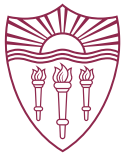
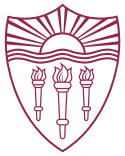


DSCI 510
PRINCIPLES OF PROGRAMMING FOR DATA SCIENCE

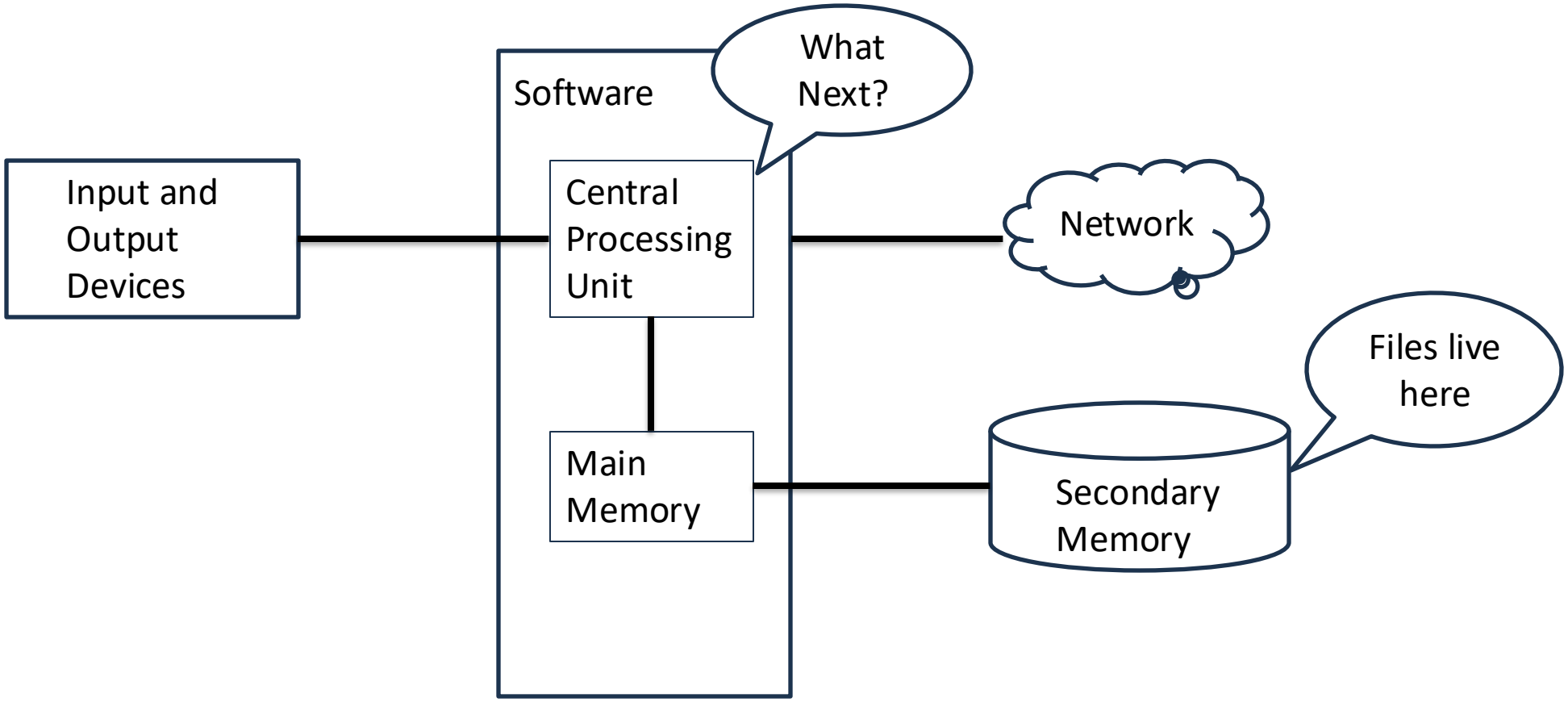
Itay Hen



File Processing in Python

FILES

Files and Persistent Storage



Binary vs text files

- In general, there are two main types of files that we can handle and process.

Text Files

- Store data as human-readable characters (letters, digits, symbols).
- Encoded in standards like **ASCII** or **UTF-8**.
- Examples: `.txt`, `.csv`, `.html`.
- Easy to open and edit with a text editor.

Binary Files

- Store data as raw bytes (not directly human-readable).
- May represent images, audio, video, executables, etc.
- Examples: `.jpg`, `.mp3`, `.exe`.
- Require special programs to interpret and display correctly.

Key Difference

- **Text files** → readable and editable as plain text.
- **Binary files** → optimized for storage/processing, need decoding.

File Processing

- A text file can be thought of as a sequence of lines.
- A snippet from mbox-short.txt:

```
From stephen.marquard@uct.ac.za Sat Jan  5 09:14:16 2008
Return-Path: <postmaster@collab.sakaiproject.org>
Received: from murder (mail.umich.edu [141.211.14.90])
        by frankenstein.mail.umich.edu (Cyrus v2.3.8) with LMTPA;
        Sat, 05 Jan 2008 09:14:16 -0500
X-Sieve: CMU Sieve 2.3
Received: from murder ([unix socket])
        by mail.umich.edu (Cyrus v2.2.12) with LMTPA;
        Sat, 05 Jan 2008 09:14:16 -0500
Received: from holes.mr.itd.umich.edu (holes.mr.itd.umich.edu [141.211.14.79])
       by flawless.mail.umich.edu () with ESMTP id m05EEFR1013674;
       Sat, 5 Jan 2008 09:14:15 -0500
```

- http://www.py4inf.com/code/mbox-short.txt
- http://www.py4inf.com/code/mbox.txt
- Will be uploaded to Brightspace

Opening a File

- Before we can read the contents of the file, we must tell Python which file we are going to work with and what we will be doing with the file
- This is done with the open() function
- open() returns a "file handle" - a variable used to perform operations on the file
- Similar to "File -> Open" in a Word Processor

Using open()

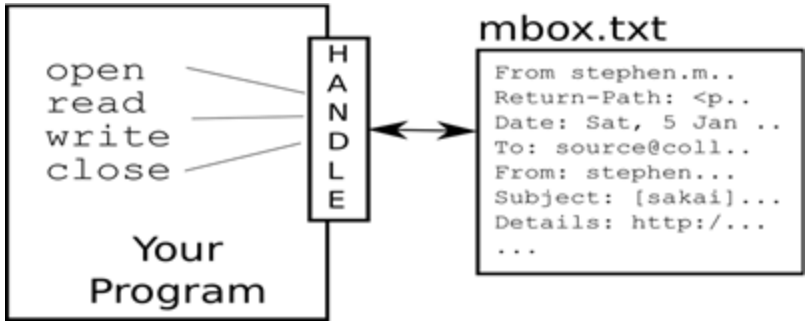
- file_handle = open(filename, mode)
- **returns a handle used to manipulate the file**
- filename **is a string**
- mode **is optional and should be 'r' (default) if we are planning to only read the file and 'w' if we are going to write to the file**

- file_handle = open('mbox.txt', 'r')

What is a Handle?

```
>>> f = open('mbox.txt')
>>> f
<_io.TextIOWrapper name='mbox.txt' mode='r' encoding='UTF-8'>
>>>
```

- f is of type _io.TextWrapper
- mode is 'r' by default
- encoding is 'UTF-8' by default



When Files are Missing

```
>>> f = open('missing.txt')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
FileNotFoundError: [Errno 2] No such file or directory: 'missing.txt'
>>>
```

You get a FileNotFoundError
if you try to read from a missing file.

- 'missing.txt' is a relative path
- This means python will look for this file in the current directory
- To check current directory in Python ,

```
>>> import os
>>> os.getcwd()
'/Users/itayhen'
```

But What if it is Opened for Writing

```
>>> open('missing.txt', 'w')
```

- You often want to write to a new file, so
- No problem if it doesn’t exist in the 'w' mode
- https://docs.python.org/3/library/functions.html#open

open(filename, mode)

Character	Meaning
`'r'`	open for reading (default)
`'w'`	open for writing, truncating the file first
`'x'`	open for exclusive creation, failing if the file already exists
`'a'`	open for writing, appending to the end of file if it exists
`'b'`	binary mode

- Text mode (default): returns contents as str (bytes decoded)
- Binary mode: return contents as bytes (no decoding).

The **newline** Character

```
>>> a_string = "Hello\nWorld"
>>> a_string
'Hello\nWorld'
>>> print(a_string)
Hello
World
>>> a_string = 'X\nY'
>>> print(a_string)
X
Y
>>> len(a_string)
3
>>>
```

- We use a special character called "**newline**" to indicate when a line ends.
- Is it represented as **\n** in strings
- **Newline** is still one character. not two.
- It is an example of ann **escape sequence:** a way to represent characters in a string that are otherwise difficult or impossible to type directly (like newlines, tabs, quotes, or non-printable characters).
- Escape sequences always start with a **backslash** \ followed by one or more characters

Escape sequences

🔷 Common Escape Sequences

Here are the most frequently used ones:

Escape Sequence	Meaning	Example	Output
`\n`	Newline	`print("Hello\nWorld")`	`Hello World`
`\t`	Horizontal tab	`print("Hello\tWorld")`	`Hello World`
`\'`	Single quote	`print('It\'s fine')`	`It's fine`
`\"`	Double quote	`print("She said \"Hi\"")`	`She said "Hi"`
`\\`	Backslash itself	`print("C:\\path\\to\\file")`	`C:\path\to\file`

File Processing

A text file has **newlines** at the end of each line

```
From stephen.marquard@uct.ac.za Sat Jan  5 09:14:16 2008\n
Return-Path: <postmaster@collab.sakaiproject.org>\n
Date: Sat, 5 Jan 2008 09:12:18 -0500\n
To: source@collab.sakaiproject.org\n
From: stephen.marquard@uct.ac.za\n
Subject: [sakai] svn commit: r39772 - content/branches/\n
\n
Details: http://source.sakaiproject.org/viewsvn/?view=rev&rev=39772\n
```

File Handle as a Sequence

- A file handle opened for read can be treated as a sequence of strings where each line in the file is a string in the sequence
- We can use the for statement to iterate through a sequence

```
>>> file_handler = open('mbox.txt')
>>> for line in file_handler:
...     print(line)
...
```

Counting Lines in a File

```
>>> f = open('mbox.txt')
>>> count = 0
>>> for line in f:
...     count += 1
...
>>> print(f'Line Count: {count}')
Line Count: 132045
>>> f.close()
>>>
```

- Open a **file** in read mode
- Use for loop for read each line
- **Count** the lines and print out the number of lines

Always close the file using the f.close() function

Reading the **Whole** File using .read()

```
>>> f = open('mbox.txt')
>>> file_content = f.read()
>>> print(len(file_content))
6685028
>>> print(file_content[:20])
From stephen.marquar
>>> print(type(file_content))
<class 'str'>
```

- You can read the **whole** file (newlines and all) into a single **string**.
- Avoid doing this when dealing with large files (eg: of the size of GBs.)
- The whole file will be read into memory.
- More often than not, you would read the file line by line.

Reading the **Whole** File using .readlines()

```
>>> f = open('mbox.txt')
>>> file_content = f.readlines()
>>> len(file_content)
132045
>>> file_content[:2]
['From stephen.marquard@uct.ac.za Sat Jan  5
>>> type(file_content)
<class 'list'>
```

- You can also read the **whole** file using the readlines() **function** into a **list**.
- Each line, delimited by the **newline** character will be an item in the **list**.
- This also reads the whole file into memory.

Filter the Contents of a File

```
>>> f = open('mbox-short.txt')
>>> for line in f:
...     if line.startswith('From:'):
...         print(line)
...
From: stephen.marquard@uct.ac.za \n
\n
From: louis@media.berkeley.edu \n
\n
From: zqian@umich.edu \n
\n
From: rjlowe@iupui.edu \n
\n
From: zqian@umich.edu \n
\n
From: rjlowe@iupui.edu \n
\n
From: cwen@iupui.edu \n
\n
From: cwen@iupui.edu \n
\n
```

- Read the file, line by line
- Print the line, only if it starts with "From:"
- Why do we have these blank lines?
  - Each line from the file has a newline at the end.
  - The print statement adds a **newline** to each line

Filter the Contents of a File

```
>>> f = open('mbox-short.txt')
>>> for line in f:
...     line = line.rstrip()
...     if line.startswith('From:'):
...         print(line)
...
From: stephen.marquard@uct.ac.za
From: louis@media.berkeley.edu
From: zqian@umich.edu
From: rjlowe@iupui.edu
From: zqian@umich.edu
From: rjlowe@iupui.edu
From: cwen@iupui.edu
From: cwen@iupui.edu
From: gsilver@umich.edu
From: gsilver@umich.edu
From: zqian@umich.edu
From: gsilver@umich.edu
```

- We can strip the whitespace from the right-hand side of the string using the rstrip() method from the string library.
- The **newline** is considered "white space" and is stripped.
- print() still adds the **newline** so each line is printed in a new line.

Processing contents of a File

```
>>> f = open('mbox-short.txt')
>>> for line in f:
...     line = line.rstrip()
...     if not '@uct.ac.za' in line:
...         continue
...     print(line)
...
From stephen.marquard@uct.ac.za Sat Jan  5 09:14:16 2008
X-Authentication-Warning: nakamura.uits.iupui.edu: apache set sender to stephen.marquard@uct.ac.za using -f
From: stephen.marquard@uct.ac.za
Author: stephen.marquard@uct.ac.za
From david.horwitz@uct.ac.za Fri Jan  4 07:02:32 2008
X-Authentication-Warning: nakamura.uits.iupui.edu: apache set sender to david.horwitz@uct.ac.za using -f
From: david.horwitz@uct.ac.za
Author: david.horwitz@uct.ac.za
r39753 | david.horwitz@uct.ac.za | 2008-01-04 13:05:51 +0200 (Fri, 04 Jan 2008) | 1 line
From david.horwitz@uct.ac.za Fri Jan  4 06:08:27 2008
```

- If 'uct.ac.za' not in the stripped line, continue to the next iteration
- Otherwise print the line

Processing a file with “with”

- We can actually replace the opening and closing of files:

```
f = open("example.txt", "r")
data = f.read()
f.close()  # must not forget!
```

with the simpler control statement / keyword “with”:

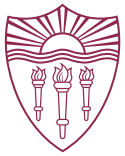
```
with open("example.txt", "r") as f:
    data = f.read()
    print(data)
```

- “with…as” is used to automatically manage opening and closing a file.
- No need to call close() explicitly; cleaner and more readable code.

write() and close()

```
output_file = open('my-notes.txt', 'w')
output_file.write('Make sure to close all files,\n')
output_file.write('especially when writing to a file,\n')
output_file.write('because output is often buffered.\n')
output_file.close()
```

- The .write() method is similar to print() but the content is “printed” to a file



MUTABLE AND IMMUTABLE TYPES

Immutable

immutable [ih-**myoo**-*tuh*-*buh*l] SHOW IPA

See synonyms for: immutable / immutability / immutably on Thesaurus.com

adjective

1. not mutable; unchangeable; changeless.
2. *Computers*. (in object-oriented programming) of or noting an object with a fixed structure and properties whose values cannot be changed.

In object-oriented and functional programming, an **immutable object** is an object whose state or value cannot be modified after it is created

More

✅ **Immutable built-in types in Python**

- **Numbers**
  - `int`
  - `float`
  - `complex`
- **Boolean**
  - `bool`
- **Text**
  - `str`
- **Tuples**
  - `tuple` (but note: if a tuple contains a mutable object like a list, that list can still be changed)
- **Frozen sets**
  - `frozenset`
- **Binary data**
  - `bytes` (immutable version of `bytearray`)

❌ **Mutable built-in types**

- `list`
- `dict`
- `set`
- `bytearray`

The id() function

- The id() function returns an integer representing an object's identity.
- ID is guaranteed to be unique integer, and it will never change during the life of an object.
- In practice, we rarely use id() function. Identity checks are often done with the is operator, which compares the object IDs so our code doesn't need to call id() explicitly
- It is also used while debugging, when you need to understand whether two references are aliases or point to separate objects.

Lists are Mutable

```
>>> x = [1, 4, 6, 7]
>>> x
[1, 4, 6, 7]
>>> id(x)
4319519232
>>> x.append(8)
>>> x
[1, 4, 6, 7, 8]
>>> id(x)
4319519232
>>>
```

assign a list to the variable x

print x

print ID of x

append 8 to the list x

print x

print ID of x, it is the same as before

Mutable vs Immutable

```
>>>
>>> x=3
>>> id(x)
    4359268856
>>> y=x
>>> id(y)
    4359268856
>>> x=x+1
>>> print(x)
    4
>>> print(y)

>>>
```

??

Mutable vs Immutable

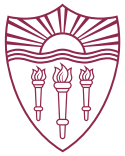
```
>>>
>>> x=3
>>> id(x)
4359268856
>>> y=x
>>> id(y)
4359268856
>>> x=x+1
>>> print(x)
4
>>> print(y)
3
>>> id(x)
4359268888
```

Mutable vs Immutable

```
>>> x=[1,2,3]
>>> id(x)
4352717056
>>> y=x
>>> id(y)
4352717056
>>> x.append(4)
>>> print(x)
[1, 2, 3, 4]
>>> print(y)

>>>

??

>>>
```

Mutable vs Immutable

```
>>> x=[1,2,3]
>>> id(x)
4352717056
>>> y=x
>>> id(y)
4352717056
>>> x.append(4)
>>> print(x)
[1, 2, 3, 4]
>>> print(y)
[1, 2, 3, 4]
>>> id(x)
4352717056
>>> id(y)
4352717056
```

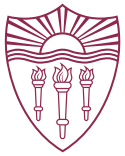


Lists, Dictionaries, Tuples, Sets

COLLECTIONS

Collections

- Collection refers to a data structure or a container that is used to store and organize multiple elements.
- Collections allow you to work with groups of data efficiently.
- **List**: ordered sequence of objects
  - Mutable
  - Allows for duplicates
  - Objects can be of any type (can mix types in same list)
- **Dictionary**: unordered collection of key:value pairs
  - Mutable
  - Keys must be unique (no duplicates)
  - Keys can be of any *immutable* type
  - Values can be of any type
  - Fast to access keys (using hashing)
- **Tuples**: like lists, but immutable
- **Sets**: unordered collection of unique immutable objects



A key-value data structure

DICTIONARIES

How many Jelly Beans are there of Each Flavor?

Can it be done with Lists?

Answer is to use a Dictionary !

Buttered Popcorn	**15**
Chocolate Pudding.	**3**
Mixed Berry Smoothie.	**20**
Orange Sherbet.	**7**
Peach	**7**
Raspberry	**18**
Red Apple	**0**
Sparkling Berry Blue	**1**
Sparkling Blueberry	**…**
Sparkling Cream Soda	
Sparkling Grape Soda	
Sparkling Green Apple	
Sparkling Island Punch	
Sparkling Orange	
Sparkling Sour Apple	
Sparkling Sour Lemon	
Sparkling Very Cherry	
Sparkling Wild Blackberry	
Strawberry Jam	
Watermelon	

Dictionary

- Dictionaries are Python's most powerful data collection.
- Python Dictionaries are highly optimized, driven by *Hash tables*.
- Dictionaries are a key-value data structure.
- Dictionary keys are unique.
- Item access by keys is very fast.
- Dictionaries are mutable.
- Since Python 3.6, Dictionary instances preserve key insertion order. In other words, Dictionaries are ordered (since Python 3.6)

A Dictionary Example

```
>>> purse = dict()
>>> purse['money'] = 12
>>> purse['candy'] = 3
>>> purse['tissues'] = 75
>>> print(purse)
{'money': 12, 'candy': 3, 'tissues': 75}
>>> type(purse)
<class 'dict'>
>>> print(purse['candy'])
3
>>> purse['candy'] = purse['candy'] + 2
>>> print(purse)
{'money': 12, 'candy': 5, 'tissues': 75}
>>>
```

- Lists **index** their entries based on the position in the list.
- In contrast, Dictionaries index their entries based on the keys.
- Dictionary keys must be **hashable**, which means they must be **immutable**.
- The dictionary keys are hashed to determine their position in the hash table. If the key could change, they would potentially end up in a different position in the hash table, leading to inconsistencies and incorrect lookups.

Comparing Lists and Dictionaries

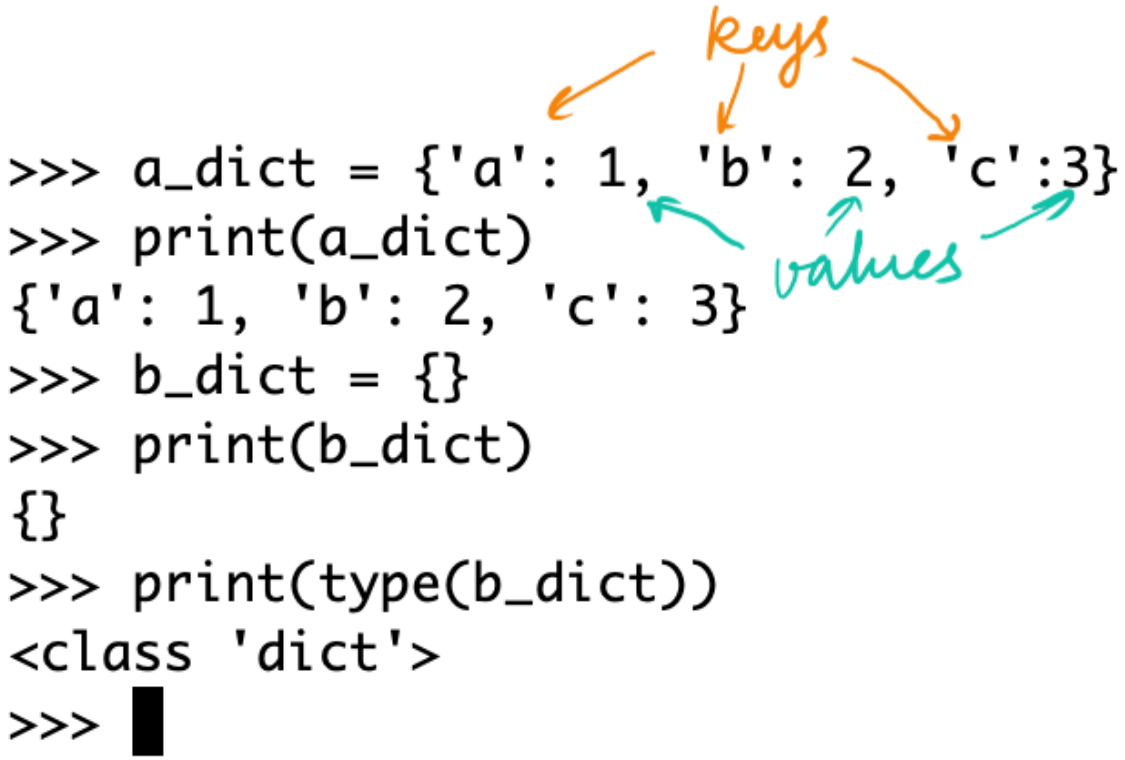
```
>>> numbers_list = []
>>> numbers_list.append(12)
>>> numbers_list.append(21)
>>> numbers_list.append(43)
>>> print(numbers_list)
[12, 21, 43]
>>> numbers_list[0] = 13
>>> print(numbers_list)
[13, 21, 43]
```

Lists use indices to look up values

- >>> jelly_bean_dict = {}
- >>> jelly_bean_dict['strawberry'] = 12
- >>> jelly_bean_dict['blackberry'] = 21
- >>> jelly_bean_dict['apple'] = 43
- >>> print(jelly_bean_dict)
- {'strawberry': 12, 'blackberry': 21, 'apple': 43}
- >>> jelly_bean_dict['strawberry'] = 13
- >>> print(jelly_bean_dict)
- {'strawberry': 13, 'blackberry': 21, 'apple': 43}

Dictionaries use keys to look up values

Dictionary Literals (Constants)



- Dictionary literals use curly braces and have list of key: value pairs.
- You can create an empty dictionary using empty curly braces, OR using the constructor dict()

Counting Different Types of Objects Using Dictionaries

Most Common Name?

marquard	cwen	cwen
zhen	marquard	zhen
csev	zhen	marquard
zhen	csev	zhen

csev ||
cwen ||
zhen ||||
marquard |||

Counting Different Types of Objects Using Dictionaries (1)

```
names = ['marquard', 'cwen', 'cwen', 'zhen',
         'marquard', 'zhen', 'csev', 'zhen',
         'marquard', 'zhen', 'csev', 'zhen']
name_dict = dict()
for name in names:
    if name not in name_dict:
        name_dict[name] = 1
    else:
        name_dict[name] += 1
print(name_dict)
```

{'marquard': 3, 'cwen': 2, 'zhen': 5, 'csev': 2}

- Loop though all the names in the names list.
- When we encounter a new name, add an entry in the dictionary.
- If a name exists in the dictionary, increment the count by 1.
- There is a better way, using the get() method

The get() method

```
>>> a_dict = {'a': 1, 'b': 2, 'c': 3}
>>> print(a_dict['d'])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'd'
>>>
>>> print(a_dict.get('c'))
3
>>> print(a_dict.get('d'))
None
>>>
>>> print(a_dict.get('d', 0))
0
>>>
```

Accessing a nonexistent key raises KeyError

You can also use the get() method to get values by keys, in addition of using []

get() returns None if a value does not exist, no KeyError

We can ask the get() method to return a default value is the key does not exist in the dictionary

Counting Different Types of Objects Using Dictionaries (2)

```
names = ['marquard', 'cwen', 'cwen', 'zhen',
         'marquard', 'zhen', 'csev', 'zhen',
         'marquard', 'zhen', 'csev', 'zhen']
name_dict = dict()
for name in names:
    name_dict[name] = name_dict.get(name, 0) + 1
print(name_dict)
```

```
{'marquard': 3, 'cwen': 2, 'zhen': 5, 'csev': 2}
```

- Loop through all the names in the list
- Use the get() method to retrieve the name from the dictionary, if the name does not exist, a default value of 0 is returned.

Iterating over Dictionaries

```
>>> d = {'a': 1, 'b': 2, 'c': 3}
>>> for key in d:
...     print(key, d[key])
...
a 1
b 2
c 3
>>>
>>> d.keys()
dict_keys(['a', 'b', 'c'])
>>>
>>> d.values()
dict_values([1, 2, 3])
>>>
>>> d.items()
dict_items([('a', 1), ('b', 2), ('c', 3)])
>>>
>>>
```

Iterate over the keys in the dictionary using the for loop

The keys() method returns a list of keys in the dictionary

The values() method returns a list of values in the dictionary

The items() method returns a list of (key, value) tuples in the dictionary

Iterating over Dictionaries

🔷 Iterate over keys (default)

python

```python
my_dict = {"a": 1, "b": 2, "c": 3}

for key in my_dict:
    print(key)   # 'a', 'b', 'c'
```

Equivalent to:

python

```python
for key in my_dict.keys():
    print(key)
```

🔷 Iterate over values

python

```python
for value in my_dict.values():
    print(value)   # 1, 2, 3
```

🔷 Iterate over key–value pairs

python

```python
for key, value in my_dict.items():
    print(key, value)
# Output:
# a 1
# b 2
# c 3
```

Iterating over Dictionaries

🔷 Iterate with index (using `enumerate`)

python

```python
for i, (key, value) in enumerate(my_dict.items()):
    print(i, key, value)
```

Example output:

css

```
0 a 1
1 b 2
2 c 3
```

Dictionary Functions/Methods

list(d) Return a list of all the keys used in the dictionary d.

len(d) Return the number of items in the dictionary d.

del d[key] Remove d[key] from d. Raises a KeyError if ***key*** is not in the map.

clear() Remove all items from the dictionary (method). Leaves the dictionary empty.

pop(*key*[, *default*]) If key is in the dictionary, remove it and return its value, else return default. If default is not given and key is not in the dictionary, a KeyError is raised.

popitem() Remove and return a (key, value) pair from the dictionary. Pairs are returned in LIFO order.

Dictionary Functions/Methods

reversed(d) Return a reverse iterator over the keys of the dictionary. This is a shortcut for reversed(d.keys()).

setdefault(*key*[, *default*]) If key is in the dictionary, return its value. If not, insert key with a value of default and return default. default defaults to None.

update([*other*]) Update the dictionary with the key/value pairs from other, overwriting existing keys. Return None.

d | other Create a new dictionary with the merged keys and values of d and other, which must both be dictionaries. The values of other take priority when d and other share keys.

d |= other Update the dictionary d with keys and values from other, which may be either a mapping or an iterable of key/value pairs. The values of other take priority when d and other share keys.