# COSC2429 Introduction to Programming

## Dictionaries

# Outline

- Dictionaries
- Dictionary operations
- Dictionary functions
- in and not in operators
- Aliasing and copying
- Sparse matrices
- Example: Alphabet counting

# Dictionaries

- All of the compound data types we have studied so far — **strings**, **lists**, and **tuples** — are **sequential** (or ordered) collections.
- **Dictionaries** are different. They are **unordered** collections of **key-value pairs** in which each **key**, which can be any immutable type, can be mapped to a **value**, which can be any Python data object.
- As an example, let's **create a dictionary** to translate English words to Spanish.
- Each item can be **added** into the dictionary using the assignment operator.

```
eng2sp = {}                 # create an empty dictionary
eng2sp['one'] = 'uno'       # add the 1st key-value pair into the dictionary
eng2sp['two'] = 'dos'       # add the 2nd key-value pair into the dictionary
eng2sp['three'] = 'tres'    # add the 3rd key-value pair into the dictionary
print(eng2sp)
```

# Dictionaries

- Another way to create a dictionary is to provide a list of key-value pairs using the same syntax as the previous output.

```
eng2sp = {'three': 'tres', 'one': 'uno', 'two': 'dos'}
print(eng2sp)
```

- Python uses complex algorithms, designed for very fast access, to determine where the key-value pairs are stored in a dictionary thus we don’t know their orders.
- It doesn’t matter what order we write the pairs. The values in a dictionary are accessed with **keys**, not with indices, so there is no need to care about ordering.

```
eng2sp = {'three': 'tres', 'one': 'uno', 'two': 'dos'}
value = eng2sp['one']
print(value)
```

# Dictionary operations

- The **del** statement removes a key-value pair from a dictionary.

```
inventory = {'apples': 430, 'bananas': 312, 'oranges': 525, 'pears': 217}
del inventory['pears']
```

- We can change the values but not the keys in a dictionary.
- The **len** function also work on dictionaries, returning the number of key-value pairs in a dictionary.

```
inventory = {'apples': 430, 'bananas': 312, 'oranges': 525, 'pears': 217}
inventory['pears'] = 0
print(len(inventory))
```

# Dictionary methods

- Dictionaries have a number of useful built-in methods as described in the following table. More details can be found in the Python Documentation.

| Method | Parameters | Description |
|---|---|---|
| keys | none | Returns a new view of dictionary's keys |
| values | none | Returns a new view of dictionary's values |
| items | none | Returns a new view of dictionary's (key, value) pairs |
| get | key | Returns the value associated with key; None otherwise |
| get | key, alt | Returns the value associated with key; alt otherwise |

```
inventory = {'apples': 430, 'bananas': 312, 'oranges': 525, 'pears': 217}

for key in inventory.keys():     # the order in which we get the keys is not defined
   print("Got key", key, "which maps to value", inventory.get(key))

print(list(inventory.keys()))          # list() convert the view of keys into a list of keys
print(list(inventory.values()))
print(list(inventory.items()))
```

# Dictionary functions

- It is so common to iterate over the keys in a dictionary that you can omit the **keys** function call in the for loop.

```
inventory = {'apples': 430, 'bananas': 312, 'oranges': 525, 'pears': 217}

for k in inventory:
   print("Got key", k)
```

- And here is an example that use the **items** function in the for loop:

```
inventory = {'apples': 430, 'bananas': 312, 'oranges': 525, 'pears': 217}

for (k, v) in inventory.items():
    print("Got", k, "that maps to", v)

for k in inventory:
    print("Got", k, "that maps to", inventory[k])
```

# **in** and **not in** operators

- The **in** and **not in** operators can test if **a key is in the dictionary**:

```
inventory = {'apples': 430, 'bananas': 312, 'oranges': 525, 'pears': 217}
print('apples' in inventory)
print('cherries' in inventory)

if 'bananas' in inventory:
    print(inventory['bananas'])
else:
    print("We have no bananas")
```

- The **in** operator can be very useful since looking up a non-existent key in a dictionary causes a runtime error.

# Aliasing and copying

- Because dictionaries are mutable, you need to be aware of aliasing (as we saw with lists). Whenever two variables refer to the same dictionary object, changes to one affect the other.

```
opposites = {'up': 'down', 'right': 'wrong', 'true': 'false'}
alias = opposites

print(alias is opposites)

alias['right'] = 'left'
print(opposites['right'])
```

- If you want to modify a dictionary and keep a copy of the original, use the dictionary copy method. Since *a_copy* is a copy of the dictionary, changes to it will not effect the original.

```
a_copy = opposites.copy()
a_copy['right'] = 'left'    # does not change opposites
```

# Sparse matrices

- A matrix is a two dimensional collection, typically thought of as having rows and columns of data. For example, consider the following matrix:

$$\begin{bmatrix} 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 3 & 0 \end{bmatrix}$$

- The easiest way to create a matrix in Python is to use a list of lists.

```
matrix = [[0, 0, 0, 1, 0],
          [0, 0, 0, 0, 0],
          [0, 2, 0, 0, 0],
          [0, 0, 0, 0, 0],
          [0, 0, 0, 3, 0]]
```

- One thing that you might note about this matrix is that there are many items that are zero. This type of matrix is called a **sparse matrix**.

# Sparse matrices

- Since there is really no need to store all of the zeros, the list of lists representation is considered to be **inefficient**.
- An alternative representation is to use a dictionary to only store the non-zero items. For the keys, we can use tuples that contain the row and column numbers.

```
matrix = {(0, 3): 1, (2, 1): 2, (4, 3): 3}
```

- Note that we use 0 (not 1) as the starting value for row and column.
- Sparse matrices are often used in image processing and board games (chess, tic-tac-toe, etc.)

# Example: Alphabet counting

Write a program that asks user for a string in English and returns a table of the letters of the English alphabet in alphabetical order which occur in the string together with the number of times each letter occurs. Case should be ignored.

A sample run of the program would look like this:

Enter a string: ThiS is a String with Upper and lower case Letters.

Here are the alphabet counting results.

a 3

c 1

d 1

e 5

g 1

h 2

i 4

…