SI 506 Lecture 05

Topics

1. Sequences: strings and lists
2. Indexing
3. Slicing
4. String and list methods
5. Select `str` methods
6. Select `list` methods
7. String formatting

Vocabulary

- **Concatenation**. Joining one object to another in order to create a new object. Joining two strings together (e.g., `greeting = 'Hello ' + 'SI 506'`) is an example of string concatenation.
- **Index**. Numeric position of an element or item contained in an ordered sequence. Python indexes are zero-based, i.e., the first element's index value is 0 not 1.
- **Iterable**. An object capable of returning its members one at a time. Both strings and lists are examples of an iterable.
- **Sequence**. An ordered set such as `str`, `list`, or `tuple`, the members of which (e.g., characters, elements, items) can be accessed individually or in groups.
- **Slice**. A subset of a sequence. A slice is created using the subscript notation `[]` with colons separating numbers when several are given, such as in `variable_name[1:3:5]`. The bracket notation uses slice objects internally.

w3schools string and list methods reference pages

Open the following w3schools reference pages in your browser and bookmark them. The pages provide useful summaries of `str` and `list` methods.

1. w3schools, "Python String Methods"
2. w3schools, "Python List Methods"

Lecture data

It is quite natural to assume that the Python programming language is named after the family of snakes known as *Pythonidae* or python. But you would be wrong. Guido van Rossum, the creator of the Python programming language named it after the absurdist English comedy sketch series *Monty Python's Flying Circus* (1969-1974) which starred the "Pythons" Graham Chapman, John Cleese, Eric Idle, Terry Jones, Michael Palin and the animator Terry Gilliam.

Today's lecture features data derived from the Pythons' famous "Spam" sketch (1970) including the Spam dominated cafe menu. During the Second World War and after Britain imposed rationing restrictions and, starting in 1941, imported massive quantities of canned spam from the United States as a protein substitute for imports of beef, pork, and poultry. The public, including my parents, grew to loathe it--which the sketch plays upon in surrealist fashion.

💡 Have you ever wondered why unwanted email is referred to as "spam". Watch the "Spam" sketch and you'll quickly understand why.

1.0 Sequences: strings and lists

This week we discuss Python data structures, focusing in particular on two sequence types: strings and lists.

1.1 String basics

A string (type: `str`) is an ordered sequence of characters. Once created, the string is considered *immutable* and cannot be modified. The string is also an *iterable*, a type of object whose members (in this case, characters), can be accessed.

String objects (an instance of the `str` class) are also provisioned with *methods* that permit operations to be performed on the string. These behaviors are discussed in greater detail during the next lecture.

```
# A string
comedy_series = 'Monty Python'

# The object's unique identifier in memory
comedy_series_id = id(comedy_series)

# Return the object's type
comedy_series_type = type(comedy_series)

# Return the object's length
comedy_series_len = len(comedy_series)
```

You can confirm that a string is immutable by attempting to change one of its characters:

```
# UNCOMMENT: Immutability check
# comedy_series[0] = 'm' # TypeError: 'str' object does not support item
assignment
```

💡 You can use the plus (`+`) operator to build a string. This is known as string *concatenation*.

In the example below the variable `comedy_series` is (re)assigned to a new string object comprising the value of `comedy_series` plus the hard-coded string `"'s Flying Circus"`. The output of `print()` demonstrates that the new concatenated string is assigned a new identity that remains unchanged for the life of the object.

```
# String concatenation
comedy_series = comedy_series + "'s Flying Circus" # string concatenation
(new object)
```

```
print(f"\n1.1 comedy_series (id={id(comedy_series)}) = {comedy_series}") #
f-string
```

1.2 List basics

A list (type: `list`) represents an *ordered* sequence of elements (e.g., strings, lists, and/or other object types). The list is also an *iterable*, a type of object whose members (in this case, characters), can be accessed. Unlike a string a Python list is mutable and capable of modification. Elements can be added or removed from a list and, if the element is mutable, (e.g., a nested list) can be modified. List elements are accessed by position using a zero-based index value.

List objects are also provisioned with methods that permit operations to be performed on the list. These behaviors are discussed in greater detail in a later section.

```
# A list
pythons = [
    'Graham Chapman',
    'John Cleese',
    'Terry Jones',
    'Eric Idle',
    'Michael Palin'
    ]

# The object's unique identifier in memory
pythons_id = id(pythons)

# Return the type
pythons_type = type(pythons)

# Return the length
pythons_len = len(pythons)
```

Unlike a string a list can be *mutated* (i.e., modified) by adding, updating, substituting, and/or removing elements. In the example below Terry Gilliam, the American animator (and later director), was also a Python so let's add him to the list `pythons`. We can utilize a number of list methods to accomplish the task. Each method call mutates the list *in-place*, returning `None` implicitly to the caller. We will discuss list methods in greater detail at our next meeting.

```
# In-place method call mutates the list
pythons.append('Terry Gilliam')
# pythons.insert(-1, 'Terry Gilliam')
# pythons.extend(['Terry Gilliam'])
```

 You can also create a *new* list by *concatenating* two or more lists using the plus (`+`) operator. In the example below a single element list containing the string 'Neil Innes' (considered by many to be the seventh

Python) is added to the `pythons` list. This results in a new list which is assigned to the existing variable `pythons`.

```
# List concatenation
pythons = pythons + ['Neil Innes']
```

2.0 Indexing

You can access individual members of a sequence by their position or index value. Python's index notation is **zero-based**. Individual characters in a string or individual elements in a list can be accessed using an index operator. Index operator notation employs two brackets `[< index >]` enclosing either a positive or negative index value (eg., `some_sequence[0]`).

0	1	2	3	4	5	6	7	8	9	10	11
M	o	n	t	y		P	y	t	h	o	n
-12	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

2.1 Accessing a character by position

```
name = 'Monty Python'
letter = name[0] # first letter (zero-based index)

letter = name[4]

letter = name[-1]
```

 `name[0]` is considered an expression since it resolves to a value (e.g., "M").

2.2 Accessing a list element by position

In the example below, the second item in the Bromley cafe menu (`'Egg, sausage and bacon'`) is accessed using a positive index value while the second to the last item in the menu (`'Spam, Spam, Spam, Spam, Spam, Spam, baked beans, Spam, Spam, Spam and Spam'`) is accessed using a negative index value.

```
menu = [
    'Egg and bacon',
    'Egg, sausage and bacon',
    'Egg and Spam',
    'Egg, bacon and Spam',
    'Egg, bacon, sausage and Spam',
    'Spam, bacon, sausage and Spam',
    'Spam, egg, Spam, Spam, bacon and Spam',
```

```
    'Spam, Spam, Spam, egg and Spam',
    'Spam, Spam, Spam, Spam, Spam, Spam, baked beans, Spam, Spam, Spam and
Spam',
    'Lobster Thermidor aux crevettes with a Mornay sauce, garnished with
truffle pâté, brandy and a fried egg on top and Spam'
    ]

menu_item = menu[1] # second element (zero-based index)

menu_item = menu[-2]
```

2.3 IndexError

If an index value references a non-existent position in a sequence an `IndexError` will be raised.

```
# UNCOMMENT
# menu_item = menu[10] # IndexError: list index out of range
```

3.0 Slicing

You can access a `list` element, `tuple` item, or `str` character by position using an index operator. You can also access a subset or *slice* of elements, items, or characters using Python's slicing notation.

To initate a slicing operation specify a range of index values by extending the index operator to include an *optional* integer `start` value, a *required* integer `end` value that specifies the position in which to end the slicing operation, and an *optional* `stride` value that specifies the slicing step (default = 1).

The slicing notation syntax simplifies referencing and/or extracting a subset of a given sequence. List slicing can result in list traversal performance gains since slicing obviates the need to loop over an entire list in order in order to operate on a targeted subset of elements. We will explore this aspect of slicing when we explore list iteration in more detail starting next week.

```
cast = [
    'Terry Jones, Waitress',
    'Eric Idle, Mr Bun',
    'Graham Chapman, Mrs Bun',
    'John Cleese, The Hungarian',
    'Michael Palin, Historian',
    'Extra, Viking 01',
    'Extra, Viking 02',
    'Extra, Viking 03',
    'Extra, Viking 04',
    'Extra, Viking 05',
    'Extra, Viking 06',
    'Extra, Police Constable'
]
```

3.1 Slicing start/end range

💡 In the slicing example below the start value 1 is considered *inclusive* while the end value 3 is considered *exclusive*.

```
# Return Mr and Mrs Bun.
cast_members = cast[1:3] # Returns ['Eric Idle, Mr Bun', 'Graham Chapman,
Mrs Bun']
```

Negative slicing can also be employed to return the Buns:

```
# Return Mr and Mrs Bun.
cast_members = cast[-11:-9]
```

Let's explore more examples.

3.2 slice from index 0 to index n (stride = 1)

```
# Return named cast members.
cast_members = cast[:5] # or cast[0:5]
```

3.3 slice from index -n to end of list inclusive (stride = 1)

```
# Return cast extras (i.e., Vikings 01-06, Police Constable) using
negative slicing.
cast_members = cast[-7:] # warn: not the same as cast[-7:-1]
```

3.4 slice with a specified stride

You can set a stride value to increase the number of steps taken by each slice.

```
# Return cast members in reverse order.
cast_members = cast[::-1]
```

```
# Return every other cast member starting from the first element.
cast_members = cast[::2]
```

```

```

```
# Return every other cast member starting from the last element (negative
stride).
cast_members = cast[::-2] # reverse
```

```
# Return every other Viking starting with Viking 01.
cast_members = cast[5:11:2]
```

```
# Return every other Viking starting with Viking 01 in reverse order.
cast_members = cast[5:11:-2] # empty list returned

# Workaround
cast_members = cast[5:11]
cast_members = cast_members[::-2]
```

Challenge 01

1. Employ slicing to access the elements in `cast` that represent John Cleese and Michael Palin using *negative* index values. Assign the return value (a new list) to the variable `cleese_palin`.
2. Employ slicing to access the elements in `cast` that represent Terry Jones, Graham Chapman, and Michael Palin using *positive* index values. Assign the return value (a new list) to the variable `jones_graham_palin`.

3.5 Slice Assignment

You can replace a subset of a list with another list or subset of a list using slice assignment.

```
mounties = [
    'Extra, Canadian Mountie 01',
    'Extra, Canadian Mountie 02',
    'Extra, Canadian Mountie 03',
    'Extra, Canadian Mountie 04',
    'Extra, Canadian Mountie 05',
    'Extra, Canadian Mountie 06'
]
```

```
# Replace part of a list (length unchanged).
cast[5:11] = mounties[0:] # replace Vikings with Mounties
```

```
# Replace part of a list (length changes).
cast[5:11] = mounties[1:5] # replace Vikings with mounties 02-04 (negative
```

```
slice)
```

3.6 Built-in del() function and slicing

You can employ slicing and the built-in `del()` function to remove subsets of a sequence.

```
# Delete the Mounties (retain the Police Constable)
del(cast[-5:-1])
# del(cast[5:9]) # alternative
```

3.7 built-in slice() function

You can also use the built-in `slice()` function to return a slice object and apply it to a sequence. `slice()` accepts three arguments: an optional integer `start` value (default = 0), a required integer `end` value that specifies the position in which to end the slicing operation, and an optional `step` value that specifies the slicing step (default = 1).

```
# slice([start, ]end[, step]) object
s = slice(1, 4, 2)
cast_members = cast[s] # Returns Idle and Cleese
```

4.0 String and list methods

When you create a string or list you create an object that is based on a `class`, a type that can both hold data and perform actions. Think of a `class` as a template, blueprint, or model for creating objects. Each string or list that you create and assign to a variable represents an *instance* of the class upon which it is based (i.e., the class types `str` and `list`). Such objects possess individual characteristics (data) and common behaviors (methods).

Object methods, if defined, are *called* using dot (`.`) notation. If a method "signature" includes one or more *parameters*, these can be passed to the method as comma-separated *arguments* that are included inside the method name's parentheses. If a method definition does not specify an *explicit* return value, `None` (type: `NoneType`) will be returned implicitly to the caller.

```
menu_item = 'Spam, egg, Spam, Spam, bacon and Spam'

# str.lower() -- no argument method
menu_item_lower = menu_item.lower()

# str.count(value, start=0, end=len(str) - 1) -- start and end are
optional
spam_count = menu_item.count('Spam')

# str.split(sep=' ', maxsplit=-1) -- sep and maxsplit are optional
items = menu_item.split(', ') # returns list
```

```
# list.remove(element) -- in-place operation; removes 1st occurence;
returns None
items.remove('egg')

# Do not do this: items variable no longer points to a list object
items = items.remove('bacon and Spam') # None is returned
```

Method calls can also be "chained". Each method call returns a value (object) to which the next method call is bound. Order matters. Note that calling a method not defined for an object will raise an `AttributeError`.

```
menu_item = 'Egg, bacon, sausage and Spam'

# Good. Replace, convert to lower case, and split.
items = menu_item.replace(' and', ',').lower().split(', ')

# Bad. The trailing list.append() returns None (oops!)
items = menu_item.replace(' and', ',').lower().split(',
').append('pancakes')

# Ugly. Premature split. Calling lower on a list object raises a runtime
error
# AttributeError: 'list' object has no attribute 'lower'
items = menu_item.replace(' and', ',').split(', ').lower()
```

5.0 Select `str` methods

String objects (type `str`) include "built-in" behaviors that are defined as *methods*. Calling a `str` method may require that one or more arguments (values) be passed to it in order to perform the requested computation. Depending on the method definition, the operation may result in a computed value being returned to the caller.

Below is a select list of `str` methods. For a more complete list of available `str` methods see the w3schools Python String Methods page.

```
menu = [
    'Egg and bacon',
    'Egg, sausage and bacon',
    'Egg and Spam',
    'Egg, bacon and Spam',
    'Egg, bacon, sausage and Spam',
    'Spam, bacon, sausage and Spam',
    'Spam, egg, Spam, Spam, bacon and Spam',
    'Spam, Spam, Spam, egg and Spam',
    'Spam, Spam, Spam, Spam, Spam, Spam, baked beans, Spam, Spam, Spam and
Spam',
    'Lobster Thermidor aux crevettes with a Mornay sauce, garnished with
```

```
    truffle pâté, brandy and a fried egg on top and Spam'
        ]
```

5.1 str.startswidth()

Returns `True` if string commences with the specified value.

```
is_first = menu.startswith('Spam')
```

💡 you can call `str.endswith()` to check if a string *ends* with the specified value.

5.2 str.lower()

Switch menu text to lower case.

```
lower_case = menu.lower()
```

💡 you can call `str.upper()` to switch text to upper case.

5.3 str.count()

Return the number of times a specified value occurs in a string.

```
spam_count = menu.count('Spam')
```

5.4 str.replace()

Returns a new string by replacing the specified substring with a new value.

```
gummies_menu = menu.replace('Spam', 'Gummies') # replace Spam with Gummies
```

5.5 str.strip()

Returns a "trimmed" version of the string, removing leading and trailing spaces as well as newline escape sequences (`\n`).

```
monty_python = " Monty Python's Flying Circus \n"
monty_python = monty_python.strip()
```

💡 To remove spaces from either the beginning or the end of the string only use `str.lstrip()` or `str.rstrip()`.

5.6 str.join()

Returns a new string by joining each element in the passed in *iterable* to the specified string.

```
items = ['Oatmeal', 'Fruit', 'Spam'] # a list
menu_item = ' '.join(items) # build a string by joining each element to an
empty string

menu_item = ', '.join(items) # build a string by joining each element to a
comma
```

5.7 str.find()

Finds the *first* occurence of the specified value and returns its index value. If the value is not located -1 is returned.

```
menu_item = 'Spam, Spam, Spam, egg and Spam'
position = menu_item.find('Spam')

menu_item = 'Spam, Spam, Spam, egg and Spam'
position = menu_item.find('ham')
```

💡 `str.rfind()` attempts to locate the *last* occurence of the specified value. If the value is not located -1 is returned.

5.8 str.index()

Finds the *first* occurence of the specified value and returns its index value. If the value is not located a `IndexError` is raised.

```
menu_item = 'Spam, Spam, Spam, egg and Spam'
position = menu_item.index('egg and Spam')

# TODO UNCOMMENT and raise runtime exception
# position = menu_item.index('ham') # IndexError
```

5.9 str.split()

Split a string into a new list per the provided *delimiter* (e.g., ",", " ", "|"). Default behavior is to split on a space.

```
menu_item = 'Spam, bacon, sausage, Spam'
dish_items = menu_item.split(', ') # returns list
```

5.10 str.splitlines()

Split a multiline string at each line break and return a list of individual lines.

```
menu_items = menu.splitlines() # returns list
```

Challenge 02

The Python's cafe in Bromley is under new management. Implement the following changes to the multiline string named `menu`. Use method chaining to accomplish the task. Assign the modified menu to the variable `menu_v2`.

❗ The requirements are unordered and may not represent the correct order of operations to perform.

💡 Break the problem into subproblems solving each problem in turn by calling a `str` method. Uncomment the `print()` function and run your file after every change, printing `menu_v2` to the terminal screen in order to check your work.

- Replace every instance of the substring `sausage` with the string `toast`.
- Substitute every third (3rd) instance of `Spam` listed in a menu item with the value assigned to the variable `healthy_choice`.

  Example substring substitution:

  `'Spam, Spam, Spam, egg and Spam' -> 'Spam, Spam, Oatmeal, egg and Spam'`
- Convert the menu to lower case.
- Split the multiline string into a list with each new list element corresponding to a line in `menu`.

```
menu = """Egg and bacon
Egg, sausage and bacon
Egg and Spam
Egg, bacon and Spam
Egg, bacon, sausage and Spam
Spam, bacon, sausage and Spam
Spam, egg, Spam, Spam, bacon and Spam
Spam, Spam, Spam, egg and Spam
Spam, Spam, Spam, Spam, Spam, Spam, baked beans, Spam, Spam, Spam and Spam
Lobster Thermidor aux crevettes with a Mornay sauce, garnished with
truffle pâté, brandy and a fried egg on top and Spam"""

healthy_choice = 'Oatmeal'
```

```
menu_v2 = None # TODO Implement
```

Bonus: Create a new menu subject to the requirements specified above but also add the menu item `Cereal, Toast, and Blueberries` as the first line in the multiline string employing a `str` method to accomplish the task. Do this **prior** to splitting the multiline string. Assign the new list to the variable `bonus_menu_v2`.

```
bonus_menu_v2 = None # TODO Implement (if time permits)
```

6.0 List methods

As with strings, List objects (type: `list`) also include "built-in" behaviors that are defined as *methods*. Calling a `list` method may require that one or more arguments (values) be passed to it in order to perform the requested computation. Depending on the method definition, the operation may result in a computed value being returned to the caller; otherwise `None` is returned.

Below is a select list of `list` methods. For more complete list of available `list` methods see the w3schools Python List/Array Methods page.

6.1 list.append()

Appends element to the end of a list. The operation mutates the list *in-place*, returning `None` implicitly to the caller.

```
menu_v2.append('red beans and rice') # modify in-place (no variable
assignment)
```

❗ Do not assign the return value of `list.append(< element >)` to the current list variable. Doing so will assign `None` to the variable.

6.2 list.remove()

Remove element from list. The operation mutates the list *in-place*, returning `None` implicitly to the caller.

```
item = menu_v2[-2] # Lobster Thermidor
menu_v2.remove(item)
```

6.3 list.extend()

Extend list with another list. The operation mutates the list *in-place*, returning `None` implicitly to the caller.

```
healthy_items = ['cereal, yogurt, and spam', 'oatmeal, fruit plate, and
spam']
menu_v2.extend(healthy_items)
```

6.4 list.sort()

Sort the list. The operation mutates the list *in-place*, returning `None` implicitly to the caller.

💡 You can pass the optional arguments `reverse=True|False` (pipe = 'or') as well as `key=some_function` in order to further specify the sorting criteria (out of scope for the moment).

```
menu_v2.sort() # default alpha sort
```

6.5 list.index()

Return index position by value.

```
index = menu_v2.index('egg, bacon, and spam')
```

6.6 list.insert()

Insert element at specified index position. The operation mutates the list *in-place*, returning `None` implicitly to the caller.

```
menu_v2.insert(1, 'belgian waffle, strawberries, and spam')
```

6.7 list.pop()

Returns the element at the specified position *after* removing it from the list.

```
retired_item = menu_v2.pop(-1) # pop the last item out of the list
```

💡 `str.rindex()` attempts to locate the *last* occurence of the specified value. If the value is not located a `IndexError` is raised.

6.8 list.copy()

❗ Assigning an object to a variable **is not** a copy operation. Variable assignment involves binding a name or pointer to an object; nothing beyond labeling the object should be assumed.

Use `list.copy()` to return a *shallow copy* of a list. The method call returns a new list containing *references* to the elements found in the original. *Deep copies* that return both a new list and *copies* of the elements found in the original can also be created using the copy module.

```
# Don't do this
new_menu = menu_v2 # not a copy
popped = new_menu.pop(-1) # mutates both new_menu and menu_v2

print(f"\n4.8 popped item = {popped}")
print(f"\n4.8 menu_v2 (len={len(menu_v2)}) = {menu_v2}") # lost an item

# Do this
new_menu = menu_v2.copy() # shallow copy
popped = new_menu.pop(-1) # mutates new_menu only
```

Challenge 03

Let's continue tweeking the menu. First make a copy of `menu_v2` and assign it to `menu_v3`. Then implement another set of requested changes to `menu_v3`. Use indexing, slicing, and `list` and `str` method calls to accomplish the task. For this challenge implement the requirements in the order specified.

💡 Break the problem into subproblems solving each problem in turn by calling a `str` method. Uncomment the `print()` function and run your file after every change, printing `menu_v2` to the terminal screen in order to check your work.

1. Create a "shallow copy" of `menu_v2` and assign it the variable `menu_v3`.
2. Reverse the order of `menu_v3` and assign the return value to `menu_v3`.
3. Return the length of `menu_v3` and assign it to the variable `menu_v3_len`. Use the length value in an arithmetic expression to access the last element in `menu_v3`. Assign the menu item accessed to the variable `menu_item`. Then call the appropriate `list` method to remove the `menu_item` from the list `menu_v3`.

   💡 recall that Python indexes are zero-based.
4. Slim down `menu_v3` by returning every other menu item. Assign the return value to `menu_v3`.

7.0 String formatting

There are three ways to format one or more values as a string. We recommend that you utilize the newest approach: the *formatted string literal* (f-string). That said, you will encounter the other string formatting routines when reading older code or tutorials so its important to understand how to implement the other approaches.

7.1 Formatted string literal (f-string)

The f-string syntax `f"some_string {some variable}"` is less verbose and easier to construct than earlier string formatting approaches. Employ curly braces to denote embedded variables in the expression.

```
special_item = 'egg, bacon, spam and sausage'
question = f"Why can't she have {special_item}?" # embedded variable
```

💡 Recall that `\n` represents an escape sequence, specifically an ASCII linefeed (LF). Think of `\n` as "new line". Passing `\n` in a string will insert a new line at the position of the escape sequence.

7.2 str.format()

Formats the specified value(s) and inserts them inside the string's using curly braces `{}` as a placeholder.

```
question = "Could I have {}, {}, {} and {}, without the
spam?".format('egg', 'bacon', 'spam', 'sausage')
```

💡 The placeholders can be identified using empty placeholders `{}`, numbered indexes `{0}`, or named indexes `{egg}`.

7.3 C-style or simple positional formatting

The oldest of the three string formatting approaches. Uses the `%` character as a placeholder.

Placeholders (select list):

- `%c` = single character placeholder
- `%d` = decimal placeholder
- `%i` = integer placeholder
- `%s` = string placeholder

```
question = "No, it wouldn't be %s, %s, %s and %s, would it?" % (egg,
bacon, spam, sausage)
```

For a summary of ye olde C-style / simple positional formatting see Frank Hofman, "Python String Interpolation with the Percent (%) Operator" (Stack Abuse, nd).