Information, Computation, Communication
Learning Python

Tuples and Sets

Agenda

Tuples

Tuples vs. Lists

- Tuples and lists are similar
  - Both are **sequences** of data
  - Both store a collection of items, where **each item can be of any type**
  - In both, one can access any item by its **index**

- What is the difference then?
  - **Lists are mutable** (can be changed)
  - **Tuples are immutable** (cannot be changed)

Tuples vs Lists

- **Unlike lists**: Tuples are delimited by parentheses
- **Like lists**: Commas separate tuple elements

```
fruits_lst = ['raspberry', 'mango', 'kiwi'] # ['raspberry', 'mango', 'kiwi']

fruits_tpl = ('raspberry', 'mango', 'kiwi') # ('raspberry', 'mango', 'kiwi')
```

Tuples are Immutable Cannot be Modified

```
fruits_lst = ['raspberry', 'mango', 'kiwi']  # ['raspberry', 'mango', 'kiwi']

fruits_tpl = ('raspberry', 'mango', 'kiwi')  # ('raspberry', 'mango', 'kiwi')


fruits_lst.append('apple')  # ['raspberry', 'mango', 'kiwi', 'apple']

fruits_tpl.append('apple')  # won't work because a tuple cannot be changed!
```

```
File ".\tuples.py", line 13, in <module>
fruits_tpl.append('apple')
AttributeError: 'tuple' object has no attribute 'append'
```

Tuples are Immutable

```
fruits_lst = ['raspberry', 'mango', 'kiwi'] # ['raspberry', 'mango', 'kiwi']

fruits_tpl = ('raspberry', 'mango', 'kiwi') # ('raspberry', 'mango', 'kiwi')


fruits_lst[0] = 'orange'  # ['orange', 'mango', 'kiwi', 'apple']

fruits_tpl[0] = 'orange'  # won't work either!
```

```
File ".\tuples.py", line 20, in <module>
    fruits_tpl[0] = 'orange'
TypeError: 'tuple' object does not support item assignment
```

How to Work with Tuples?

- Functions and methods that do **not** attempt to **modify** a tuple will work equally well for tuples and lists

```
fruits_tpl = ('raspberry', 'mango', 'kiwi')
len(fruits_tpl)     # number of elements, 3
fruits_tpl[1]       # indexing, 'mango'
fruits_tpl[:2]      # slicing, ('raspberry', 'mango')
'mango' in fruits_tpl             # True
'watermellon' in fruits_tpl       # False
```

Tuple Packing and Unpacking

- In Python, there is a very powerful tuple assignment feature that assigns the right-hand side values to the left-hand side
- Packing: From values **to a tuple**
- Unpacking: **From a tuple** to a variable

Packing: Comma-separated right-side values are converted to a tuple.

```
values = 123, 'crayon', -9.5
print(values) # (123, 'crayon', -9.5)
type(values)  # <class 'tuple'>
```

Tuple Packing and Unpacking

Unpacking: Comma-separated left-side values are unpacked automatically from a tuple.

```
a, b, c = values # a=123, b='crayon', c=-9.5
type(a) # <class 'int'>
type(b) # <class 'str'>
type(c) # <class 'float'>
```

Ignoring Values When Unpacking

- Sometimes, we do not care for **all** values in a tuple we are unpacking

```
# To ignore a value, use an underscore _
d, _, f = values
print(d, f)
# 123 -9.5
```

Packing&Unpacking Example: Swap Values

Using temporary variable

```
x = 1
y = 99

temp = x      # temp = 1
x = y         # x = 99
y = temp      # y = 1
print(x, y)   # 99 1
```

Packing&Unpacking Example: Swap Values

Using tuple packing and unpacking

```
x = 1
y = 99
```

Unpacking to x, y

Packing into a tuple (99, 1)

```
x, y = y, x

print(x, y) # 99 1
```

Reminder: **Built-in Function enumerate()**

- Allows to iterate over an object <u>and</u> to keep count of iterations

- Takes two arguments
  - A sequence or an object that supports iteration
  - Start (optional, default zero): iterates starting from this number
- Returns:
  - Enumerate object, which you can convert to a list or tuple using list( ) and tuple( ) methods
- Typical use case
  - Obtain an index of an element of a list or a tuple besides its value

Tuples and Enumerate Neat Way of Traversing Sequences

- To extract the index of an element of a list (or tuple) besides its value, the most Pythonic way is to use enumerate( )

```python
fruits_lst = ['raspberry', 'mango', 'kiwi']
enumerate_fruits = enumerate(fruits_lst)
# Returns an object of enumerate type

print(enumerate_fruits) # Unusable value printed
# Convert the enumerate object to a list to print it
enumerate_fruits_l = list(enumerate_fruits)
# [(0, 'raspberry'), (1, 'mango'), (2, 'kiwi')]
# Note that every element of this list is a tuple
```

Tuples and Enumerate Neat Way of Traversing Sequences

- To extract the index of an element of a list (or tuple) besides its value, the most Pythonic way is to use enumerate( )
- Note the use of tuples and **unpacking** in the example below

```
fruits_lst = ['raspberry', 'mango', 'kiwi']

for index, fruit in enumerate(fruits_lst):
    print(index, fruit)
# 0 raspberry
# 1 mango
# 2 kiwi
```

Sets

Sets

- **Unordered** collections of **distinct** elements
- Sets are delimited by curly braces

```
my_set = {1, 2, 3, 'a', 'b', 'c'}
type(my_set)  # <class 'set'>
```

- When to use sets?
  - When having duplicates is not an option
  - When performing set operations is the aim

Example: Removal of Duplicates from a List

- Write a function that takes a list of characters and returns another list containing all the original list's **unique** elements.
  Sort the returned list in the alphabetic order.

```
def remove_duplicates(input_list):
    return list(set(input_list))


l = list('johnsnow') # ['j', 'o', 'h', 'n', 's', 'n', 'o', 'w']
l_without_repetitions = remove_duplicates(l)
print(sorted(l_without_repetitions))
# ['h', 'j', 'n', 'o', 's', 'w']
```

Common Set Operations: Creating Sets

```
# Creating an empty set
my_set = set()

# Creating a set from a list
my_set = set([1, 2, 2, 3, 'a', 'a', 'b'])
# {1, 2, 3, 'b', 'a'}
```

Unordered and unique elements.
To order, call the built-in function sorted( ). But, beware, integers and strings cannot be compared (typeError will be raised). Also, note that sorted( ) returns a list.

Common Set Operations

```
# Finding the set size
len(my_set) # 5

# Figuring out if an element is in the set
3 in my_set   # True
'c' in my_set # False

# Adding an element to a set
my_set.add('c') # {1, 2, 3, 'b', 'c', 'a'}
```

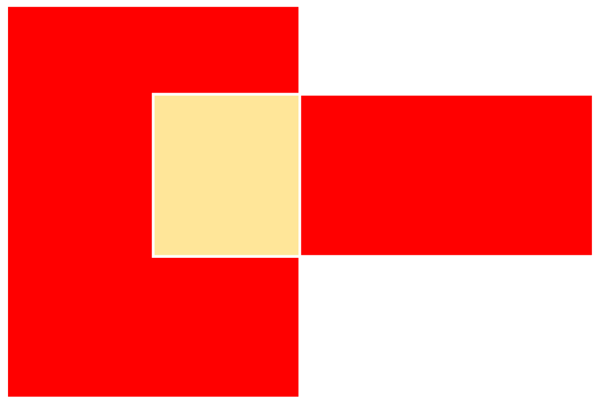
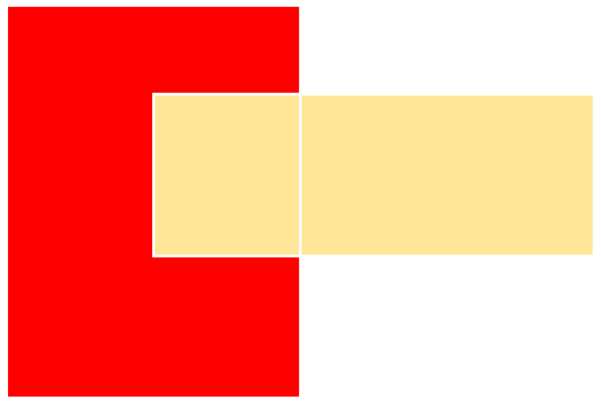
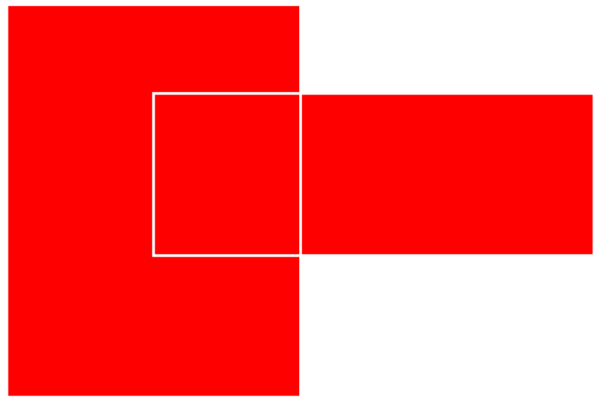
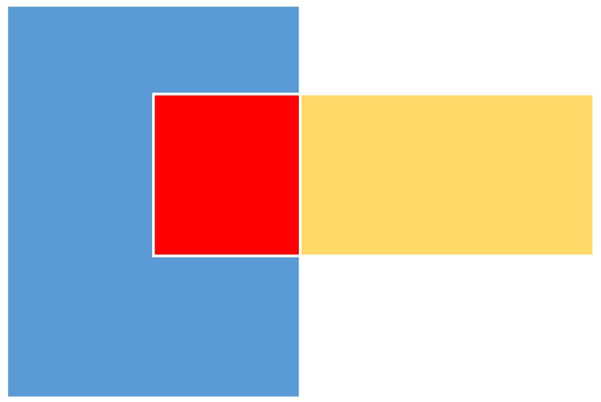
EXAMPLES

Common Set Operations: Removing Elements

```
# Removing a specific element
my_set.discard('c')  # {1, 2, 3, 'b', 'a'}
# discard() does not raise an error if
# the element does not exist


# Removing an arbitrary element
my_set.pop()
# pop() removes and returns an arbitrary element
# from the set. If the set is empty, it raises an
# error KeyError: 'pop from an empty set'
```

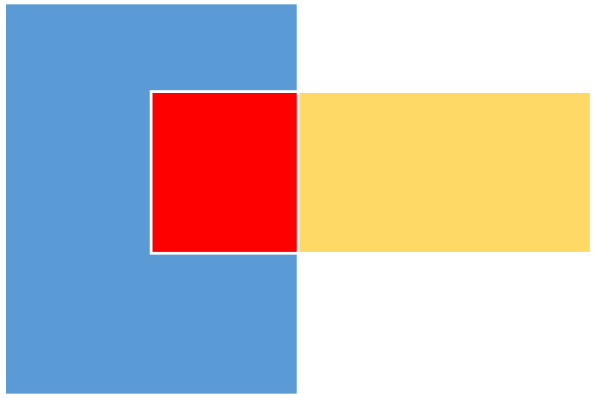
Set Operations

- Four binary operations on sets:
  - Intersection
  - Union
  - Difference
  - Symmetric difference

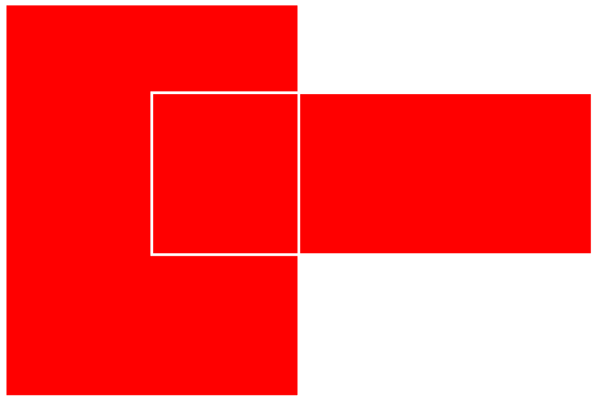
Intersection &: Elements in Both Sets

```
pirate = set('jacksparrow')
# {'p', 'o', 'k', 'r', 'a', 'w', 's', 'j', 'c'}
king_in_the_north = set('johnsnow')
# {'o', 'h', 'w', 's', 'j', 'n'}


# Intersection = elements present in both sets
pirate & king_in_the_north
# {'w','s','j','o'}
# 4 elements
# arbitrary order
```



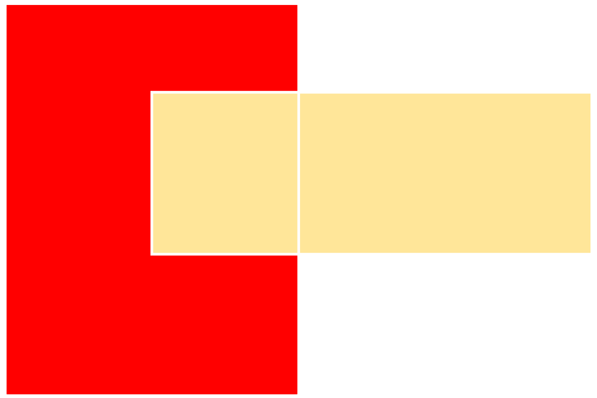
Union |: All Unique Elements in The Two Sets

```
pirate = set('jacksparrow')
# {'p', 'o', 'k', 'r', 'a', 'w', 's', 'j', 'c'}
king_in_the_north = set('johnsnow')
# {'o', 'h', 'w', 's', 'j', 'n'}


# Union, elements present in one set or the other
my_union = pirate | king_in_the_north
# {'r', 'j', 'p', 's', 'w',
# 'n', 'c', 'o', 'k', 'a', 'h'}
# 11 elements, arbitrary order
```



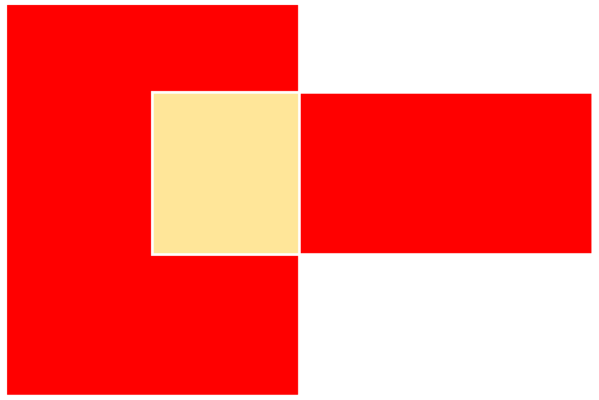
Difference -: Elements in One but Not Other Set

```
pirate = set('jacksparrow')
# {'p', 'o', 'k', 'r', 'a', 'w', 's', 'j', 'c'}
king_in_the_north = set('johnsnow')
# {'o', 'h', 'w', 's', 'j', 'n'}


# Difference = set – intersection
pirate - king_in_the_north
# {'k', 'r', 'p', 'c', 'a'}
# 5 elements, arbitrary order
```



Symmetric difference ^: All But The Intersection

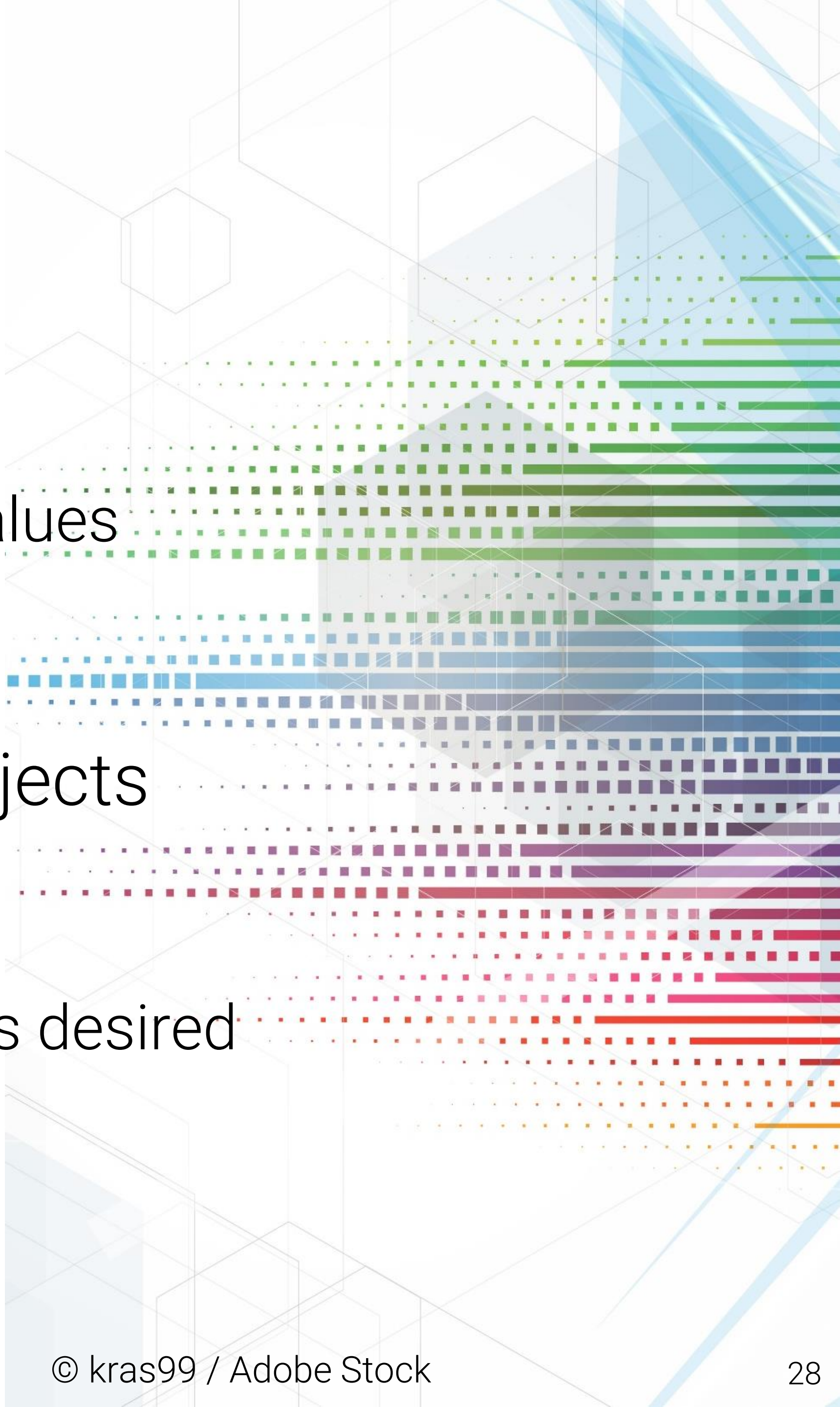
```
# reminder
# union: {'r','j','p','s','w','n','c','o','k','a','h'}
# intersection: {'w','s','j','o'}

# Symmetric difference = union – intersection
pirate ^ king_in_the_north
# {'k', 'r', 'a', 'p', 'c', 'n', 'h'}
# 7 elements, arbitrary order
```



Summary

- Tuples are **immutable** sequences of objects
  - Brackets (parentheses) delimit tuples
  - Tuples are handy for packing and unpacking values
  - enumerate( ) operates on tuples

- Sets are **unordered** collections of **distinct** objects
  - Curly brackets delimit sets
  - When printed, set elements are unordered
  - Useful when intersection or difference of sets is desired
  - Elegant removal of duplicates in a list



Next topic:
Dictionaries