to_remember.remove(42)              # O(n)
to_remember.append('september')  # O(1)
```

- **ordered** — there's a 0th element, 1st element, 2nd element, ...

```
to_remember[3] # => 'september'
```

- **heterogeneous** — they can store elements of different types

Lists

`.count(elem)`	**Counts the occurrences of elem in the list.**
`.index(elem)`	Returns the index of the first occurrence of elem in the list.
`.append(elem)`	**Appends the element elem to the end of the list.**
`.extend(iterable)`	Extends the list by appending all elements of iterable to the end.
`.insert(idx, elem)`	Inserts the element elem at the index idx of the list.
`.sort(key=None, reverse=False)`	Sorts the list in-place.
`elem in lst`	**Returns True if elem is in the list and False otherwise**
`del lst[i]`	Removes the ith element from the list
`.pop(i=-1)`	**Returns and removes the ith element from the list.**
`.remove(elem)`	**Removes the first instance of elem from the list, or raises ValueError.**

Tuples

```
pix = (190, 52, 85)
```

How and why Pantone picked 'Viva Magenta' as its 2023 color of the year

December 2, 2022 · 12:50 PM ET

By Rachel Treisman

Tuples are...

- **immutable** — can't be changed after creation (consequently, they're hashable)

```
pix[2] = 210 # TypeError: 'tuple' does not support assignment
hash(pix)    # => 8626792735414146673
```

- **ordered** — there's a 0th element, 1st element, 2nd element, ...

```
pix[0] # => 190
```

- **heterogeneous** — they can store elements of different types

Tuples

Tuples are...

- **immutable** — can't be changed after creation (consequently, they're hashable)

Immutability is powerful!

- When you guarantee that you're not going to change the entries, they can be stored in a slightly more efficient way
- Tuples can be hashed if they contain immutable data structures — remember this for later!
- Tuples contain immutable *references*...

```
tup = (1, 2, [3, 4])
tup[2].append(5)
tup # => (1, 2, [3, 4, 5])
```

This is totally valid, but inadvisable!

Putting it together: `filter_pixels`

```
def is_bright(r, g, b):
    avg_val = (r + g + b) / 3
    return avg_val >= 128

def filter_pixels(pixels):
    # apply is_bright to filter the list
    ...

filter_pixels([
  (11, 231, 128), (224, 178, 46), (226, 226, 133), (225, 83, 205),
  (37, 89, 102), (119, 67, 141), (170, 239, 125), (135, 22, 2),
  (83, 105, 96), (16, 19, 96)
])
```

`filter_pixels.py`

Sets

```python
tas = {'chase', 'arpit', 'will', 'chase', 41}
```

Sets are...

- **mutable** — they can be changed after they're created

```python
tas.add('arpit') # O(1)
tas.remove(41)   # O(1)
```

- **unordered** — there's no guarantee which element you'll pop

```python
tas.pop() # => 'will'
```

- **heterogeneous** — they can store elements of different types
- **unique** — they remove duplicates; every element of a set must be hashable (for now, just think each element must be immutable)

```python
tas # => {'chase', 'arpit'}
```

Sets

Sets are... mathematical objects!

`s & t`	**Set intersection.**
`s | t`	**Set union.**
`s < t`	Check whether s is a proper subset of t.
`s <= t`	**Check whether s is a subset of t.**
`s ^ t`	Symmetric difference.
`s - t`	**Set difference.**

Mathematical sets and efficient phrases

These are efficient phrases	**These aren't efficient phrases**
COLD WINDOWSILL	CHILLY WINDOW LEDGE
COOL MILLION	GOOD THOUSAND THOUSAND
VIVID DISILLUSIONS	GRAPHIC DISAPPOINTMENTS
SUSPICIOUS CONCLUSION	MISTRUSTFUL ENDING

What makes an efficient phrase?

Dictionaries

```
passwords = {
    'tara': 'ilovecs41',
    'arpit': None,
    'chase': 'pyth0nrock$'
}
```

Dictionaries are...

- **mutable** — they can be changed after they're created

```
passwords['arpit'] = 'un1c0rn$4lyfe'
del passwords['chase']
```

- **associative** — access values by keys, not position (no 0th, 1st, 2nd, ... element)
- **heterogeneous** — they can store elements of different types
- **unique keys** — each key can only appear once, keys must be hashable

Dictionaries

`val = d[key]`	Access the value in d corresponding to key; place this value into the val variable.
`d.copy()`	**Makes a shallow copy of d.**
`d.get(key, default)`	**Returns the value associated with key in d. If key does not exist in d, return default.**
`d.keys()`	Returns a collection of the keys in the dictionary.
`d.values()`	Returns a collection of the values in the dictionary.
`d.items()`	Returns a collection of (key, value) tuples in d.
`d.clear()`	**Removes all (key, value) pairs from d.**
`d.pop(key, default)`	Removes key, and its associated value, from d. (Returns the associated value if key is in d, otherwise returns default).

First, a summary

	mutable?	ordered?	iterable?	check inclusion	delimiters
list	✔	✔	over the entries	`O(n)`	`[ ]`
tuple	✘	✔	over the entries	`O(n)`	`( )`
set	✔	✘	over the entries	`O(1)`	`{ }`
dictionary	✔	✘	over the keys	`O(1)` for the keys	`{ }`

Patterns for working with collections

Patterns for working with collections

Patterns for working with collections

```
# number of elements in a collection
```

Patterns for working with collections

```
# number of elements in a collection
len(collection)
```

Patterns for working with collections

```
# number of elements in a collection
len(collection)
```

Patterns for working with collections

```
# number of elements in a collection
len(collection)

# loop over the elements in a collection
```

Patterns for working with collections

```
# number of elements in a collection
len(collection)

# loop over the elements in a collection
for elem in collection:
```

Patterns for working with collections

```
# number of elements in a collection
len(collection)

# loop over the elements in a collection
for elem in collection:
    ...
```

Patterns for working with collections

```
# number of elements in a collection
len(collection)

# loop over the elements in a collection
for elem in collection:
    ...
```

Patterns for working with collections

```
# number of elements in a collection
len(collection)

# loop over the elements in a collection
for elem in collection:
    ...

# create a new data structure from an iterable
```

Patterns for working with collections

```
# number of elements in a collection
len(collection)

# loop over the elements in a collection
for elem in collection:
    ...

# create a new data structure from an iterable
list("abcabc") # => ['a', 'b', 'c', 'a', 'b', 'c']
```

Patterns for working with collections

```
# number of elements in a collection
len(collection)

# loop over the elements in a collection
for elem in collection:
    ...

# create a new data structure from an iterable
list("abcabc") # => ['a', 'b', 'c', 'a', 'b', 'c']
set("abcabc")  # => {'a', 'b', 'c'}
```

Patterns for working with collections

```python
# number of elements in a collection
len(collection)

# loop over the elements in a collection
for elem in collection:
    ...

# create a new data structure from an iterable
list("abcabc") # => ['a', 'b', 'c', 'a', 'b', 'c']
set("abcabc")  # => {'a', 'b', 'c'}
```

Patterns for working with collections

```
# number of elements in a collection
len(collection)

# loop over the elements in a collection
for elem in collection:
    ...

# create a new data structure from an iterable
list("abcabc") # => ['a', 'b', 'c', 'a', 'b', 'c']
set("abcabc")  # => {'a', 'b', 'c'}

# enumerate a collection
```

Patterns for working with collections

```
# number of elements in a collection
len(collection)

# loop over the elements in a collection
for elem in collection:
    ...

# create a new data structure from an iterable
list("abcabc") # => ['a', 'b', 'c', 'a', 'b', 'c']
set("abcabc")  # => {'a', 'b', 'c'}

# enumerate a collection
enumerate(['a', 'b', 41]) # => <(0, 'a'), (1, 'b'), (2, 41)>
```

Patterns for working with collections

```
# number of elements in a collection
len(collection)

# loop over the elements in a collection
for elem in collection:
    ...

# create a new data structure from an iterable
list("abcabc") # => ['a', 'b', 'c', 'a', 'b', 'c']
set("abcabc")  # => {'a', 'b', 'c'}

# enumerate a collection
enumerate(['a', 'b', 41]) # => <(0, 'a'), (1, 'b'), (2, 41)>
for i, elem in enumerate(['a', 'b', 41]):
```

Patterns for working with collections

```
# number of elements in a collection
len(collection)

# loop over the elements in a collection
for elem in collection:
    ...

# create a new data structure from an iterable
list("abcabc") # => ['a', 'b', 'c', 'a', 'b', 'c']
set("abcabc")  # => {'a', 'b', 'c'}

# enumerate a collection
enumerate(['a', 'b', 41]) # => <(0, 'a'), (1, 'b'), (2, 41)>
for i, elem in enumerate(['a', 'b', 41]):
    ...
```

Patterns for working with collections

Patterns for working with collections

```
# sort a collection
```

Patterns for working with collections

```
# sort a collection
sorted("cbda")                  # => ['a', 'b', 'c', 'd']
```

Patterns for working with collections

```
# sort a collection
sorted("cbda")                # => ['a', 'b', 'c', 'd']
sorted("cbda", reverse=True) # => ['d', 'c', 'b', 'a']
```

Patterns for working with collections

```
# sort a collection
sorted("cbda")                      # => ['a', 'b', 'c', 'd']
sorted("cbda", reverse=True) # => ['d', 'c', 'b', 'a']
```

Patterns for working with collections

```
# sort a collection
sorted("cbda")                 # => ['a', 'b', 'c', 'd']
sorted("cbda", reverse=True) # => ['d', 'c', 'b', 'a']

# pairwise combinations
```

Patterns for working with collections

```
# sort a collection
sorted("cbda")                 # => ['a', 'b', 'c', 'd']
sorted("cbda", reverse=True) # => ['d', 'c', 'b', 'a']

# pairwise combinations
zip(
```

Patterns for working with collections

```
# sort a collection
sorted("cbda")                   # => ['a', 'b', 'c', 'd']
sorted("cbda", reverse=True) # => ['d', 'c', 'b', 'a']

# pairwise combinations
zip(
    ['arpit', 'chase', 'will'],
```

Patterns for working with collections

```
# sort a collection
sorted("cbda")                      # => ['a', 'b', 'c', 'd']
sorted("cbda", reverse=True) # => ['d', 'c', 'b', 'a']

# pairwise combinations
zip(
    ['arpit', 'chase', 'will'],
    ['B+', 'A', 'A-']
```

Patterns for working with collections

```
# sort a collection
sorted("cbda")                      # => ['a', 'b', 'c', 'd']
sorted("cbda", reverse=True) # => ['d', 'c', 'b', 'a']

# pairwise combinations
zip(
    ['arpit', 'chase', 'will'],
    ['B+', 'A', 'A-']
) # => <('arpit', 'B+'), ('chase', 'A'), <('will', 'A-')>
```

Patterns for working with collections

```
# sort a collection
sorted("cbda")                  # => ['a', 'b', 'c', 'd']
sorted("cbda", reverse=True) # => ['d', 'c', 'b', 'a']

# pairwise combinations
zip(
    ['arpit', 'chase', 'will'],
    ['B+', 'A', 'A-']
) # => <('arpit', 'B+'), ('chase', 'A'), <('will', 'A-')>
for a, b in zip(collection1, collection2):
```

Patterns for working with collections

```
# sort a collection
sorted("cbda")                  # => ['a', 'b', 'c', 'd']
sorted("cbda", reverse=True) # => ['d', 'c', 'b', 'a']

# pairwise combinations
zip(
    ['arpit', 'chase', 'will'],
    ['B+', 'A', 'A-']
) # => <('arpit', 'B+'), ('chase', 'A'), <('will', 'A-')>
for a, b in zip(collection1, collection2):
    ...
```

Patterns for working with collections

```
# sort a collection
sorted("cbda")                # => ['a', 'b', 'c', 'd']
sorted("cbda", reverse=True) # => ['d', 'c', 'b', 'a']

# pairwise combinations
zip(
    ['arpit', 'chase', 'will'],
    ['B+', 'A', 'A-']
) # => <('arpit', 'B+'), ('chase', 'A'), <('will', 'A-')>
for a, b in zip(collection1, collection2):
    ...
```

Patterns for working with collections

```
# sort a collection
sorted("cbda")                # => ['a', 'b', 'c', 'd']
sorted("cbda", reverse=True)  # => ['d', 'c', 'b', 'a']

# pairwise combinations
zip(
    ['arpit', 'chase', 'will'],
    ['B+', 'A', 'A-']
) # => <('arpit', 'B+'), ('chase', 'A'), <('will', 'A-')>
for a, b in zip(collection1, collection2):
    ...

# range
```

Patterns for working with collections

```
# sort a collection
sorted("cbda")                      # => ['a', 'b', 'c', 'd']
sorted("cbda", reverse=True) # => ['d', 'c', 'b', 'a']

# pairwise combinations
zip(
    ['arpit', 'chase', 'will'],
    ['B+', 'A', 'A-']
) # => <('arpit', 'B+'), ('chase', 'A'), <('will', 'A-')>
for a, b in zip(collection1, collection2):
    ...

# range
range(a, b, c) # => ints from a (inclusive) to b (exclusive)
```

Patterns for working with collections

```
# sort a collection
sorted("cbda")                  # => ['a', 'b', 'c', 'd']
sorted("cbda", reverse=True) # => ['d', 'c', 'b', 'a']

# pairwise combinations
zip(
    ['arpit', 'chase', 'will'],
    ['B+', 'A', 'A-']
) # => <('arpit', 'B+'), ('chase', 'A'), <('will', 'A-')>
for a, b in zip(collection1, collection2):
    ...

# range
range(a, b, c) # => ints from a (inclusive) to b (exclusive)
               #    with a step size of c
```

Patterns for working with collections

```
# sort a collection
sorted("cbda")                # => ['a', 'b', 'c', 'd']
sorted("cbda", reverse=True) # => ['d', 'c', 'b', 'a']

# pairwise combinations
zip(
    ['arpit', 'chase', 'will'],
    ['B+', 'A', 'A-']
) # => <('arpit', 'B+'), ('chase', 'A'), <('will', 'A-')>
for a, b in zip(collection1, collection2):
    ...

# range
range(a, b, c) # => ints from a (inclusive) to b (exclusive)
               #    with a step size of c
range(3, 10, 2) # => <3, 5, 7, 9>
```

Comprehensions

Write a function that returns a list of all odd square numbers below 100

`odd_squares.py`

Comprehensions

Write a function that returns a list of all odd square numbers below 100

`odd_squares.py`

```
for i in range(loop_max):
    if (i ** 2) % 2 != 0:
        output.append(i ** 2)
```

Comprehensions

Write a function that returns a list of all odd square numbers below 100

`odd_squares.py`

```
for i in range(loop_max):
    if (i ** 2) % 2 != 0:
        output.append(i ** 2)
```

Go through a collection...

...check some condition...

...apply some operation to the element.

Comprehensions

Write a function that returns a list of all odd square numbers below 100

odd_squares.py

```
for i in range(loop_max):
    if (i ** 2) % 2 != 0:
        output.append(i ** 2)
```

Go through a collection...

...check some condition...

...apply some operation to the element.

Comprehensions

Write a function that returns a list of all odd square numbers below 100

`odd_squares.py`

```
return [
      i ** 2
      for i in range(int(num ** (1/2)))
      if (i ** 2) % 2 != 0
]
```

...apply some operation to the element.

Go through a collection...

...check some condition...

Comprehensions

```
[fn(x) for x in iterable]
```

Comprehensions

```
[fn(x) for x in iterable if cond(x)]
```

Comprehensions

Square brackets define a list.

```
[fn(x) for x in iterable if cond(x)]
```

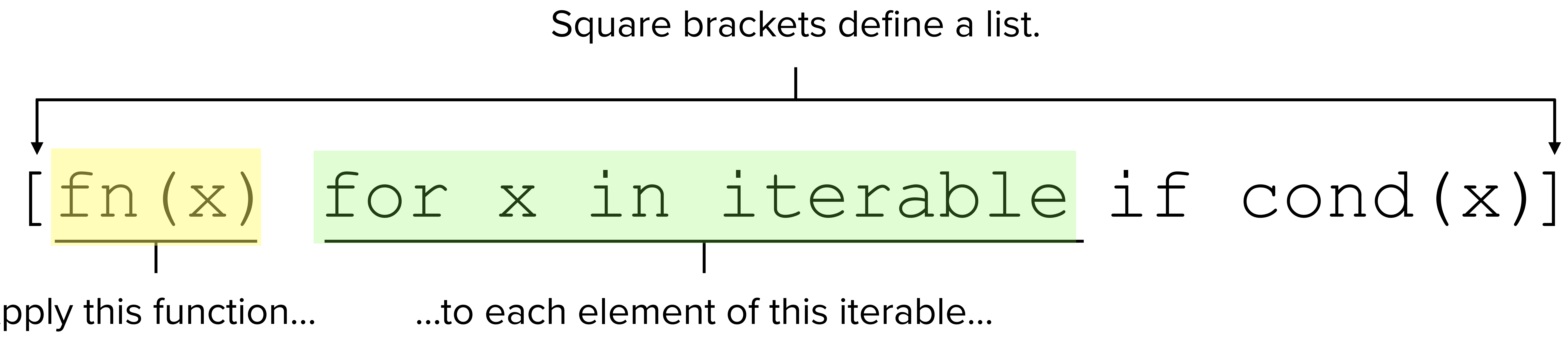
Comprehensions

Square brackets define a list.

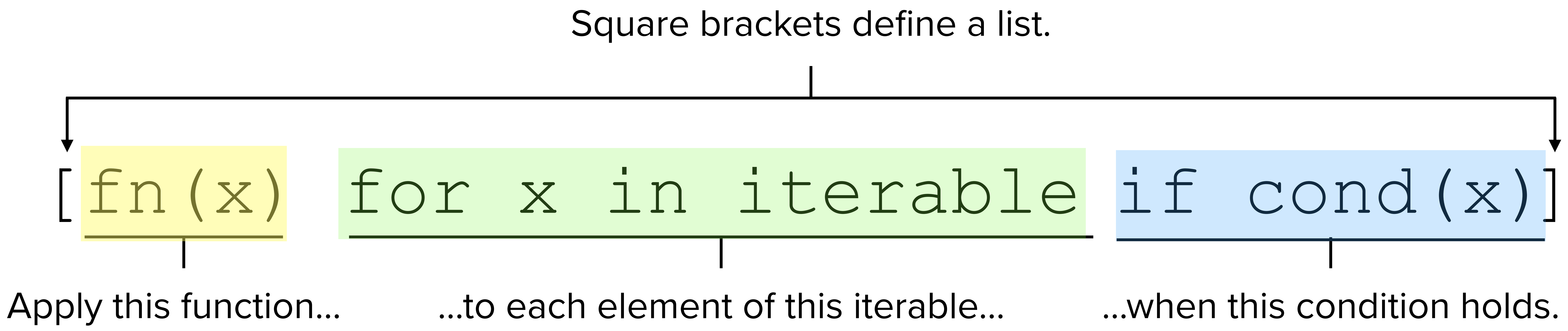
```
[fn(x) for x in iterable if cond(x)]
```

Apply this function...

Comprehensions


Square brackets define a list.
[fn(x) for x in iterable if cond(x)]
Apply this function...
...to each element of this iterable...

Comprehensions



Square brackets define a list.

```
[fn(x) for x in iterable if cond(x)]
```

Apply this function…

…to each element of this iterable…

…when this condition holds.

Comprehensions

```
{f(k):g(v) for k, v in iterable if cond(k, v)}
```

Comprehensions

Curly brackets, colon denote a dictionary!

```
{f(k):g(v) for k, v in iterable if cond(k, v)}
```

Comprehensions

Curly brackets, colon denote a dictionary!

```
{f(k):g(v) for k, v in iterable if cond(k, v)}
```