SI 506 Lecture 05

Topics

1. Sequences: strings, lists, and tuples
2. Creating a list from a string
3. Indexing
4. Slicing
5. In-class coding challenges

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.
- **Subscript notation**. Square brackets [`[]`] enclosing either an index value or a slicing expression that is used to access sequence characters, elements or items.
- **Tuple**. An ordered sequence that cannot be modified once it is created.

Reference

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

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

This week's 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 various Monty Python comedy sketches including 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, lists, and tuples

This week we discuss Python sequences, focusing on three sequence types: strings, lists, and tuples.

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 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
```

As you saw last week 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)
```

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, elements), 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 joined to the `pythons` list. This results in a new list which is assigned to the existing variable `pythons`.

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

1.3 Tuple basics

A Python tuple (type: `tuple`) is an *ordered* sequence of items. Once created, the tuple is considered *immutable* and cannot be modified. The tuple is also an *iterable*, a type of object whose members (in this case, items), can be accessed. Tuples provide an optimized data structure for referencing groups of values that form "natural" associations that do not change over the life of a program or script (e.g., 'Ann Arbor', 'Michigan', 'USA').

Tuples are typically defined by enclosing the items in parentheses `()` instead of square brackets `[]` as is the case with lists.

```
# A tuple
silly_walks = ('Monty Python', 'Sketch', 'The Ministry of Silly Walks',
'15 September 1970')
```

A single item tuple **must** include a trailing comma (`,`) or the Python interpreter will consider the expression a string.

```
python_theme_song = ('The Liberty Bell',) # Note trailing comma
```

💡 You can also create a *new* list by *concatenating* two or more tuples using the plus (`+`) operator. In the example below the `python_theme_song` tuple is joined with two other tuples to form a new tuple that names the unwitting composer of the Python's *Flying Circus* theme song along with its original publication date.

```
python_theme_song = ('John Philip Sousa', 'Composer') + python_theme_song
+ (1893,) # Concatenation
```

❗ Unlike a string a tuple can reference *mutable* items like lists and dictionaries that are capable of modification despite their inclusion in the tuple as is illustrated below:

```
holy_grail = (
    'Monty Python and the Holy Grail',
    1975,
    [
        'Arthur, King of the Britons',
        'Sir Lancelot the Brave',
        'Sir Bedevere the Wise',
        'Sir Galahad the Pure'
        ]
    )
# holy_grail[1] = '3 April 1975' # Illegal
holy_grail[2].append('Patsy') # Mutates tuple list item with a new element
```

In a later lecture we will discuss how to compare two or more tuples using comparison operators ('=', '<', '>') in a conditional statement.

2.0 Creating a list from a string

String objects are provisioned with two methods that return a version of the string converted to a list: `str.split()` and `str.splitlines()`.

2.1 `str.split()`

The `str.split()` method will return a list of string elements split on a separator or delineator. Default behavior is to split the string whenever a space is encountered.

The `str.split()` method defines two parameters:

- sep: the separator (or *delineator*) used to split the string (default = `' '`)
- max_splits: the maximum number of splits from left to right (default = `-1` (i.e., no limit))

```
sketch_comedy = "Monty Python's Flying Circus"
words = sketch_comedy.split() # Returns ['Monty', "Python's", 'Flying',
'Circus']
```

Passing a separator value to `str.split()` will return a list of string elements split at the specified separator:

```
sketches = 'Dead Parrot Sketch, The Spanish Inquisition, The Argument
Clinic'
sketches = sketches.split(', ') # Returns ['Dead Parrot Sketch', 'The
Spanish Inquisition', 'The Argument Clinic']
```

2.2 `str.splitlines()`

The `str.splitlines()` method will return a list of string elements split on a line boundary or break.

The `str.splitlines()` method defines a single parameter:

- keepends: each string element retains the trailing line break (`\n`) if `True` is specified

```
excerpt = """Nobody expects the Spanish Inquisition.
Our chief weapon is surprise.
Surprise and fear. Fear and surprise.
Our two weapons are fear and surprise ...
and ruthless efficieny.
Our three weapons are fear and surprise and ruthless efficiency ...
and an almost fanatical devotion to the pope.
Uh! Four. No.
Amongst our weapons ....
Amongst our weaponry are such elements as fear, su -- I'll come in again.
"""

lines = excerpt.splitlines() # Returns list of string elements split on
each line break
```

2.3 Challenge 01

Task: Combine two strings and then split the string and return a list of sentence elements.

💡 A string can span multiple lines by employing a trailing backslash (`\`) after each line or by surrounding the multiple lines by a parentheses (`()`).

Below is a short excerpt from a longer exchange between Arthur, King of the Britons, and a peasant named Dennis that occurs early on in the 1975 film `Monty Python and the Holy Grail`.

```
arthur = 'The Lady of the Lake, '\
    'her arm clad in the purest shimmering samite, '\
    'held aloft Excalibur from the bosom of the water '\
    'signifying by Divine Providence that '\
    'I, Arthur, was to carry Excalibur. '\
    'That is why I am your king.'

dennis = (
    'Listen, strange women lying in ponds distributing swords '
    'is no basis for a system of government. '
    'Supreme executive power derives from a mandate from the masses, '
    'not from some farcical aquatic ceremony.'
    )
```

1. Combine the `arthur` and `dennis` strings. Assign the new string to a variable named `excalibur`.

   ❗ You *must* maintain proper spacing when joining the strings (i.e., one space between sentences).

2. Split the string into a list of sentence elements and assign the new list to a variable named `excalibur`.

3. Uncomment `print()` and check your work.

3.0 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 the subscript operator (`[]`) and an index value.

💡 *Subscript notation* employs two brackets `[< index | slice >]` enclosing either a positive or negative index value (eg., `some_sequence[0]`) or a slicing expression `some_sequence[:5]`.

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

3.1 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").

3.2 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]
```

3.3 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
```

3.4 Challenge 02

Task: Return the first `menu` item that includes more than one helping of Spam.

1. Employ indexing to return the first `menu` item that includes multiple helpings of Spam. Assign the slice to a variable named `order`.

2. Uncomment `print()` and check your work.

4.0 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'
]
```

4.1 Slicing start/end range

💡 In the slicing example below the start value `1` is considered *inclusive* while the end value `3` is considered *exclusive* (i.e., the element is excluded from the slice).

```
# 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 Mr and Mrs Bun:

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

If you were asked to return a `cast` slice containing *only* the named cast members the following slicing expression would get the job done:

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

On the other hand, if you were asked to return a `cast` slice containing *only* the unnamed cast members (the extras) consider employing negative slicing:

```
# 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]
```

4.2 Challenge 03

Task: Retrieve a subset of the `cast` list using positive slicing.

1. Employ slicing to access the elements in `cast` that represent John Cleese and Michael Palin using *postive* index values. Assign the return value (a new list) to the variable `cleese_palin`.

2. Uncomment `print()` and check your work.

4.3 Challenge 04

Task: Retrieve a subset of the `cast` list using negative slicing.

1. Employ slicing to access the Viking elements in `cast` using *negative* index values. Assign the return value (a new list) to the variable `vikings`.

2. Uncomment `print()` and check your work.