Data Structures and Algorithms

Sequences & Dictionaries

CS51 – Spring 2026

Strings	'47'
Lists	['cs50', 'cs51']
Tuples	('cs50', 'cs51')
Dictionaries	{'Cecil': 47}

Sequences

- **Sequences** are ordered collections of values.
- There are many types of sequences. Together, we will see:
  - strings
  - lists
  - tuples
  - range objects
- Sequences support the following operations:
  - indexing
  - slicing
  - iteration (looping)
  - checking membership

Strings

Strings are sequences

- A **string** is an ordered sequence of characters.
- A character can be a letter (in almost any alphabet), a digit, a punctuation mark, or white space.
- Strings are enclosed in single or double quotes.
- For example:
  - favorite_class = 'cs51'
  - full_name = 'Cecil Sagehen'
  - empty_string = ''

Length of a string

- To retrieve the length of a string, we use the function `len` and pass the string as an argument. E.g.,
  - `full_name = 'Cecil Sagehen'`
  - `len(full_name)`
    - It will return 13
  - `empty_string = ''`
  - `len(empty_string)`
    - It will return 0

Strings are indexable

- Strings are indexable, with the left-most character being at index 0 and the right-most being at index len-1.
- For example, for the string `'cs51'`:

character	c	s	5	1
index	0	1	2	3

Accessing characters in a string

- You can select a character from a string by passing its index in square brackets.
- For example, for the string favorite_class = 'cs51'
    - second_character = favorite_class[1]
    - Will assign the character 's' to second_character.
- The index in the brackets can be a variable, but must be an integer within [0, len-1]. E.g.,
    - i=1
    - second_character = favorite_class[i]
- But favorite_class[1.5] would result in a TypeError: string indices must be integers
- and favorite_class[4] would result in a IndexError: string index out of range

Accessing characters from right to left

- Python also supports indexing from right to left, where the last character has the index -1.
- last_character = favorite_class[-1]
- Similarly, the second to last character is -2, and so on.

Iterating through a string

- You can use a for statement to loop through the characters of a string. For example:
    - favorite_class = 'cs51'
    - for character in favorite_class :

  print(character)
- Will print:

  c

  s

  5

  1
- character can be substituted with any variable name you think is appropriate.

Looping through a string using range

- You can also use the range function if you need to keep track of the index. E.g.,
  - favorite_class = 'cs51'
  - for i in range(len(favorite_class)):

  print(i, favorite_class[i])
- Will print:

```
0 c
1 s
2 5
3 1
```

Looping through a string using enumerate

- More conveniently, you can use the enumerate function to keep track of the index. E.g.,
  - favorite_class = 'cs51'
  - for i, character in enumerate(favorite_class):

    print(i, character)
- Will print the same:

```
0 c
1 s
2 5
3 1
```

String slices

- A continuous segment of a string is called a **slice**.
- To select a slice, we again use square brackets but indicate the range of indices with a colon (:), i.e. `some_string[start:end]`. For example,
  - `favorite_class = 'cs51'`
  - `favorite_class[1:3]` will return `'s5'`.
- The operator `[star:end]` returns the part of the string from the start-th character to the end-th character, including the first but *excluding* the second.

Slicing and receiving an empty string

- When slicing, if the first index before the colon is greater than or equal to the second index after the colon, the result is the empty string:
- For example,
    - favorite_class[2:2]
    - favorite_class[3:2]
- both return ''

Skipping start index in slices

- When slicing, we can omit the start index before the colon which will return the slice from the first character all the way to the second index. E.g.,
  - favorite_class = 'cs51'
  - favorite_class[:2]
  - Would return 'cs'.
  - This is equivalent to having typed favorite_class[0:2]

Skipping end index in slices

- Similarly, we can omit the end index after the colon which will return the slice from the first index all the way to the end of the string. E.g.,
  - favorite_class = 'cs51'
  - favorite_class[2:]
  - Would return '51'.
  - This is equivalent to having typed favorite_class[2:len(favorite_class)]

Slicing the entire string

- When we don't specify any index left and right of the colon, we receive a copy of the string, e.g.,
- favorite_class = 'cs51'
- favorite_class[:]
- returns 'cs51'

Slicing with steps

- When slicing a string, you can optionally use a step which specifies the interval between characters. That is, you can use the syntax `string[start:end:step]`
  - step defaults to 1, if omitted.
- For example:
- `message = 'I love cs51!'`
- `message[2:11:2]` returns a string with every second character between indices 2 (inclusive) and 11 (exclusive), that is it returns the string `'lv s1'`

Slicing with steps – skipping start

- If we don't define `start`, that is `string[:end:step]`, we get all the characters from index 0 to end (excluded) at an interval of `step`.
- For example:
- `message = 'I love cs51!'`
- `message[:11:2]` returns a string with every second character between indices 0 (inclusive) and 11 (exclusive), that is it returns the string `'Ilv s1'`

Slicing with steps – skipping end

- If we don't define `end`, that is `string[start::step]`, we get all the characters from index start (inclusive) to the last character at an interval of `step`.
- Note the use of two colons, `string[start::step]`, to distinguish it from `string[start:end]`,
- For example:
- `message = 'I love cs51!'`
- `message[3::2]` returns a string with every second character between indices 3 (inclusive) and the last character ('!'), that is it returns the string `'oec5!'`

Slicing with steps – skipping both start and end

- If we don't define start and `end`, that is `string[::step]`, we get all the characters at an interval of `step`.
- Note the use of two colons, `string[::step]`, to distinguish it from `string[:end]`,
- For example:
- `message = 'I love cs51!'`
- `message[::3]` returns a string with every third character, that is it returns the string `'lo 5'`

Strings are immutable

- Strings are **immutable** which means that we cannot change an existing string. E.g.,
  - favorite_class = 'cs51'
  - favorite_class[-1] = '0'
- We would receive TypeError: 'str' object does not support item assignment
- How we could get around this is by typing:
  - new_favorite_class = favorite_class[:-1] + '0'
  - which would result in new_favorite_class being equal to 'cs50'.
  - This example **concatenates** a slice of favorite_class from the first character to second to last with the character '0'. It has no effect on the original string favorite_class.

An aside: objects

- So far, we have seen types like int, float, strings.
- But all these types, really almost everything in Python is an object, that is an instance of a specific class.
- For example, the string `'cs51'` is an object. `type('cs51 ')` returns `<class 'str'>`. Similarly, the object 47 is an instance of the class `int`.
- This idea comes from the object-oriented programming (OOP) paradigm.
- Python often hides this fact. We have not realized all this time that there are objects and have been working simply with functions.
- If you continue with CS62, you will see Java, an OOP language that you will have to explicitly work with objects.

Methods

- A key characteristic of objects is that they have **methods**, that is special functions that can manipulate the state of an object.
- A method can only be called through an object using the **dot operator**.
  - obj.method(arguments)

Stripping strings

- Stripping a string using the strip method removes any **leading** and **trailing** whitespaces and returns a new string with these characters removed. E.g.,
  - test = '   cs51   '
  - new_test = test.strip()
  - new_test is going to be 'cs51'. test remains unchanged.
- You can also pass specific leading and trailing characters that you want to be stripped.
  - favorite_class = 'so, cs51 is my favorite class'
  - print(favorite_class.strip('s')) would print 'o, cs51 is my favorite cla'

Capitalization in strings

- Python has bult-in string methods to test or change capitalization.
- favorite_class = 'CS51 ROCKS'
- favorite_class.isupper() returns True
- favorite_class.islower() returns False
- lower_favorite_class = favorite_class.lower() returns a new string that is all lowercase, i.e. 'cs51 rocks'
- Similarly, lower_favorite_class.upper() would return a new string that is all uppercase.

Comparing strings

- To see if two strings are equal, we can use the == operator.
  - Not the =, because this is the assignment operator!
- Similarly, < and > compare two strings in lexicographic order (this includes the comparison of letters, symbols, and numbers).
- Symbols come before numbers.
- String representation of numbers comes before letters.
- All the uppercase letters come before lowercase letters.

Comparing strings example

```
def compare_class(class_number):
    if class_number < 'cs51':
        print(class_number, 'comes before cs51.')
    elif number > 'cs51':
        print(class_number, 'comes after s51.')
    else:
        print('Hello, cs51')
```

compare_class('cs50') prints cs50 comes before cs51.

compare_class('CS62') prints CS62 comes before cs51.

Practice time

```
def compare_class(class_number):
    if class_number < 'cs51':
        print(class_number, 'comes before cs51.')
    elif number > 'cs51':
        print(class_number, 'comes after s51.')
    else:
        print('Hello, cs51')
```

What will happen if I call `compare_class('cs50')` and next `compare_class('CS62')`?

Answer

```
def compare_class(class_number):
    if class_number < 'cs51':
        print(class_number, 'comes before cs51.')
    elif class_number > 'cs51':
        print(class_number, 'comes after cs51.')
    else:
        print('Hello, cs51')
```

What will happen if I call `compare_class('cs50')` and next `compare_class('CS62')`?

cs50 comes before cs51.

CS62 comes before cs51.

Practice time

- Define a function `str_odd_indices` that takes one parameter `s` (a string) and returns a string comprised of only the odd indexed characters of `s`.
- For example, `str_odd_indices('hello!')` would return `'el!'`.

Answer – Option 1

```
def str_odd_indices(s):
    result = ''
    for i in range(len(s)):
        if i % 2 == 1:
            result += s[i]
    return result
```

Answer – Option 2

```
def str_odd_indices(s):
    result = ''
    i = 0
    for ch in s:
        if i % 2 == 1:
            result += ch
        i += 1
    return result
```

Answer – Option 3

```
def str_odd_indices(s):
    result = ''
    for i, char in enumerate(s):
        if i % 2 == 1:
            result += char
    return result
```

Answer – Option 4

```
def str_odd_indices(s):
    return s[1::2]
```

Practice time

- Define a function `find_char` that takes two parameters, a string `s` and a character `c` and returns the index of the first instance of that character. If that character does not appear in the string, it returns -1.
- For example:
- find_char('hello', 'h') == 0
- find_char('hello', 'l') == 2
- find_char('hello', 'a') == -1

Answer – Option 1

```
def find_char(s, c):
    for i in range(len(s)):
        if s[i] == c:
            return i
    return -1
```

Answer – Option 2

```
def find_char(s, c):
    i = 0
    for ch in s:
        if ch == c:
            return i
        i += 1
    return -1
```

Answer – Option 3

```
def find_char(s, c):
    for i, ch in enumerate(s):
        if ch == c:
            return i
    return -1
```

Answer – Secret Option 4

```
def find_char(s, c):
    return s.find(c)
```

Practice time

- test = 'CS51 is my favorite class'
- Evaluate the following expressions:
  - test[10]
  - test[0:2]
  - test[:5]
  - test[::2]

Answer

- test = 'CS51 is my favorite class'
- Evaluate the following expressions:
  - test[10] -> ' '
  - test[0:2] -> 'CS'
  - test[:5] -> 'CS51 '
  - test[::2] -> 'C5 sm aoiecas'

Practice time

- Define a function `second_half` that takes one parameter `s` (a string) and returns the second half of `s`. Assume that if it's an odd number of characters, you round up.

Answer

```
def second_half(s):
    mid = len(s) // 2
    return s[mid:]
```

Lists

Lists are sequences

- Like a string, a **list** is a sequence of values.
- In a string, the values are characters; in a list, they can be any type.
- The values in a list are called **list elements**.

Creating lists

- The simplest way to create a list is to enclose its elements in square brackets and separate them by comma, e.g.,
  - favorite_numbers = [24, 47]
    - This is a list of two integers
  - cheeses = ['Feta','Cheddar', 'Edam', 'Gouda']
    - This is a list of three strings
- You can also create a list using the list function.
- E.g., list('cs51') will create the list ['c','s','5','1'].

Types of elements in a list

- Python allows elements of different types in the same list (not all languages allow this!)
  - bizarre_list = [47, 'cs51']

Empty lists

- A list that contains no elements is called an empty list; you can create one with empty brackets, []. E.g.,
  - empty_list = []
- Can also achieve this using the list function:
  - empty_list = list()

Length of a list

- To retrieve the length of a list, we use the function len. E.g.,
    - cheeses = ['Feta','Cheddar', 'Edam', 'Gouda']
    - len(cheeses)
        - It will return 4
    - empty_list = []
    - len(empty_list)
        - It will return 0

Lists are indexable

- Lists are indexable, with the left-most element being at index 0 and the right-most being at index len-1.
- For example, for the list `['Feta','Cheddar', 'Edam', 'Gouda']` :

element	'Feta'	'Cheddar'	'Edam'	'Gouda'
index	0	1	2	3

Accessing elements in a list

- You can select an element in a list by passing its index in the square bracket operators.
- For example, for the list `cheeses = ['Feta','Cheddar', 'Edam', 'Gouda']`:
  - `best_cheese = cheeses[0]`
  - Will assign the element `'Feta'` to `best_cheese`.

Lists are mutable

- Lists are **mutable** which means that we can change the elements in an existing list. E.g.,
  - cheeses = ['Feta','Cheddar', 'Edam', 'Gouda']
  - cheeses[-1] = 'Haloomi'
- We would replace the last element, 'Gouda', with 'Haloomi'.
- If you try to read or write an element that does not exist, you get an IndexError.

in operator

- The in operator checks whether a given element appears anywhere in the list. E.g.,
- cheeses = ['Feta','Cheddar', 'Edam', 'Gouda']
- 'Haloomi' in cheeses would return False, but 'Feta' in cheeses would return True.

List slices

- Similarly to strings, we can slice a list by specifying a range in the square brackets, [], using the colon (:)
  - `my_list[start:end]` will return a new list with the elements from `start` index through `end-1`.
  - `my_list[start:]` will return a new list with the elements from `start` to the end of the list.
  - `my_list[:end]` will return a new list with the elements from 0 through `end-1`.
  - `my_list[:]` will return a copy of the entire `my_list`.
- Note, because `list` is the name of a built-in function, you should avoid using it as a variable name.

+ operator

- The + operator concatenates two lists into a new list. E.g.,
- list_1 = [1, 2]
- list_2 = [3, 4]
- list_3 = list_1 + list_2
- list_3 is [1, 2, 3, 4] and list_1 and list_2 remain unchanged.

* operator

- The * operator repeats a list a given number of times. E.g.,
- list_1 = [1, 2]
- list_2 = list_1 * 3
- list_2 is [1, 2, 1, 2, 1, 2] and list_1 remains unchanged.

Other operations

- No other mathematical operators work with lists, but the built-in function `sum` adds up the elements in lists of numbers. E.g.,
- `sum([1.5, 2])` returns 3.5
- `min` and `max` return the minimum and maximum element in a list. E.g.,
- `min([2, 1])` returns `1`
- `max('a', 'b')` returns `'b'`

Methods to add elements in a list

- append adds a new element to the end of a list:
  - cheeses = ['Feta','Cheddar', 'Edam', 'Gouda']
  - cheeses.append('Haloumi')
  - cheeses is now ['Feta', 'Cheddar', 'Edam', 'Gouda', 'Haloumi']
- extend takes a list as an argument and appends all of the elements:
  - cheeses.extend(['Emmental', 'Gruyere'])
  - cheeses is now ['Feta', 'Cheddar', 'Edam', 'Gouda', 'Haloumi', 'Emmental', 'Gruyere']
- insert adds a new element to a specified index of a list:
  - cheeses.insert(1, 'Brie')
  - cheeses is now ['Feta', 'Brie', 'Cheddar', 'Edam', 'Gouda', 'Haloumi', 'Emmental', 'Gruyere']

Methods to remove elements from a list

- Python also provides methods that operate on lists to remove elements.
- For example, pop removes an element from the specified index (or last position, if not specified). For example:
  - cheeses = ['Feta','Cheddar', 'Edam', 'Gouda']
  - removed_cheese = cheeses.pop(1)
  - cheeses is now ['Feta', 'Edam', 'Gouda'] and removed_cheese has been assigned the popped element 'Cheddar'
- If you know the element you want to remove but not the index, you can use the remove method, as long as the element indeed exists in the list. For example:

  cheeses = ['Feta','Cheddar', 'Edam', 'Gouda']

  cheeses.remove('Cheddar')

  cheeses is ['Feta', 'Edam', 'Gouda'] and in contrast to pop nothing is returned.

Splitting strings to list of words

- If you want to break a string into words separated by whitespace, you can use the split method.
  - motto = 'cs51 is my favorite class'
  - broken_motto = motto.split()
  - broken_motto is ['cs51', 'is', 'my', 'favorite', 'class']

Converting strings to lists

- You can also pass a **delimeter** to split specify which characters you want to use as word boundaries. For example:
  - motto = 'cs51_is my_favorite class'
  - broken_motto = motto.split('_')
  - broken_motto is ['cs51', 'is my', 'favorite class']

From list of strings to a single string

- If you have a list of strings, you can concatenate them into a single string using join.
- join is a string method, so you have to invoke it on the delimiter and pass the list as an argument. For example:
- delimiter = ' '
- favorite_list = ['cs51', 'is', 'my', 'favorite', 'class']
- favorite_string = delimiter.join(favorite_list)
- favorite_string will be 'cs51 is my favorite class', joined by whitespaces.

Looping through a list

- You can use a for statement to loop through the elements of a list. For example:
- cheeses = ['Feta','Cheddar', 'Edam', 'Gouda']
- for cheese in cheeses:

  print(cheese)
- Will print:

```
Feta
Cheddar
Edam
Gouda
```

Looping through a list using range

- You can also use the range function if you need to keep track of the index. E.g.,
- cheeses = ['Feta','Cheddar', 'Edam', 'Gouda']
- for i in range(len(cheeses)):

  print(i, cheeses[i])
- Will print:

0 Feta

1 Cheddar

2 Edam

3 Gouda

Looping through a list using enumerate

- You can also use the enumerate function if you need to keep track of the index. E.g.,
- cheeses = ['Feta','Cheddar', 'Edam', 'Gouda']
- for i, cheese in enumerate(cheeses):

  print(i, cheese)
- Will print the same:

```
0 Feta

1 Cheddar

2 Edam

3 Gouda
```

References

- If we run this assignment statement:
- x = 47
- We associate the name on the left hand with the value on the right side. We say that x is a **reference** to 47.

$$\begin{tikzcd} x \arrow[r] & 47 \end{tikzcd}$$

- More practice with references: https://pythontutor.com/
- Read more about references: https://nedbatchelder.com/text/names.html

References

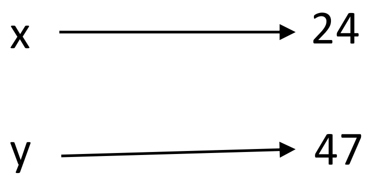
- Let's say we have the following two statements:
- x = 47
- y = x
- Both x and y refer to the same value:

$$\begin{tikzcd}
x \arrow[dr] & \\
& 47 \\
y \arrow[ur] &
\end{tikzcd}$$

- Assigning a value to a name never copies the data, it never makes a new value. Assignment just makes the name on the left refer to the value on the right.

References

- What will happen with the following code?
- x = 47
  y = x
  x = 24
- When we said y = x doesn't mean that they will always be the same forever. Reassigning x leaves y alone.


$$\begin{tikzcd} x \arrow[r] & 24 \\ y \arrow[r] & 47 \end{tikzcd}$$

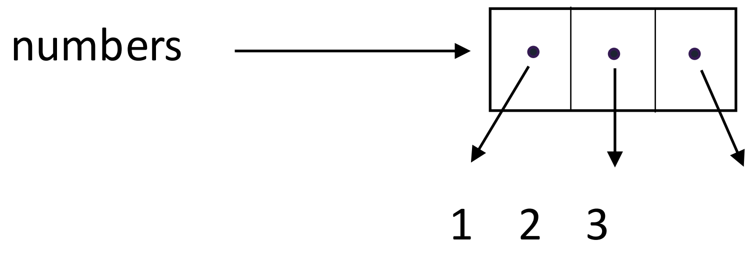
References

- What will happen with the following code?
- x = 47
  x = x + 1
- Because `int`s are immutable, you can’t change one in-place, you can only make a new value and assign it to the same name:

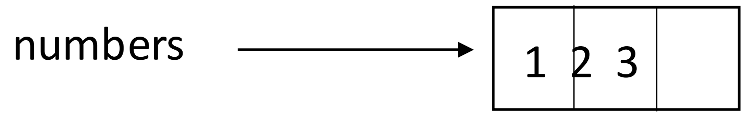
$$\begin{tikzcd} x \arrow[r] & 48 \end{tikzcd}$$

References

- What will happen with the following code?
- numbers = [1, 2, 3]

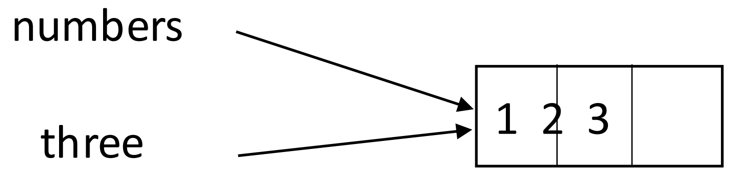


- For simplicity



References

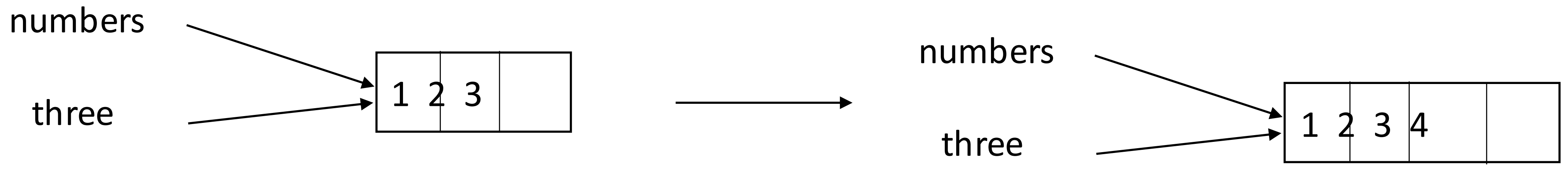
- What will happen with the following code?
- numbers = [1, 2, 3]
  three = numbers
- Remember, assignment never makes new values, and it never copies data. We have one list, referred to by two names.



References

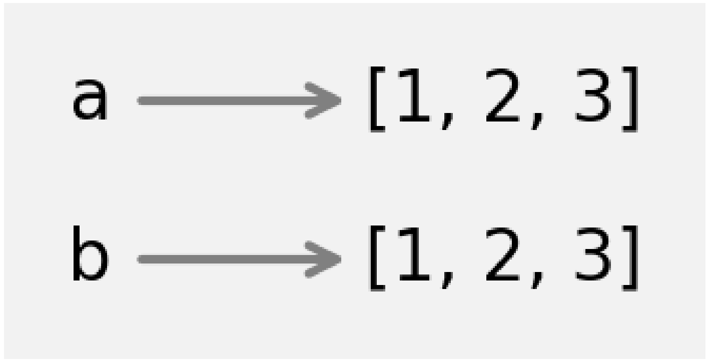
- What will happen with the following code?

-
  ```
  numbers = [1, 2, 3]
  three = numbers
  numbers.append(4)
  ```
- We mutated numbers through calling the append method. Since three refers to that list, when we look at three we see the same list as numbers , which has been changed, so tri now shows four numbers also:

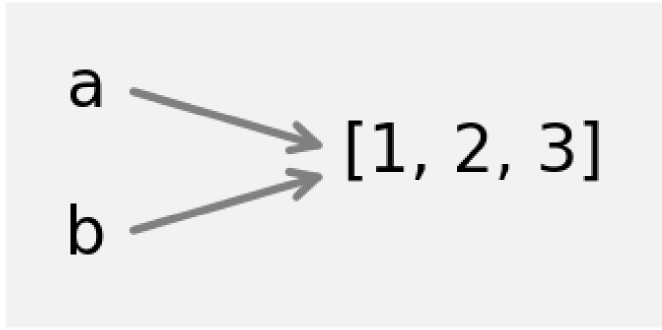


Equivalent but not identical lists

- If we run these assignment statements:
- a = [1, 2, 3]
- b = [1, 2, 3]
- a is b
- Will return False!
- In this case we would say that the two lists are **equivalent**, because they have the same elements, but not **identical**, because they are not the same object.
- If two objects are identical, they are also equivalent, but if they are equivalent, they are not necessarily identical.



Aliasing

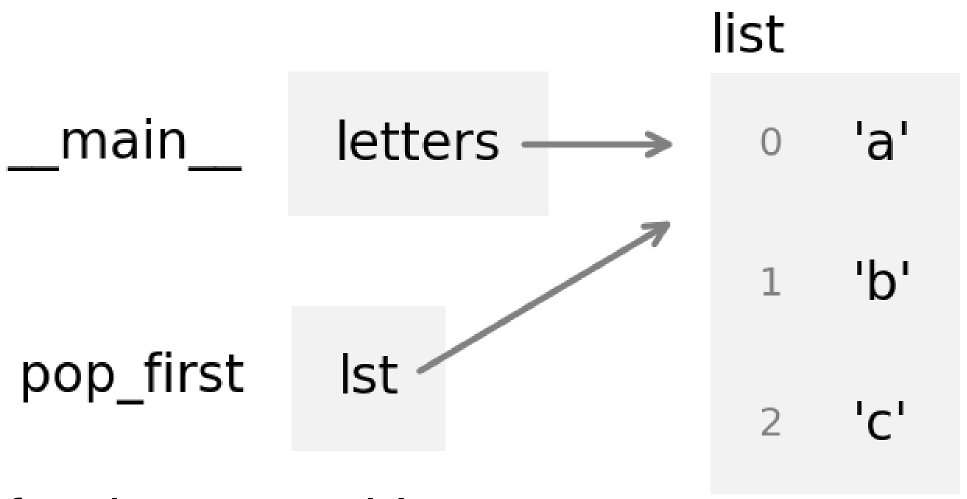


- An object with more than one reference has more than one name, so we say the object is **aliased**. If the aliased object is *mutable*, changes made with one name affect the other.
- In this example, if we change the object `b` refers to, we are also changing the object `a` refers to.
- So we would say that `a` “sees” this change. Although this behavior can be useful, it is error-prone. In general, it is safer to avoid aliasing when you are working with mutable objects.
- For immutable objects like strings and numbers, aliasing is not a problem because we cannot change their contents.

Passing a list to a function

- When you pass a list to a function, the function gets a reference to the list. If the function modifies the list, the caller sees the change.
- For example, pop_first uses the list method pop to remove the first element from a list.
- def pop_first(lst):

  return lst.pop(0)
- letters = ['a', 'b', 'c']
- pop_first(letters)
- The parameter lst and the variable letters are aliases for the same object.
- If the function modifies the object, those changes persist after the function is done. Thus, this call will return 'a'. letters is now ['b', 'c'].



Passing a list to a function

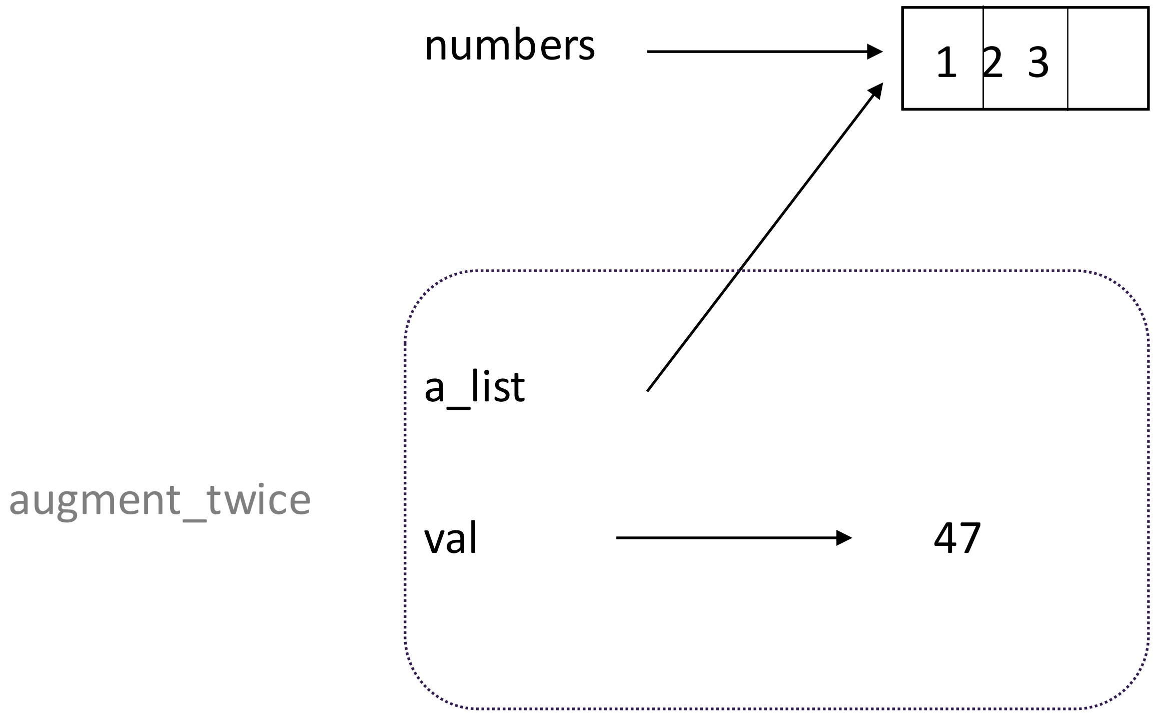
- What do you think would happen here?

-
```
def augment_twice(a_list, val):
    a_list.append(val)
    a_list.append(val)

numbers = [1, 2, 3]
augment_twice(numbers, 47)
```

Passing a list to a function

- What do you think would happen here?

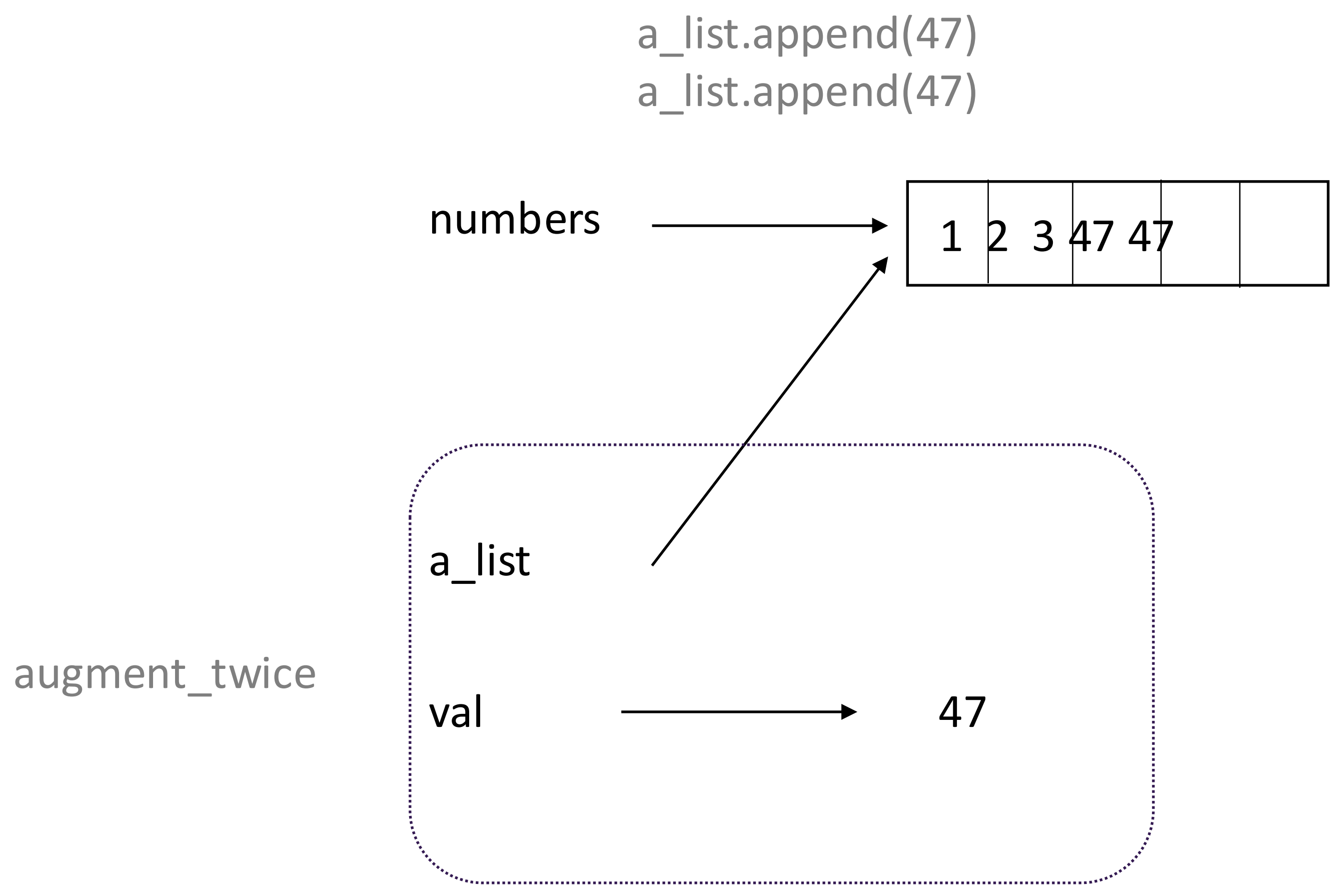
-
  ```
  def augment_twice(a_list, val):
      a_list.append(val)
      a_list.append(val)

  numbers = [1, 2, 3]
  augment_twice(numbers, 47)
  ```



Passing a list to a function

- What do you think would happen here?

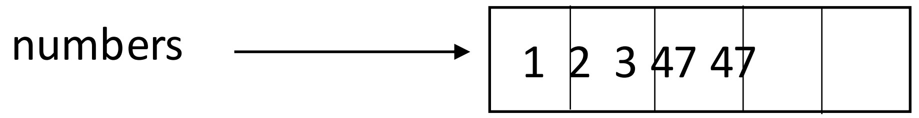
-
  ```
  def augment_twice(a_list, val):
      a_list.append(val)
      a_list.append(val)

  numbers = [1, 2, 3]
  augment_twice(numbers, 47)
  ```



Passing a list to a function

- What do you think would happen here?

-
  ```
  def augment_twice(a_list, val):
      a_list.append(val)
      a_list.append(val)

  numbers = [1, 2, 3]
  augment_twice(numbers, 47)
  ```



Passing a list to a function

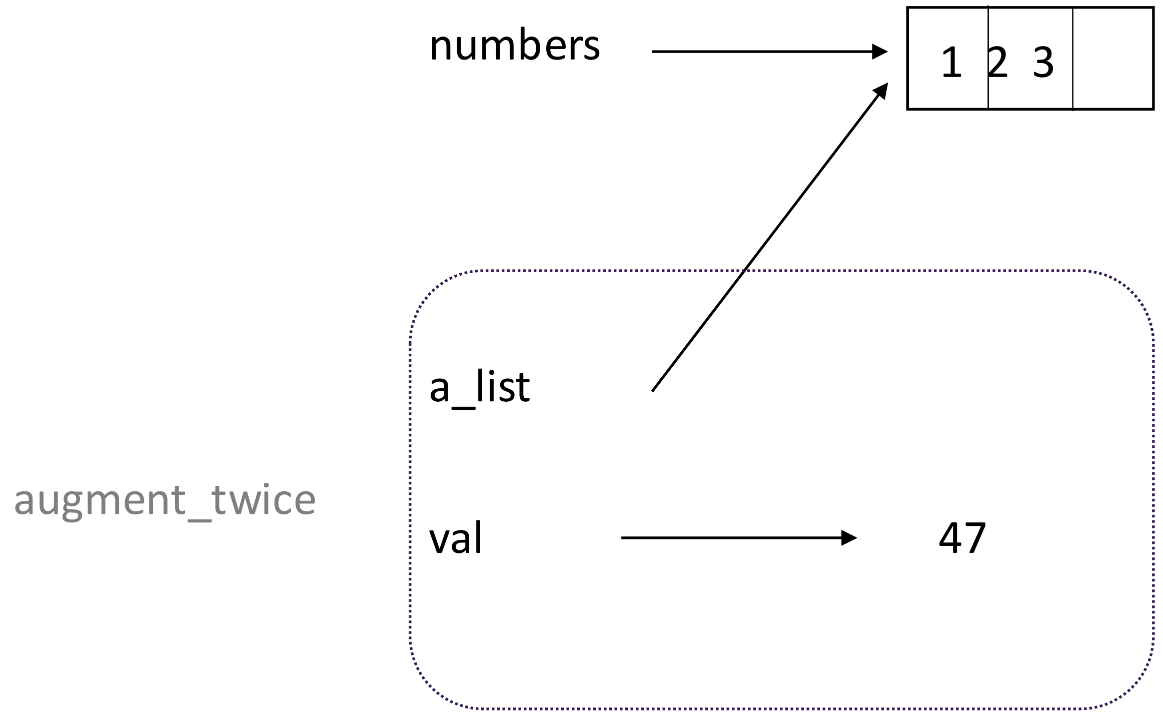
- What do you think would happen here?

-
```
def augment_twice(a_list, val):
    a_list = a_list + [val, val]

numbers = [1, 2, 3]
augment_twice(numbers, 47)
```

Passing a list to a function

- What do you think would happen here?

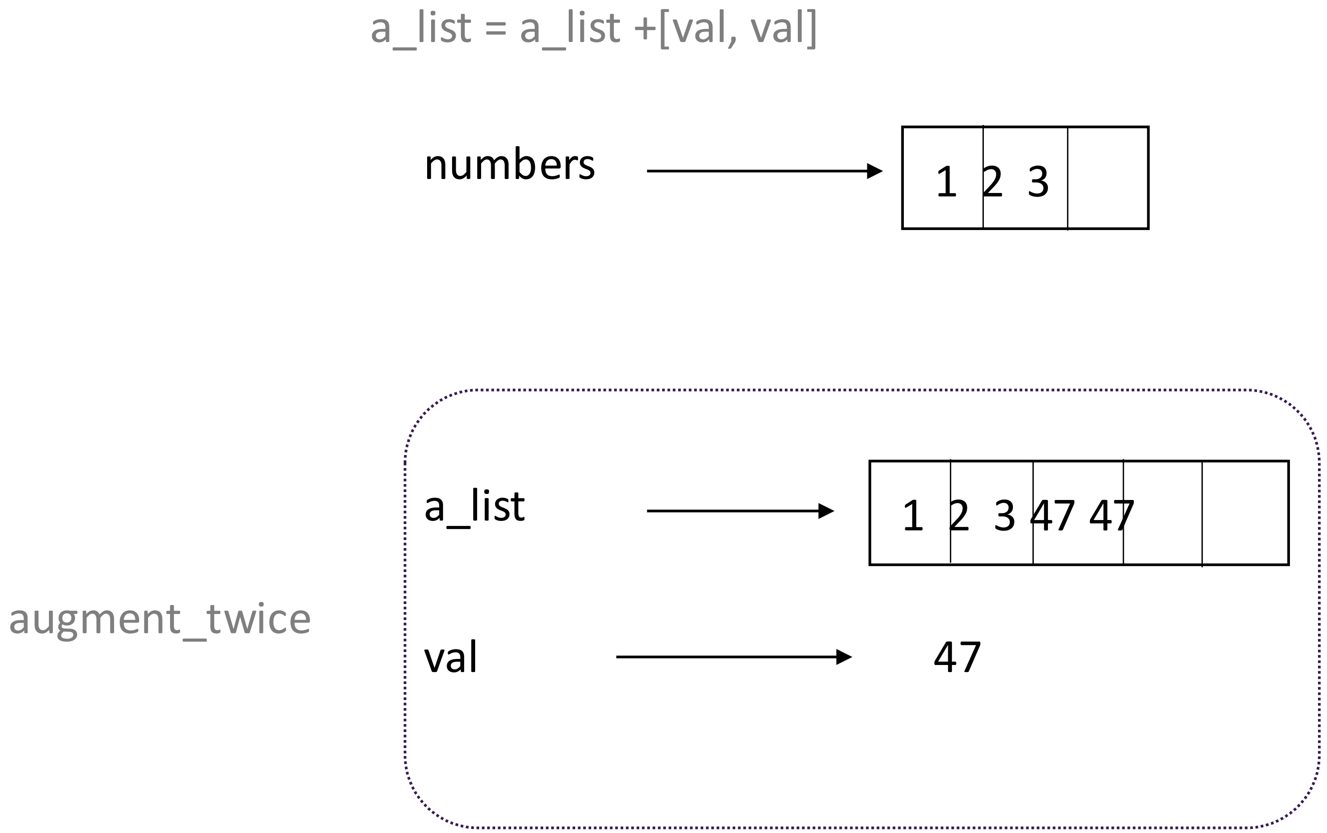
-
  ```
  def augment_twice(a_list, val):
      a_list = a_list + [val, val]

  numbers = [1, 2, 3]
  augment_twice(numbers, 47)
  ```



Passing a list to a function

- What do you think would happen here?

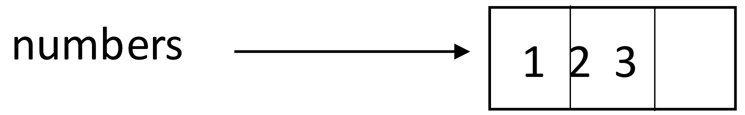
-
  ```
  def augment_twice(a_list, val):
      a_list = a_list + [val, val]

  numbers = [1, 2, 3]
  augment_twice(numbers, 47)
  ```



Passing a list to a function

- What do you think would happen here?

-
  ```
  def augment_twice(a_list, val):
      a_list = a_list + [val, val]

  numbers = [1, 2, 3]
  augment_twice(numbers, 47)
  ```



Typical mistakes

- Note that most *list* methods modify the argument and return None.
- This is the opposite of the *string* methods, which return a new string and leave the original alone.
- For example:
- word = 'plumage!'
- word = word.strip('!')
- word is now 'plumage'
- But you don't want to write
- t = [1, 2, 3]
- t = t.remove(3)   # WRONG! t is now NoneType!

Practice time

- Write a recursive Python function called `rec_linear_search` that takes a list `lst`,an `element`, and an `index` that has a default value of 0, and returns the index of the first encounter of the `element` in the list, if it exists, or -1 if it does not.

Answer

```
def rec_linear_search(lst, element, index=0):
    if index == len(lst):
        return -1
    elif lst[index] == element:
        return index
    else:
        return rec_linear_search(lst, element, index+1)
```

Practice time

- Given a list *my_list* (of size `n`) of numbers, write an iterative function `sum_of_list_numbers` that calculates the sum of the numbers in *my_list*.
- What are the pre-and post-conditions?
- What is a good loop invariant?
- Use loop invariants to prove that your function works correctly.

Practice time

- Pre: a list of n numbers
- Post: a number that is equal to the sum of the n numbers in *my_list*

```
def sum_of_list_numbers(my_list):
    answer = 0
    n=len(my_list)
    for i in range(n):
        answer += my_list[i]
    return answer
```

- Loop invariant: At the start of iteration i of the loop, the variable answer should contain the sum of the numbers from the sublist my_list[0:i].

Practice time

- *Initialization*: At the start of the first loop, the loop invariant states: 'At the start of the first iteration of the loop, the variable answer should contain the sum of the numbers from the sublist my_list[0:0], which is an empty list. The sum of the numbers in an empty list is 0, and this is what answer has been set to.
- *Maintenance*: Assume that the loop invariant holds at the start of iteration i. Then it must be that answer contains the sum of numbers in slice my_list[0:i]. In the body of the loop, we add my_list[i] to answer. Therefore, at the end of iteration and i and before i+1 iteration begins, answer will contain the sum of numbers in my_list[0:i+1], which is what we needed to prove.
- *Post-condition*: When the loop terminates, i should be equal to n and the loop invariant gives that answer contains the sum of all numbers in slice my_list[0:n] which is equal to list. Thus, we will indeed get the sum of all numbers in my_list.
- *Termination*: i increases by 1 in every iteration and ranges from 0 to maximum n==len(my_list). The for loop will terminate in a finite number of steps.

Tuples

Tuples are sequences

- Like a list, a **tuple** is a sequence of values.
- The values can be any type, and they are indexed by integers, so tuples are a lot like lists.
- The important difference is that tuples are **immutable**. Once we create a tuple, we cannot change its contents.
- Tuples are usually used when we can safely assume that the collection of tuple values will not change.

Creating a tuple

- To create a tuple, you can write a *comma-separated* sequence of values. Optionally, you can enclose the values in parentheses, that is:
- `t = 'l', 'u', 'p', 'i', 'n'` and `t = ('l', 'u', 'p', 'i', 'n')` are both valid options.
- Note that the important syntax here is the commas, not the parentheses.
  - To create a tuple with one element we would write `t1 = 'p',`
  - `t2 = ('p')` would actually create a string!
- Another way to create a tuple is the built-in function tuple. With no argument, it creates an empty tuple.
  - `t = tuple()` results in the empty tuple `()`. Equivalent to `t=()`
- If the argument is a sequence (string, list or tuple), the result is a tuple with the elements of the sequence.
- `t = tuple('lupin')` results in the tuple `('l', 'u', 'p', 'i', 'n')`

Operating on tuples

- Most list operators also work with tuples.
  - For example, the bracket operator indexes an element.
  - `t = tuple('lupin')`
  - `t[0]` returns `'l'`
- And the slice operator selects a range of elements.
  - `t[1:3]` returns `('u', 'p')`
- The `+` operator concatenates tuples.
  - `tuple('lup') + ('i', 'n')` results in `('l', 'u', 'p', 'i', 'n')`
- And the `*` operator duplicates a tuple a given number of times.
  - `tuple('spam') * 2` results in `('s', 'p', 'a', 'm', 's', 'p', 'a', 'm')`

Tuples are immutable

- If you try to modify a tuple with the bracket operator, you get a TypeError.
- t = tuple('lupin')
- t[0] = 'L' will result in TypeError: 'tuple' object does not support item assignment
- And tuples do *not* have any of the methods that modify lists, like append and remove.

Tuples assignment

- You can put a tuple of variables on the left side of an assignment, and a tuple of values on the right. E.g.,
- `a, b = 1, 2`
- The values are assigned to the variables from left to right – in this example, `a` gets the value 1 and `b` gets the value 2.
- We can display the results like this:
  - `a, b` which results in `(1, 2)`

Tuples assignment

- More generally, if the left side of an assignment is a tuple, the right side can be any kind of sequence – string, list or tuple.
- For example, to split an email address into a username and a domain, you could write:
- email = 'cecil47@pomona.edu'
- username, domain = email.split('@')
- The return value from split is a list with two elements – the first element is assigned to username, the second to domain.
- username, domain results in ('cecil47', 'pomona.edu')
- The number of variables on the left and the number of values on the right have to be the same – otherwise you get a ValueError.

Range objects

Range objects

- A range is a sequence that represents an arithmetic progression of integers. E.g.,
- `my_range = range(1, 10)` creates a sequence of integers between 1 (inclusive) and 10 (exclusive), i.e. the numbers 1, 2, 3, 4, 5, 6, 7, 8, 9.
- In general, to create a range, we use the syntax `range(start, end)`.
- We can skip the start, that is only write `range(end)`. That would be equivalent to `range(0, end)`.
- Optionally, we can define a step, that is an interval in the numbers we generate. E.g.,
- `my_range = range(1, 10, 2)` would represent the numbers 1, 3, 5, 7, and 9 (we skip every other number).
- If we don't define the step, the default value is 0.
- Thus, the full syntax to create a range is `range(start, end, step)` with start and step being optional.

Operations supported

- To retrieve the length of a range, we use the function len.
- We can index a range the usual way, with the first integer in the progression being at index 0 and the last one being at index len-1.
- For example, for the range(1, 10, 2)

integer	**1**	**3**	**5**	**7**	**9**
index	0	1	2	3	4

- Ranges are immutable, so we can index to ask for a specific number in a range but not change it.
- Slicing also works as usual.

Dictionaries

Dictionaries

- **Dictionaries** are data structures that store pairs of keys and their associated values.
- Examples of dictionaries from the physical world:
  - English dictionary: key is a word in English, value is the definition in English
  - English-Spanish dictionary: key is a word in English, value is the translation in Spanish
  - address/phone book: key is the name of the person, value is their address/phone number
  - index at the end of a book: key is a term, value is a list of pages that the term appears
- The key has to be unique and we use it to "look up" (i.e. find) the associated value.
- Two keys can have the same associated value. E.g., two synonyms can have the same definition.
- We say that dictionaries represent a mapping from keys to values.
- Dictionaries also are called maps, symbol tables, or associative arrays.

Dictionaries in Python

- In Python, dictionaries are represented with the object type `dict`.
- Any immutable type can be a key. That means that lists are not allowed to be keys.
- Values can be any type of object, including lists.

Creating empty dictionaries

- To create an empty dictionary, we use the curly braces. E.g.,
  - `offices = {}`
- We can also use the `dict()` function. E.g., `offices = dict()`

Associating keys with values

- To insert a key-value pair to the dictionary we use the square brackets.
- offices['Alexandra Papoutsaki'] = 'Edmunds 222'
- offices['Dave Kauchak'] = 'Edmunds 220'
- offices is now {'Alexandra Papoutsaki': 'Edmunds 222', 'Dave Kauchak ': 'Edmunds 220'}
- We could also have created it as
  offices = {'Alexandra Papoutsaki': 'Edmunds 222', 'Dave Kauchak ': 'Edmunds 220'}
- len(offices) will return 2, since there are two key-value pairs in the dictionary.

Retrieving values

- Given a key, we can look up the associated value using the square brackets. For example:
    - offices = {}
    - offices['Alexandra Papoutsaki'] = 'Edmunds 222'
    - offices['Dave Kauchak '] = 'Edmunds 220'
    - offices['Alexandra Papoutsaki'] will return 'Edmunds 222'
- If we try to retrieve the associated value of a key that is not part of the dictionary, we will receive a KeyError. For example,
- offices['Eleanor Birrell'] would return KeyError: 'Eleanor Birrell'

Updating values

- When a key is already part of the dictionary, we can use it to update its associated value.
- For example,
  - `offices['Alexandra Papoutsaki'] = 'Edmunds 220'`
  - would update the value `'Edmunds 222'` to `'Edmunds 220'`.

in operator

- The in operator works in dictionaries, too. It tells us whether a *key* exists in the dictionary.
- 'Alexandra Papoutsaki' in offices would return True but 'Cecil Sagehen' would return False.
- To test whether a value exists in a dictionary, you can use the method values, e.g.,
- 'Edmunds 113' in offices.values()

pop/del/clear

- To delete a key-value pair from a dictionary, you have a few options:
  - The pop(key) method deletes the the key-value pair that matches the key **and** returns the associated value.
    - E.g., offices.pop('Alexandra Papoutsaki') would remove the pair **and** return 'Edmunds 220'
  - The del keyword deletes the key-value pair that matches the key
    - E.g., del offices['Alexandra Papoutsaki'] would remove the pair
  - The popitem() method removes and returns the last key-value pair to be added to the dictionary.
- To remove all key-value pairs and be left with an empty dictionary, you can use the clear method.
  - E.g., offices.clear() will result in offices={}
- del offices would delete the entire dictionary object (any other type of object), rather than empty it.

dict methods

- keys(): returns the keys of a dictionary in a type of object called dict_keys.
  - You can iterate through dict_keys using a for loop and use the in operator but you can not index or slice them. If you want, you can convert them into a list by passing them to the list() function.
- values(): returns the values of a dictionary in a type of object called dict_values.
  - You can iterate through dict_values using a for loop and use the in operator but you can not index or slice them. If you want, you can convert them into a list by passing them to the list() function.
- items(): returns the key value pairs of a dictionary in a type of object called dict_items.

```
for name, office in offices.items():
    print(name, office)
```

Practice time

- Write a function `get_counts` that given a list of (possibly duplicate) items, it creates a dictionary where the key-value pairs are the list items and the frequencies with which they appear in the list.

Answer

- Write a function get_counts that given a list of (possibly duplicate) items, it creates a dictionary where the key-value pairs are the list items and the frequencies with which they appear in the list.

```
def get_counts(data):
    counts = {}
    for element in data:
        if element in counts:
            counts[element] += 1
        else:
            counts[element] = 1
    return counts
```

Practice time

- Write a function `get_most_frequent` that given a list of (possibly duplicate) numbers, it returns a number-frequency tuple, where number is the one with the highest frequency in the list.
- Hint: you can reuse the `get_counts` function.

Answer

```
def get_most_frequent(data):
  counts = get_counts(data)
   max_key = 0
   max_value = -1
   for key in counts:
       if counts[key] > max_value:
           max_key = key
           max_value = counts[key]
   return (max_key, max_value)
```