Sequences

Learning Objectives

- Identify and use different kinds of basic sequences: strings, ranges, lists and tuples
- Understand the limitations and restrictions of each type of sequence
- Understand the difference between mutable and immutable sequences
- Use an index to access a value in a sequence
- Use slicing to obtain a subsequence

Strings as Sequences

Why is a String a Sequence?

Create a string by writing out a literal as a bunch of characters placed between a pair of the quotation marks of your choice:

```
vocabulary_word = "vermiculate"
```

Sequences are *collections of data*.

Why is a String a Sequence?

A string is defined not just by the characters it contains,
but by the order in which those characters are stored.

```python
a = "relatives"
b = "versatile"
print(a == b)   # prints False!
```

Sequences are *ordered collections of data.*

Indexing in Sequences

Sequences in Python are **indexable:** we can refer to values at specific positions in the sequence by their positions.

- first value lives at index `0`
- second value lives at index `1`

```
"indexing"
 01234567
```

Notice that "indexing" is a string with eight characters: since we start counting at `0`, the index of the last character is `7`.

Indexing in Sequences

For a sequence of length `n`, the valid indices always range from `0` to `n-1`.

- Negative indices & indices >= `n` lead to `IndexError`

```
"short"     # 5 characters long
 01234      # biggest index: 4

"lengthy"   # 7 characters long
 0123456    # biggest index: 6
```

Indexing in Strings

For any sequence `s`, the operation to get the value at index `i` is `s[i]`.

```
full_name = "Travis Q. McGaha"
middle_initial = full_name[7]   # "Q"
first_initial = full_name[0]    # "T"
last_initial = full_name[10]    # "M"
```

In a `str`, the values at each index are individual characters—actually `str` values themselves

Indexing in Strings

When `i` is too big, we get `IndexError` and the program will crash.

```
>>> "HSS"[100]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
```

Sequences and Concatenation

Since each initial is just a `str`, we can concatenate them all together using the `+` operator.

- Many (not all) sequences support concatenation.

```
full_name = "Travis Q. McGaha"
middle_initial = full_name[7]   # "Q"
first_initial = full_name[0]    # "T"
last_initial = full_name[10]    # "M"

full_initials = first_initial + middle_initial + last_initial
print(full_initials)            # prints "TQM"
```

Slicing Sequences

Slicing: Generating Subsequences

We know how to refer to one position in a sequence at a time with a single index.

- How about a group of positions—a **subsequence**?
- If we want to obtain a subsequence of a string `s` including all characters starting at index `i` and stopping *before* index `j`, then we can do that by writing `s[i:j]`

```
print("earth"[1:4])      # prints "art" 🖌️
print("earth"[0:3])      # prints "ear" 👂
```

This operation is called **slicing.**

Slicing: Starting and Stopping

When slicing, we always *excluding* the character at the end position:

- `"earth[1:4]"` gives `"art"`, which is the subsequence consisting of characters at positions `1`, `2`, and `3` only.
- For a string `s`, `s[i:j]` will always have a length of `j - i` characters.
- To include the last character in a string of length `n`, use a stop index of `n`

Slicing: Shortcuts

```python
title = "crossroads"
# all three examples below give exactly the same value
roads_one = title[5:10]
roads_two = title[5:len(title)]
roads_three = title[5:]

print(roads_one)                                # prints "roads"
print(roads_one == roads_two == roads_three)    # prints True
```

This last version—`title[5:]`—is a useful syntactical shorthand for getting all characters in `title` at & after index `5`.

Slicing: Shortcuts

```
title = "crossroads"
# both examples below give exactly the same value
cross_one = title[0:5]
cross_two = title[:5]

print(cross_one)                   # prints "cross"
print(cross_one == cross_two)      # prints True
```

Can similarly omit the first number to take everything from the beginning.

Slicing and Stepping

If you only want every `k`th element of a string `s` starting at index `i` and ending at index `j`, you can write

```
s[i:j:k]
```

```
>>> "AaBbCc"[2:5:2]
'BC'
```

- Start at index `2` ("B"), take that character.
- Take `2` steps forward to index `4`.
- Since index `4` is before stop index `5`, take it. ("C")
- Take `2` steps forward to index `6`.
- Since index `6` is not before stop index `5`, stop.

Slicing and Stepping

```
>>> "AaBbCc"[0:6:3]
'Ab'
```

- Start at index `0` ("A"), take that character.
- Take `3` steps forward to index `3`.
- Since index `3` is before stop index `6`, take it. ("b")
- Take `3` steps forward to index `6`.
- Since index `6` is not before stop index `5`, stop.

Slicing and Stepping

Stepping can go backwards. The start index will be larger than the stop index. 🙃

```
>>> "devolve"[4:0:-1]
'love'
```

- Start at index `4` ("l"), take that character.
- Take `1` steps backward to index `2`.
- Since index `3` is after stop index `0`, take it. ("o")
- Take `1` steps backward to index `1`.
- Since index `2` is after stop index `0`, take it. ("v")
- Take `1` steps backward to index `0`.
- Since index `1` is after stop index `0`, take it. ("e")
- Take `0` steps backward to index `0`. Stop.

Reversing

Omit the start and stop values to get a "slice" of the entire string but in reverse.

```
>>> "stop"[::-1]
'pots'
```

A little confusing to parse *why* that works, but a handy tool to keep in mind.

Membership

Slicing allows us to pull a subsequence out of another sequence.

- **For strings only**, we can check to see if a subsequence is found anywhere in a larger string
- Use the `in` keyword to ask if a subsequence `s` is present in a larger string `t`: `s in t`

```
>>> "art" in "earth"
True
>>> "at" in "earth"
False
>>> "e" in "earth"
True
>>> "q" in "earth"
True
>>> "earth" in "earth"
True
```

Ranges

Ranges

A **range** is a sequence of numbers defined by a start point, stop point, and step size.

- Like a string is a sequence of characters, a range is a sequence of numbers
- Created by writing `range(start, stop, step)`
    - Both `start` and `step` can be omitted for convenience to get a range from `0` to `stop`.

Creating Ranges

Contents	Expression
0, 1, 2, 3, 4	`range(5)`
1, 2, 3, 4, 5	`range(1, 6)`
1, 3, 5	`range(1, 6, 2)`
0, 10, 20, 30, 40, 50	`range(0, 51, 10)`
empty!	`range(6, 0)`
6, 5, 4, 3, 2, 1	`range(6, 0, -1)`

Ranges: Support Indexing & Slicing

```python
big_range = range(0, 100)
smaller_range = big_range[0:11]
print(smaller_range)                        # prints range(0, 11)
print(big_range[10])                        # prints 10
```

Ranges: Membership

Using `in` for ranges can only check to see if **individual numbers** are present inside of a larger range.

```
big_range = range(0, 100)
smaller_range = big_range[0:11]
print(smaller_range in big_range)     # prints False
print(10 in big_range)                # prints True
```

Ranges: Limitations

You cannot:

- concatenate two ranges
- nicely inspect all the contents of a range by printing

```
>>> range(1, 3) + range(10, 100)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'range' and 'range'
>>> print(range(1, 3))
range(1, 3)
```

Tuples & Lists

Tuples

A **tuple** is an **immutable** sequence of values

- Potentially all of different types
- Denoted using parentheses
- Indexable, sliceable, supports membership checking
- Cannot add or change things without creating a new tuple.

Tuples: The Basics

My initials, as a tuple of three strings:

```
>>> initials = ("H", "S", "S")
>>> len(initials)
3
>>> initials[0]
'H'
>>> initials[0:2]
('H', 'S')
>>> "H" in initials
True
>>> ("H", "S") in initials
False
```

Tuples With Mutliple Types

Tuples can contain values of different types

```
>>> some_data = ("H", 27, False)
>>> len(some_data)
3
>>> some_data[0]
'H'
>>> some_data[0:2]
('H', 27)
>>> 27 in some_data
True
>>> ("H", 27) in some_data
False
```

Concatenating Tuples

```python
letters = ("a", "b", "c")
numbers = (1, 2, 3)

everything = letters + numbers
print(everything)
```

Prints:

```
("a", "b", "c", 1, 2, 3)
```

This leaves `letters` and `numbers` unchanged—a new tuple is created!

List

A **list** is a **mutable** sequence of values

- Potentially all of different types
- Denoted using square brackets (`[]`)
- Indexable, sliceable, supports membership checking
- Can add, remove, and change things in the list!

List: The Basics

My initials, as a list of three strings:

```
>>> initials = ["H", "S", "S"]
>>> len(initials)
3
>>> initials[0]
'H'
>>> initials[0:2]
('H', 'S')
>>> "H" in initials
True
>>> ["H", "S"] in initials
False
```

Lists With Mutliple Types

Tuples can contain values of different types

```
>>> some_data = ["H", 27, False]
>>> len(some_data)
3
>>> some_data[0]
'H'
>>> some_data[0:2]
['H', 27]
>>> 27 in some_data
True
>>> ["H", 27] in some_data
False
```

Concatenating Lists

```
letters = ["a", "b", "c"]
numbers = [1, 2, 3]

everything = letters + numbers
print(everything)
```

Prints:

```
["a", "b", "c", 1, 2, 3]
```

This leaves `letters` and `numbers` unchanged—a new list is created!

Changing Lists

```
numbers_list = [1, 2, 3]

numbers_list[2] = -3
print(numbers_list)
```

Prints:

```
[1, 2, -3]
```

Changing Tuples: No Can Do!

```
numbers_tuple = (1, 2, 3)

numbers_tuple[2] = -3           # this line leads to a TypeError!
print(numbers_tuple)
```

Results in:

```
TypeError: 'tuple' object does not support item assignment
```

Growing Lists: `append`

`append()` allows us to add a single value to the end of a list.

```
numbers_list = [1, 2, 3]

numbers_list.append(4)
print(numbers_list)
```

Prints:

```
[1, 2, 3, 4]
```

Growing Lists: `extend`

`extend()` allows us to add all contents of another list onto this list.

```
numbers_list = [1, 2, 3]
letters_list = ["a", "b", "c"]

numbers_list.extend(letters_list)
print(numbers_list)
```

Prints:

```
[1, 2, 3, "a", "b", "c"]
```

No new list is created!

Concatenating Lists: `+`

`+` allows us to create a new list combining the contents of one list before the contents of another list

```
numbers_list = [1, 2, 3]
letters_list = ["a", "b", "c"]

new_list = numbers_list + letters_list
print(numbers_list)
print(new_list)
```

Prints:

```
[1, 2, 3]
[1, 2, 3, "a", "b", "c"]
```

`numbers_list` is unchanged!

Immutability

Tuples are suitable for fixed-length, permanent collections.

- `append`, `extend`, and setting the value at a particular index (e.g. `t[3] = "new"`) do not work!

Lists are suitable for variable-length, changing collections.

Summary

Sequences: Ordered Collections

No matter what, all sequence types are ordered collections of elements.

- Ordering gives rise to indexing, which allows for selecting individual elements or subsequences

Different sequence types have different restrictions on what they contain.

- `str`: characters
- `range`: `int` values
- `tuple`: anything
- `list`: anything

Sequences

Type	Index/Subsequence	Membership	`len()`	Concatenation	Modification
`str`	yes	individual elements or subsequences	yes	yes	no
`range`	yes	individual elements	yes	no	no
`tuple`	yes	individual elements	yes	yes	no
`str`	yes	individual elements	yes	yes	yes