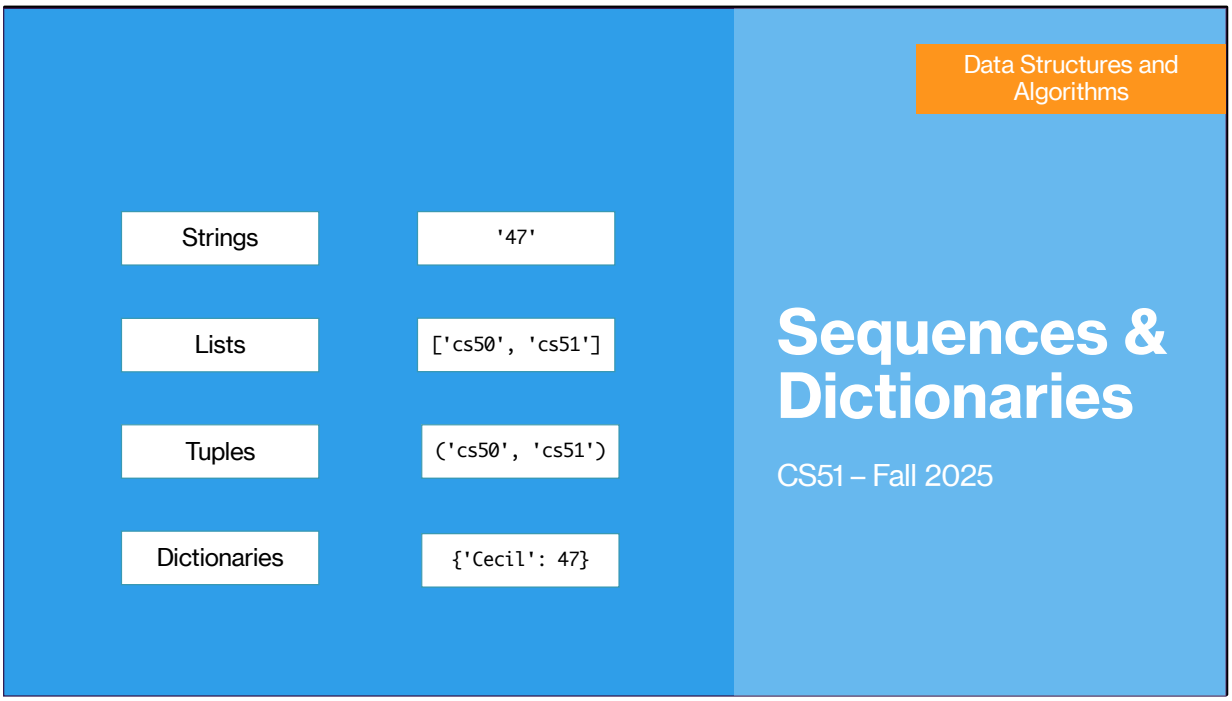



We are entering a new unit in the class, where we will be talking about major data structures and algorithms. This will allow us to understand the common ways that we can organize data, fundamental operations on them, and questions around how fast or slow our code is. We will start this new chapter by talking about sequences and dictionaries. Some of the topics we will cover from now on will seem familiar if you have taken CS50 or some other programming class in the past. We will also combine them with work we have already done on mathematical foundations of computer science. This unit will act as a preview of classes like CS62 and CS140.

## 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

2

A lot of times, we can organize data in our programs using some form of a sequence. Sequences are ordered collections (that is, there is a first, second, third, etc, element) of values. Sequences come in many forms. Together, we will see strings, lists, tuples, and range objects. What sequences have in common is that they support the operations of indexing, slicing, iteration (i.e. looping), and checking for membership. We will see what these mean shortly.




Let's see the first type of sequence, strings, which should largely feel familiar.

## 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 = ''`

4

A string is an ordered sequence of characters. A character can be a letter, a digit, a punctuation mark, or whitespace. Strings are enclosed in single or double quotes but by convention Python recommends we use single quotes. Here are a few examples of strings that contain a mixture of allowed characters.

## 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

5

We often need to know how long a string is. To do so, we use the function len and pass the string we are interested in as argument. Note that an empty string has a length of 0 and that whitespaces are counted toward the length.

## 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 |

6

Strings are indexable with the left-most character being at index 0. That means that the right most character is at index len-1. In other words, an index is the offset from the left-most character.

## 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

7

To isolate a single character from a string, we can pass its index in square brackets. We can then assign the returned character to a new string. The index can be an integer variable, as long as it's within the valid range of 0 and len-1. Otherwise, we will get an error that reminds us either that we must work with integers or that we are 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.

8

Python has a handy way of indexing from right to left that is not common in many popular programming languages. The last character is considered to be at index -1, rather than len-1 which is more verbose. That means, the second to last character is at -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.

9

To loop through a string, your first option is to use a for statement where you name a variable that will index through each character. A good name would be char or character rather than I here.

## 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
```

10

You can also use the range function if you want to keep track of the index. Here, it makes sense to name our variable something that i that commonly stands for index. By default, print() will add a space between each argument.

## 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
```

11

Finally, more conveniently, you can use the enumerate function to keep track of the index using the above syntax which circumvents the need to index a string.

## 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.

12

We saw that we can index specific characters using square brackets. Sometimes, we want to retrieve continuous segments of strings. These are known as slices. To select a slice, we again use square brackets but indicate the range of indices using a colon. Note that this will return the slice of the string from the start-th character we specified (inclusive) to the end-th character (exclusive).

## 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 `''`

13

Note that if the first index is greater than or equal to the second index after the colon, the slice is an empty string.

## 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]`

14

Start is an optional index. If we omit it, we assume that the entire slice from the $0^{th}$ character to the end character (Exclusive).

## 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)]`

15

Similarly, we can skip the end index to ask the slice from the start-th index to the end of the string.

## 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'`

16

If we skip both the start and end index, we receive. copy of the entire string.

## 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'`

17

We can additionally specify a step which defines the interval between characters. the default value is 1.

## 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'`

18

Perhaps unsurprisingly, if we define step but skip start, we can get all the characters from index 0 to end (excluded) at an interval of `step`.

## 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!'`

19

If we define start and step but not end, we have two colons in a row to distinguish it from string[start:end]

## 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 `'Io 5'`

20

Finally, we can skip both start and end but define a step which would return all the characters at an interval at the specicified step.

## 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`.

21

A key characteristic of strings is that they are immutable. That means we cannot change an existing string, e.g., we cannot swap a character with another one. We would have to create a new string and concatenate to it the character we want. Our original string remains unaffected.

## 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.

22

Let me pause here and say that so far in the course, we have seen different data types such as int, float, and strings. But despite these distinctions, almost everything in Python is an object. For example, the string `'cs51'` is an object. `type('cs51 ')` returns `<class 'str'>` that is it says that `'cs51' is an instance of the class str`. Similarly, the object 47 is an instance of the class `int`. Objects are central to the Object Oriented Programming paradigm. Python is an OOP language but often hides this fact and you can write plenty of code without realizing you are working with objects. For example, you can just have functions and global variables and have plenty of functionality. If you continue with CS62, you will learn Java, an OOP language that will force you to explicitly work with classes and 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)`

23

We won't do so in our class but I am bringing it up to understand why suddenly I will talk about methods. Methods are like special functions but they belong to objects and manipulate their state. To call a method, we don't just invoke its name like with functions. We have to call them through an object using the dot operator. Let's see what that means with a practical need when working with strings.

## 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'`

24

Let's say you are working with strings and want to strip any leading (starting) and trailing (ending) whitespaces. The class str supports a method called strip that does exactly that. Since strings are immutable, it doesn't change the actual string we call it on but returns a new string with these characters removed. The original string remains unchanged. We can optionally pass a delimeter, that is a character other than the default whitespace we might want to strip from the beginning and end (not middle!) of the string.

## 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.

25

Other useful methods that the str class supports are isupper and islower that return a bool for whether a string is all uppercase or lowercase. We can also use the lower and upper methods to return a new string that is all lowercase, or uppercase respectively.

## 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.

26

When comparing two strings, we can use the == operator (note the distinction from = which simply assigns the right hand to the left hand).
Similarly, we can compare two strings in lexicographic order. We say lexicographic and not alphabetic because we don't just compare letters. We can compare symbols and numbers, too.
The precedence is that symbols come before numbers and numbers come before letters. Peculiarly, all uppercase letters come before lowercase letters. That means that 'a'<'b' but 'B'<'a'.

# 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')`?

28

What do you think will happen if I call `compare_class('cs50')` and `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.
```

29

cs50 comes before cs51. But, since CS62 is capitalized, we get 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!'`.

30

Time to practice with strings by writing a function that returns a string that contains only the odd indexed characters of s.

## Answer – Option 1

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

31

You probably ended up with some sort of loop.

## 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
```

32

## Answer – Option 3

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

33

## Answer – Option 4

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

34

Or you could have sliced the string.

## 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`

35

Next challenge is to write a function that finds the index of the first instance of a string character or returns -1.

## Answer – Option 1

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



Again, multiple ways of going about it

## Answer – Option 2

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

37

## 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)
```

39

There is also another way that takes advantage of a built in function from the str class called find. If you want to see what methods the str class has, you can type help(str).

## Practice time

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

40

You are given the following string and expressions. What do they evaluate to?

## 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'`

41

## 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.

42

Let's work on this practice problem

## Answer

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

43

Did you come up with a different way?

## Practice time

- Write a recursive function called `reverse` that takes a string as a parameter and reverses the string.
- Remember the steps for writing a recursive function:

1. Define what the function the name and parameters of the function are.
2. Define the recursive case
3. Define the base case
4. Put it all together

44

Your next practice problem is to write a function that reverses a string recursively.

# Answer

1. Define what the function the name and parameters of the function are.
   - `def reverse(some_string)`
2. Define the recursive case
   - Pretend like we have a function called `reverse` that we can use but only on smaller strings
   - To reverse a string:
     - remove the first character,
     - reverse the remaining characters,
     - put that first character at the end
   - The recursive relationship is:
     - `reverse(some_string) = reverse(some_string[1:]) + some_string[0]`

# Answer

3. Define the base case
    - in each case, the string is getting shorter.
    - Eventually, it will be an empty string.
- 4. Put it all together

```
def reverse(some_string):
    if some_string == '':
        return ''
    else:
        return reverse(some_string[1:]) + some_string[0]
```

Could have also worked the other direction,
`return some_string[-1] + reverse(some_string[:-1])`

46

## Answer

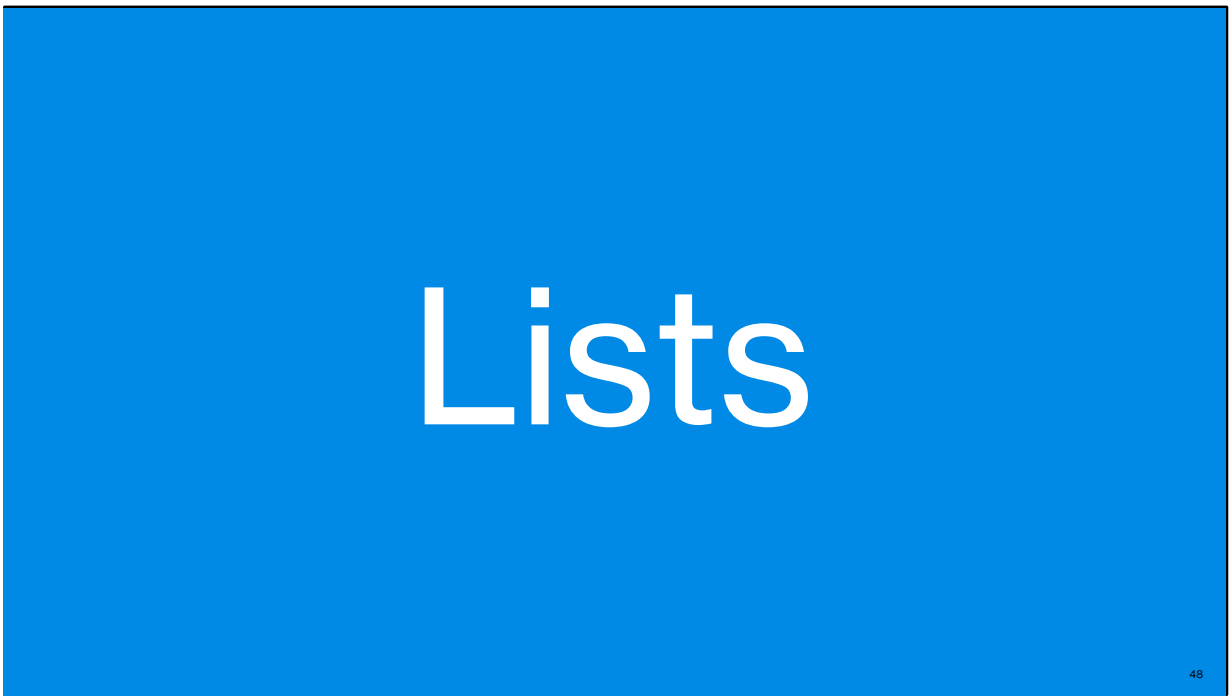
How does this work? Let's look at an example:

```
>> reverse('cs51')
  >> reverse('s51') + 'c'
     >> reverse('51') + 's'
        >> reverse('1') + '5'
             >> reverse('') + '1'
                  >> '' + '1'
        >> '1' + '5'
     >> '15' + 's'
  >> '15s' + 'c'
>> '15sc'
```

```
def reverse(some_string):
    if some_string == '':
        return ''
    else:
        return reverse(some_string[1:]) + some_string[0]
```

47

This is what happens in the call stack.




We are now ready to move to our next type of sequence, 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**.

49

Like strings, lists are sequences of values, but they are not limited to characters. The values, known as list elements, can be any type.

## 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']`.

50

To create a list, you enclose the elements in square brackets and separate them by comma. You can also use the list function and pass it an iterable element like a string.

## Types of elements in a list and nested lists

- Python allows elements of different types in the same list (not all languages allow this!)
  - `bizarre_list = [47, 'cs51', ['Feta','Cheddar', 'Edam', 'Gouda']]`
  - You can even have elements of lists be lists themselves.
  - A list within another list is **nested**.

51

Bizarrely, Python allows a list to hold elements of different type. Even lists themselves can be elements of another list, resulting to nested lists.

## 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()`

52

To create an empty list, you can either use the empty square brackets or the empty list function.

## 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
  - `bizarre_list = [47, 'cs51', ['Feta','Cheddar', 'Edam', 'Gouda']]`
  - `len(bizarre_list)`
    - It will return 3.

53

Similarly to strings, we can inquire on the length of a list using the len function. Note that a nested list only contributes 1 to the total number of elements.

## 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 |

54

Lists are sequences and we said that sequences are indexable. Again, we start our indexing from 0 and go to the right.

## 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`.

55

As before, we can ask for a specific element in a list by passing its index in the square brackets.

## 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`.

56

In contrast to strings though, lists are mutable which means we can change the elements in a list, using the square brackets to index the element we want to change. Of course, our index needs to be an integer in the valid range of 0 to len-1.

## `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`.

57

Belonging can be checked using the in operator.

## `in` operator for nested lists

- Although a list can contain another list, the nested list still counts as a single element – so in the following list, there are only three elements.
- `bizarre_list = [47, 'cs51', ['Feta','Cheddar', 'Edam', 'Gouda']]`
- Remember, `len(bizarre_list)` returns 3.
- `'Feta' in bizarre_list` would return `False`
- `['Feta','Cheddar', 'Edam', 'Gouda'] in bizarre_list` would return `True`

58

Note that in nested lists, a list is an element.

## 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.

59

Slicing is similar to what we already saw with strings.

## + 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.

60

To concatenate two lists into a new list, we would use the + operator.

## * 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.

61

To repeat a list's elements at a given number of times, you can use the * operator.

## 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'`

62

Other useful operations would be to sum all elements or take the min amd max element of a list.

## 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']`

63

To add elements into a list, we have three options. append which adds a new element to the end of a list, extend, which takes a list as an argument and appends all of its element, and insert which inserts a new element to the specified index.

## 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.

64

To remove elements, we can use either pop which removes amd returns an element from a specified index. You can also use the remove method which removes the first instance of an element.

## 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']`

65

We already saw that we cam convert a string to a list of indiviual characters. You can also use the split method to break a string into a list of its words.

## 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']`

66

Or you can specify a delimiter and split it based on that.

## 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.

67

Another intersection of strings and lists is that if you have a list of strings, you can concatenate them into a single string using the join string method, kinda weirdly, on the delimiter you want to use to join the list of strings on.

## 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
```

68

Looping through lists is really similar to what we have seen with strings.

## 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
```

69

## 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
```

70

## 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.



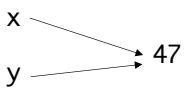

- More practice with references: https://pythontutor.com/
- Read more about references: https://nedbatchelder.com/text/names.html

71

We already talked about everything in Python being an object. Let's say we have an assignment x=47 which as we know associates the name x with the value 47. We say that the name x is a reference to the value 47. Schematically, in memory we can think of x pointing to 47

## References

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



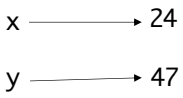

- Assigning a value to a name never copies the data, it never makes a new value. Assignment just makes the name on the left refer to the value on the right.

72

Let's say after we assign 47 to x we associate y with x. Now both x and y refer to the same value, 47. Neither x or y is the “real” name. They have equal status: each refers to the value in exactly the same way. 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.




73

What do you think will happen if we next assign the reference x to the value 24? That's right, x will be reassigned but y will stay unchanged. They won't always be the same forever.

## 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:

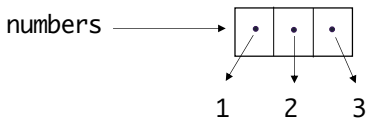



74

Remember, Immutable values include numbers, strings, and tuples, which we will see next. Almost everything else is mutable, including lists, dicts, and user-defined objects. Mutable means that the value has methods that can change the value in-place. Immutable means that the value can never change, instead when you think you are changing the value, you are really making new values from old ones. That's why x+1 computes an entirely new value, which is then assigned to x.

## References

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




- For simplicity

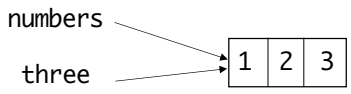



75

Let's move to lists next. What do you think will happen here? Truly we create a reference numbers to a list of three elements. Each of these elements is really a reference to a list value. For simplicity though we typically hide this part, but it is important to remember it especially when working with nested lists.

## 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.

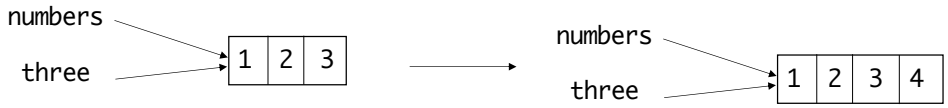



76

How about now?
  ```

## 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:

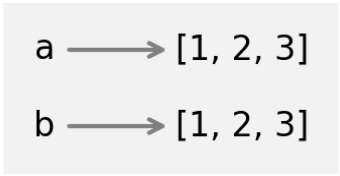



77

Let's say we next mutate numbers. This change will be reflected on three, too, as it references the same list.
  ```

## 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!

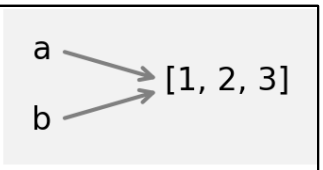



- In this case we would say that the two lists are **equivalent**, because they have the same elements, but not **identical**, because they are not the same object.
- If two objects are identical, they are also equivalent, but if they are equivalent, they are not necessarily identical.

78

Let's say that we now have two lists that have the same contents. Are they identical? No they are not. If two objects are identical, they are also equivalent, but if they are equivalent, they are not necessarily identical.

## Aliasing



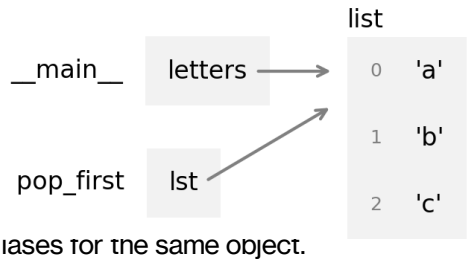

- An object with more than one reference has more than one name, so we say the object is **aliased**. If the aliased object is *mutable*, changes made with one name affect the other.
- In this example, if we change the object *b* refers to, we are also changing the object *a* refers to.
- So we would 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.

79

When an object has more than one reference, we say that its aliased. If the aliased object is mutable, changes made with on the name affect the other which can cause trouble if we are not careful when working with mutable aliased objects.

## 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']`.




80

Let's say we pass a list to a function and the function gets a reference to the list. If the function modifies the list, the function caller sees the change.

## 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)
  ```

81

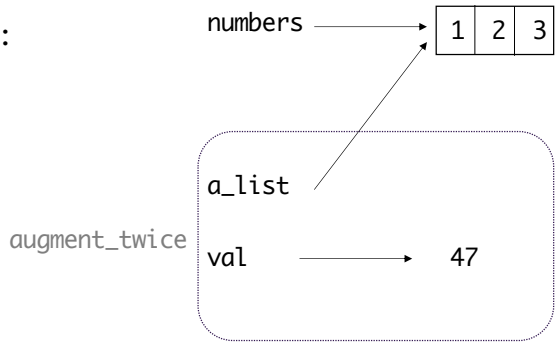
What do you think will happen after the call to augment_twice here?
  ```

## 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)
```




82

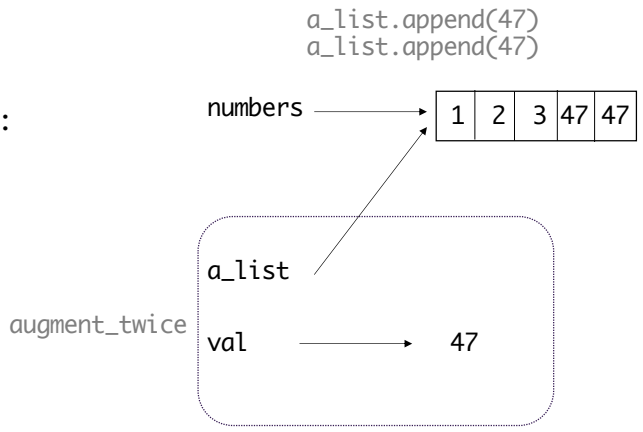
The local names in the function are drawn in a new frame. Calling the function assigned the actual values to the parameter names, just like any other assignment statement. Remember that assignment never makes new values or copies any data, so here the local name a_list refers to the same value that was passed in, numbers.

## 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)
```




83

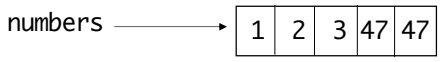
Then we call a_list.append twice, which mutates the list

## 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)
  ```




84

When the function ends, the local names for local variables and parameters are destroyed. Values that are no longer referenced are reclaimed, but global ones remain. This is how we are left with numbers still referencing. We passed the list into the function, which modified it. No values were copied. Although this behavior might be surprising, it's essential. Without it, we couldn't write methods that modify objects.
  ```

## 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)
  ```

85

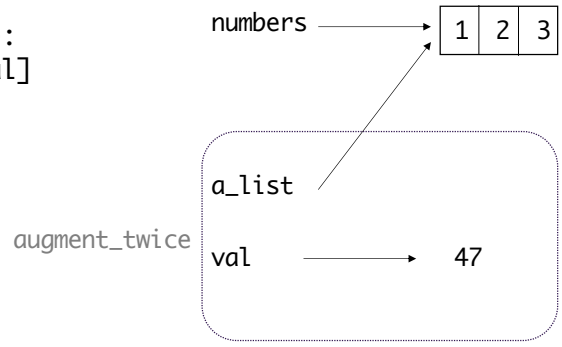
What do you think will happen after the call to augment_twice here?
  ```

## 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)
```




86

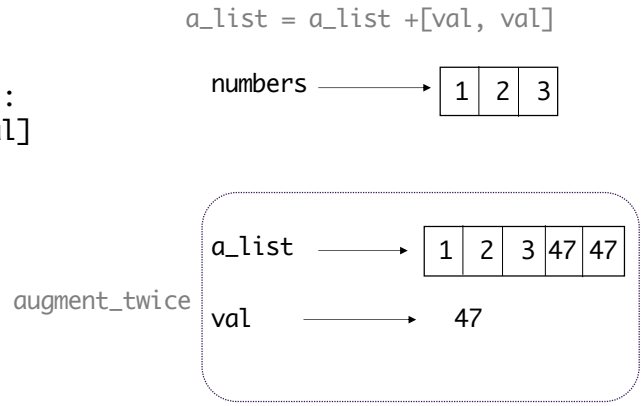
At the moment we call augment_twice, it looks the same as we saw with the earlier version of augment_twice

## 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)
  ```




87

The next statement is an assignment. The expression on the right-hand side makes a new list, which is then assigned to a_list
  ```

## 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)
```




88

When the function ends, its local names are destroyed, and any values no longer referenced are reclaimed, leaving us just where we started. It's really important to keep in mind the difference between mutating a value in place, and rebinding a name. augment_twice the first time worked because it mutated the value passed in, so that mutation was available after the function returned. the second implementation of augment_twice used an assignment to rebind a local name, so the changes weren't visible outside the function. You can read more here https://nedbatchelder.com/text/names.html

## 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!

89

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.

## Practice time

- Write a recursive function `rec_sum_list` that takes a list of numbers and returns the sum of all its elements.

90

Lets practice with lists by writing a recursive function that takes a list of numbers and returns the sum of all its elements.

# Answer

1. Define what the function the name and parameters of the function are.
   - `def rec_sum_list(some_list)`
2. Define the recursive case
   - Pretend like we have a function called `rec_sum_list` that we can use but only on smaller lists
     - what would we get back if we called `rec_sum_list` on everything except the first element?
       - the sum of the remaining elements
     - how would we get the sum to the entire list?
       - just add that element to the sum of the rest of the elements
   - The recursive relationship is:
     - `rec_sum_list(some_list) = some_list[0] + rec_sum_list(some_list[1:])`

91

## Answer

3. Define the base case
    - in each case, the list is getting smaller.
    - Eventually, it will be an empty list. What is the sum of an empty list?
        - 0.

4. Put it all together

```
def rec_sum_list(some_list):
    if some_list == []:
        return 0
    else:
        return some_list[0] + rec_sum_list(some_list[1:])
```

92

## Answer

How does this work? Let's look at an example:

```
>> rec_sum_list([1, 2, 3, 4])
  >> 1 + rec_sum_list([2, 3, 4])
     >> 2 + rec_sum_list([3, 4])
        >> 3 + rec_sum_list([4])
           >> 4 + rec_sum_list([])
              >> 4 + 0
        >> 3 + 4
     >> 2 + 7
  >> 1 + 9
>> 10
```

```
def rec_sum_list(some_list):
    if some_list == []:
        return 0
    else:
        return some_list[0] +
rec_sum_list(some_list[1:])
```

93

This is the stack of the calls that happen. You can also use print statements to see the recursion.

## Practice time

- What does this mystery function do?

```
def rec_mystery(l):
if len(l) == 1:
   return l[0]
else:
    m = rec_mystery(l[1:])
    if m > l[0]:
        return m
    else:
        return l[0]
```

94

What do you think that the rec_mystery function does?

## Answer

```
def rec_mystery(l):
    if len(l) == 1:
        return l[0]
    else:
        m = rec_mystery(l[1:])
        if m > l[0]:
            return m
        else:
            return l[0]
```

- Recursive function. Work through a small example:

```
>> rec_mystery([2, 4, 3, 1])
   >> m = rec_mystery([4, 3, 1])
         >> m = rec_mystery([3, 1])
               >> m = rec_mystery([1])
                     >> 1
               >> compares 3 and 1 and returns 3
         >> compares 4 and 3 and returns 4
   >> compares 2 and 4 and returns 4
```

- Returns the maximum element in the list!

95

I have outlined above what happens at the end of each recursive call. It returns the maximum element in the list.

## Practice time

- Write an iterative Python function called `linear_search` that takes a list `lst` and anelement `element` and returns the index of the first encounter of the `element` in `lst`, if it exists, or -1 if it does not.
- What are the pre- and post-conditions for your program?
- Write a loop invariant and prove that your program is correct

96

Time to tie our work on proofs with loop invariants and lists. Write an iterative Python function called `linear_search` that takes a `list` and an `element` and returns the index of the first encounter of the `element` in the list, if it exists, or -1 if it does not.
What are the pre- and post-conditions for your program?
Write a loop invariant and prove that your program is correct

## Answer

- Pre-condition : A list `lst` of n elements, $[e_0, e_1, \ldots, e_{n-1}]$ and an element `element` to search for.
- Post: The index `i` of the element $e_i$, so that $e_i == list[i] == element$ and $list[e_j] != element$, for every $j < i$, or -1 if no element in list is equal to $element$.

```
def linear_search(lst, element):
    n = len(lst)
    for i in range(n):
        if lst[i]==element:
            return i
    return -1
```

```
def linear_search(lst, element):
    i = 0
    n = len(lst)
    while (i<n):
        if lst[i]==element:
            return i
        i+=1
    return -1
```

97

Your code is probably one of the two, or you might have used enumerate.

## Is our output correct?

- To show that the output is correct we would need to know that:
  - If index `i` is returned, then `lst[i]==element` and `i` is the first index with a matching element.
  - If -1 is returned, then the `element` is not in list.

98

What does it mean for our output correct?

## Loop invariants

- Assuming the precondition holds, a loop invariant for a loop is a condition that is true at the beginning and end of every iteration of a loop
- To prove correctness, we need to show:
  - *Initialization*: The loop invariant is true before the loop runs for the first iteration.
  - *Maintenance*: If the loop invariant is true before an iteration, it remains true after the execution of the loop body and before entering the next iteration.
  - *Postcondition*: Show that if the guard is false (so the loop exits), then the desired result of the loop has been achieved: the loop post-condition holds.
  - *Termination*: After a finite number of iterations of the loop, the guard will become false and the loop will terminate.
- The first three steps succeed would prove partial correctness. The fourth step would be necessary to show full correctness.

99

These steps are just a reminder of loop invariants.

## Loop invariants

- Loop invariants, often make a statement depending on an index i and what you have seen so far, e.g., in `lst[0:i]` (this list slice contains all elements from index 0 to index `i-1`, that is 0 is inclusive but `i` is exclusive).
- Let's go back to the linear search example. What loop invariant can we can come up with?
  - At the start of each iteration, element does not exist in the sublist lst[0:i]

100

What would be a good loop invariant for linear search?

## Loop invariant for linear search - Initialization

- *Initialization*: Before the first iteration, `i==0`, then the slice `lst[0:0]` would be empty. Therefore, `element` would not be in it and the loop invariant holds true.

```
def linear_search(lst, element):
    n = len(lst)
    for i in range(n):
        if lst[i]==element:
            return i
    return -1
```

```
def linear_search(lst, element):
    i = 0
    n = len(lst)
    while (i<n):
        if lst[i]==element:
            return i
        i+=1
    return -1
```

101

We will work through each of the steps of loop invariants together.

# Loop invariant for linear search - Maintenance

- *Maintenance*: Let's assume the loop invariant is true at the start of the `i`-th iteration of the loop, that is the `element` does not exist in `lst[0:i]`. If `lst[i] == element`, the current iteration is the final one (the `return` statement would be reached); otherwise, if `lst[i] != element`, we have that the loop invariant holds true at the end of this iteration and before the beginning of the next iteration, since `element` is not in `lst[0:i+1]`.

```
def linear_search(lst, element):
    n = len(lst)
    for i in range(n):
        if lst[i]==element:
            return i
    return -1
```

```
def linear_search(lst, element):
    i = 0
    n = len(lst)
    while (i<n):
        if lst[i]==element:
            return i
        i+=1
    return -1
```

# Loop invariant for linear search - Postcondition

- *Postcondition:* The loop terminates when
  - i) the guard fails, that is `i==n==len(lst)`. In that case, the `element` has not been found, `lst[0:n]` does not contain the `element`, and -1 is correctly returned.
  - ii) the `return` statement is reached because `lst[i]==element` for the first time, and `i` is correctly returned as the index of first occurrence of `element` in the list (since we know that it couldn't have been in the `list[0:i]`).

```
def linear_search(lst, element):
    n = len(lst)
    for i in range(n):
        if lst[i]==element:
            return i
    return -1
```

```
def linear_search(lst, element):
    i = 0
    n = len(lst)
    while (i<n):
        if lst[i]==element:
            return i
        i+=1
    return -1
```

103

# Loop invariant for linear search - Termination

- *Termination*: `i` increases by 1 in every iteration and ranges from 0 to maximum `n=len(lst)`. Thus, in a finite number of iterations of the loop, we will either find the `element` and return its index or return -1.

```
def linear_search(lst, element):
    n = len(list)
    for i in range(n):
        if lst[i]==element:
            return i
    return -1
```

```
def linear_search(lst, element):
    i = 0
    n = len(lst)
    while (i<n):
        if lst[i]==element:
            return i
        i+=1
    return -1
```

104

## 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.

105

Your turn to write a recursive linear search function that takes a list, an element, and index that has a default value of 0.

## 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)
```

106

## 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.

Let's go back to loop invariants with one more practice problem.

## 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

110

Tuples is another type of a sequence

## 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.

111

Like lists, tuples can have values of any type but in contrast to lists are immutable thus make a good choice if we can safely assume that we won't need to change their values.

## 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')`

112

To create a tuple, we write a comma-separated sequence of values that are optionally enclosed in parentheses. Be careful about distinguishing when tuples vs other types are created. There is also a built in tuple function.

## 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')`

113

A lot of operations we have seen so far, like indexing, slicing, concatenating, and duplication also exist in tuples.

## 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`.

114

The key thing to remember about tuples is that they are immutable and thus if you try to modify them you will get an error.

## 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)`



You can put a tuple of variables on the left side of an assignment, and a tuple of values on the right. The values are assigned to the variables from left to right.

## 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`.

116

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. That can be useful if for example, we want to split a string and instead of working with a list we want to pass the elements to a tuple.

# Range objects



The final sequence we will see is range.

## 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.

118

A range is a sequence that represents an arithmetic progression of integers. You are already familiar with range objects when you called the range function.

## 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.

119

Operations supported for ranges should feel familiar by now. You can see the length of range, slice, and index. But ranges are immutable!

# Dictionaries

120

The final data structure we will see today is that of 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.

121

Dictionaries are data structures that store pairs of keys and their associated values, similar to a physical dictionary, address book, or index at the the end of the book. Keys have to be unique and we use them to look up, i.e. find the associated values. But two keys can have the same value. Because dictionaries represent a mapping from keys to values, they are known as maps. Other names for them are symbol tables or assocative rrays.

## 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.

122

In Python, dictionaries are represented using the dict data type. Any immutable type can be a key (that means no lists). Lists are fine for values because values can support any type of object.

## 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()`

123

To create an empty dictionary, we either use the curly brackets or the dict built in function. To associate a value

## Associating keys with values

- To insert a key-value pair to the dictionary we use the square brackets.
- `offices['Alexandra Papoutsaki'] = 'Edmunds 222'`
- `offices['Joseph Osborn'] = 'Edmunds 113'`
- offices is now `{'Alexandra Papoutsaki': 'Edmunds 222', 'Joseph Osborn': 'Edmunds 113'}`
- We could also have created it as
  `offices = {'Alexandra Papoutsaki': 'Edmunds 222', 'Joseph Osborn': 'Edmunds 113'}`
- `len(offices)` will return 2, since there are two key-value pairs in the dictionary.

124

To insert a key-value pair to the dictionary we use the square brackets. We can also create a dictionary in one go using curly brackets.

## 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['Joseph Osborn'] = 'Edmunds 113'`
  - `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['Dave Kauchak']` would return `KeyError: 'Dave Kauchak'`

125

Given a key, we can look up the square brackets, as long as the key is part of the dictionary.

## 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'`.

126

We can also update the associated value of an already inserted key-value pair using square brackets.

## `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()`

127

Dictionaries support the in operator which tests whether a key belongs to a dictionary. To test belonging in values you have to use the values method.

## 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.

128

To delete a key-value pair you can either use the pop method, the del keyword, or the popitem method. You can also clear all key-value pairs and get the empty dictionary. You can also delete altogether a dictionary or any object.

## 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)
```

129

Some useful methods for dictionaries to know are keys, values, and items.

## 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.

130

Let's try together to write a function that combines lists and dictionaries.

# 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 val in data:
        if val in counts:
            counts[val] += 1
        else:
            counts[val] = 1
    return counts
```

131

## 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.

132

Now use this function to built another one.

## 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)
```