Chapter 6-7: Modules and Files

Contents

- Modules
- Packages
- Some useful modules: random, time, math, numpy
- Files
- Json format
- Pickle

Modules

Reasons for using modules

- Definitions of functions and variables are lost after quitting each session of the Python interpreter
- Creating a script file of your Python input for each individual program would be cumbersome and redundant
- Modules allow you to import code from an external file and use it in your program/script, in a similar fashion as libraries used in C++ and other languages

Importing modules

- A module is a file containing Python definitions and statements.
- File name is the module's name with .py extension
- The "import" command allows the functions/statements of the module to be accessed locally
- Import functions/statements of "module.py"

```
>>> import module
```

- Use a function from "module.py"

```
>>> module.function(argument)
```

- Copy the function to a local name

```
>>> localFunction = module.function
>>> localFunction(argument)
```

Importing modules (cont.)

- Can import selected portions of the module

- Import selected names in the module as local names

```
>>> from module import function1, function2
>>> function1(argument)
>>> function2(argument)
```

- Import all names except those that start with "_"

```
>>> from module import *
```

Executing modules as scripts

- When importing modules, the global variable __name__ is set to the module's name
- When running a module directly from the command line, the __name__ variable is instead set to __main__
- This allows the ability to execute code depending on whether the module is being used as a script or as an imported module

```
$ python module.py arguments (__name__="__main__")
   >>> import module (__name__="module")
```

Executing modules as scripts (cont.)

- Within “module.py”:

```
if __name__ == "__main__":
    import sys
    function(sys.argv[1])
```

- The code will only be executed if the module is ran as the “main” file from the command line
- The “sys” module is needed to access the command line arguments, which are then defined by “sys.argv”

Executing modules as scripts (example)

- Create a file called **fibo.py**

```
# Fibonacci numbers module

def fib(n):    # write Fibonacci series up to n
    a, b = 0, 1
    while a < n:
        print(a, end=' ')
        a, b = b, a+b
    print()

def fib2(n):   # return Fibonacci series up to n
    result = []
    a, b = 0, 1
    while a < n:
        result.append(a)
        a, b = b, a+b
    return result
```

Executing modules as scripts (example)

```
if __name__ == "__main__":
    import sys
    fib(int(sys.argv[1]))
```

- The file can be used as a script

```
>>> python fibo.py 50
0 1 1 2 3 5 8 13 21 34
```

- If the module is imported, the above code is not run:

```
>>> import fibo
```

Searching for modules

- When a module is imported, the python interpreter first looks for a built-in module
- If not found, it will then search for “moduleName.py” in the locations defined by the variable “sys.path”, which initially defines these locations:
  - the directory containing the script currently running
  - the locations defined in “PYTHONPATH”
  - the default installation directory
- Python programs can modify “sys.path” after initialization

“Compiled” Python Files

- If files mod.pyc and mod.py are in the same directory, there is a byte-compiled version of the module mod
- The modification time of the version of mod.py used to create mod.pyc is stored in mod.pyc
- Normally, the user does not need to do anything to create the .pyc file
- A compiled .py file is written to the .pyc
  - No error for failed attempt, .pyc is recognized as invalid
- Contents of the .pyc can be shared by different machines

Some Tips

- -O flag generates optimized code and stores it in .pyo files
  - Only removes assert statements
  - .pyc files are ignored and .py files are compiled to optimized bytecode
- Passing two –OO flags
  - Can result in malfunctioning programs
  - _doc_ strings are removed
- Same speed when read from .pyc, .pyo, or .py files, .pyo and .pyc files are loaded faster
- Startup time of a script can be reduced by moving its code to a module and importing the module
- Can have a .pyc or .pyo file without having a .py file for the same module
- Module compileall creates .pyc or .pyo files for all modules in a directory

Standard, or “built-in” modules

- Standard modules are defined in the Python Library Reference file
- Standard modules can vary across different computer platforms
- Most notable module is “sys”, which is built in to all Python interpreters

```
>>> import sys
>>> sys.path.append('/ufs/guido/lib/python')
```

- Adds a path to the locations searched for modules

The built-in dir() function

- Is used to find out which names are defined by a module
- Returns the names as a sorted list of strings

```
>>> import module
>>> dir(module)
['__name__', 'function1', 'function2']
>>> a = 5
>>> dir()
['__builtins__', '__name__', 'module', 'a', 'function1',
 'function2']
```

- When called without an argument, dir() lists all currently defined names

Packages

Packages

- A package is a collection of modules
- A file named "__init__.py" must be placed in the directory of the package, to indicate that the directory is indeed a package
- "__init.py__" can also contain executable code
- Packages can contain other packages, or "subpackages"

```
>>> import package
>>> import package.subpackage.function
>>> package.subpackage.function(argument)
>>> from package.subpackage import function
>>> function(argument)
```

A package for the uniform handling of sound files and sound data

```
sound/                          Top-level package
      __init__.py               Initialize the sound package
      formats/                  Subpackage for file format conversions
              __init__.py
              wavread.py
              wavwrite.py
              aiffread.py
              aiffwrite.py
              auread.py
              auwrite.py
              ...
      effects/                  Subpackage for sound effects
              __init__.py
              echo.py
              surround.py
              reverse.py
              ...
      filters/                  Subpackage for filters
              __init__.py
              equalizer.py
              vocoder.py
              karaoke.py
              ...
```

A package for the uniform handling of sound files and sound data

- Users of the package can import individual modules from the package, for example:

```
import sound.effects.echo
```

- This loads the submodule `sound.effects.echo`. It must be referenced with its full name.

```
sound.effects.echo.echofilter(input, output, delay=0.7, atten=4)
```

- An alternative way of importing the submodule is:

```
from sound.effects import echo
```

- This also loads the submodule echo, and makes it available without its package prefix, so it can be used as follows:

```
echo.echofilter(input, output, delay=0.7, atten=4)
```

A package for the uniform handling of sound files and sound data

- Another variation is to import the desired function or variable directly:

```
from sound.effects.echo import echofilter
```

- This loads the submodule `echo`, but this makes its function `echofilter`() directly available:

```
echofilter(input, output, delay=0.7, atten=4)
```

Importing * From a Package

```
>>> from package.subpackage import *
```

- When the asterisk is used, the names defined by “__all__” in the package will be imported.
- For example, the file sound/effects/__init__.py could contain the following code:

```
__all__ = ["echo", "surround", "reverse"]
```

- This would mean that from sound.effects import * would import the three named submodules of the sound package.
- If “__all__” is not defined, it will only import the package name and names of any modules contained in it

Some useful modules

random, time, math, numpy

Random numbers

We often want to use random numbers in programs, here are a few typical uses:

- To play a game of chance where the computer needs to throw some dice, pick a number, or flip a coin,
- To shuffle a deck of playing cards randomly,
- To allow/make an enemy spaceship appear at a random location and start shooting at the player,
- To simulate possible rainfall when we make a computerized model for estimating the environmental impact of building a dam,
- For encrypting banking sessions on the Internet.

Random numbers

- Python provides a module `random`

```
import random

# Create a black box object that generates ra
ndom numbers

rng = random.Random()

dice_throw = rng.randrange(1,7)

# Return an int, one of 1,2,3,4,5,6

delay_in_seconds = rng.random() * 5.0

random_odd = rng.randrange(1, 100, 2)
```

Random numbers

- Shuffle a list

```
cards = list(range(52))   # Generate ints [0 .. 51]
                          # representing a pack of cards
rng.shuffle(cards)        # Shuffle the pack
```

The `time` module

- As we start to work with more sophisticated algorithms and bigger programs, a natural concern is “is our code efficient?”
- One way to experiment is to time how long various operations take. The `time` module has a function called clock that is recommended for this purpose.
- Whenever clock is called, it returns a floating point number representing how many seconds have elapsed since your program started running.

The time module

```
import time

def do_my_sum(xs):
    sum = 0
    for v in xs:
        sum += v
    return sum

sz = 10000000          # Lets have 10 million elements in the list
testdata = range(sz)

t0 = time.clock()
my_result = do_my_sum(testdata)
t1 = time.clock()
print("my_result    = {0} (time taken = {1:.4f} seconds)"
        .format(my_result, t1-t0))

t2 = time.clock()
their_result = sum(testdata)
t3 = time.clock()
print("their_result = {0} (time taken = {1:.4f} seconds)"
        .format(their_result, t3-t2))
```

The math module

- The math module contains the kinds of mathematical functions you'd typically find on your calculator (sin, cos, sqrt, asin, log, log10) and some mathematical constants like pi and e:

```
>>> import math
>>> math.pi                    # Constant pi
3.141592653589793
>>> math.e                     # Constant natural log base
2.718281828459045
>>> math.sqrt(2.0)             # Square root function
1.4142135623730951
>>> math.radians(90)           # Convert 90 degrees to radians
1.5707963267948966
>>> math.sin(math.radians(90))  # Find sin of 90 degrees
1.0
>>> math.asin(1.0) * 2  # Double the arcsin of 1.0 to get pi
3.141592653589793
```

numpy

- The standard Python data types are not very suited for mathematical operations. For example, suppose we have the list a = [2, 3, 8]. If we multiply this list by an integer, we get:

```
>>> a = [2, 3, 8]
>>> 2 * a
[2, 3, 8, 2, 3, 8]
```

- And float's are not even allowed:

```
>>> a = [2, 3, 8]
>>> 2 * a
>>> 2.1 * a
TypeError: can't multiply sequence by non-
int of type 'float'
```

numpy

- This is because Python list's are not designed as mathematical objects. Rather, they are purely a collection of items. In order to get a type of list which behaves like a mathematical array or matrix, we use Numpy.

```
>>> import numpy as np
>>> a = np.array([2, 3, 8])
>>> 2.1 * a
array([  4.2,   6.3,  16.8])
>>> a = np.array([2, 3, 8])
>>> a * a
array([ 4,  9, 64])
>>> a**2
array([ 4,  9, 64])
```

Numpy: shape

- To get the shape of an array, we use shape:

```
>>> import numpy as np
>>> a = np.array([2, 3, 8])
>>> a.shape
(3,)
>>> b = np.array([
    [2, 3, 8],
    [4, 5, 6],
    ])
>>> b.shape
(2, 3)
```

Numpy: slicing

```
>>> a = np.array([2, 3, 8])
>>> a[2]
8
>>> a[1:]
np.array([3, 8])

>>> b = np.array([
    [2, 3, 8],
    [4, 5, 6],
    ])
>>> b[1]
array([4, 5, 6])
>>> b[1][2]
6
>>> b[1, 2]
6
>>> b[:, 1]
array([3, 5])
```

Numpy: masking

```
>>> a = np.array([230, 10, 284, 39, 76])
>>> cutoff = 200
>>> a > cutoff
np.array([True, False, True, False, False])
>>> a = np.array([230, 10, 284, 39, 76])
>>> cutoff = 200
>>> a[a > cutoff] = 0
>>> a
np.array([0, 10, 0, 39, 76])
```

Numpy: broadcasting

- Another powerful feature of Numpy is broadcasting. Broadcasting takes place when you perform operations between arrays of different shapes.

```
>>> a = np.array([
    [0, 1],
    [2, 3],
    [4, 5],
    ])
>>> b = np.array([10, 100])
>>> a * b
array([[  0, 100],
       [ 20, 300],
       [ 40, 500]])
```

Numpy: broadcasting

- The shapes of a and b don't match. In order to proceed, Numpy will stretch b into a second dimension, as if it were stacked three times upon itself.
- One of the rules of broadcasting is that only dimensions of size 1 can be stretched (if an array only has one dimension, all other dimensions are considered for broadcasting purposes to have size 1).
- In the example above b is 1D, and has shape (2,). For broadcasting with a, which has two dimensions, Numpy adds another dimension of size 1 to b. b now has shape (1, 2). This new dimension can now be stretched three times so that b's shape matches a's shape of (3, 2).

Numpy: broadcasting

- The other rule is that dimensions are compared from the last to the first. Any dimensions that do not match must be stretched to become equally sized. However, according to the previous rule, only dimensions of size 1 can stretch. This means that some shapes cannot broadcast and Numpy will give you an error:

```
>>> c = np.array([
    [0, 1, 2],
    [3, 4, 5],
    ])
>>> b = np.array([10, 100])
>>> c * b
ValueError: operands could not be broadcast together wi
th shapes (2,3) (2,)
```

Numpy: broadcasting

- Numpy adds a dimension to b, making it of shape (1, 2). The sizes of the last dimensions of b and c (2 and 3, respectively) are then compared and found to differ. Since none of these dimensions is of size 1 (therefore, unstretchable) Numpy gives up and produces an error.
- To do this, specifically tell Numpy that it must add that extra dimension as the second dimension of b. This is done by using None to index that second dimension.

```
>>> c = np.array([
    [0, 1, 2],
    [3, 4, 5],
    ])
>>> b = np.array([10, 100])
>>> c * b[:, None]
array([[  0,  10,  20],
       [300, 400, 500]])
```

Numpy: dtype

- A commonly used term in working with numpy is dtype - short for data type. This is typically int or float, followed by some number, e.g. int8. This means the value is integer with a size of 8 bits.
- What happens when you set numbers bigger than the maximum value of your dtype?

```
>>> import numpy as np
>>> a = np.array([200], dtype='uint8')
>>> a + a
array([144], dtype=uint8)
```

Numpy: dtype

- To fix this, you should make sure that your numbers where not stored as uint8, but as something larger; uint16 for example.

```
>>> import numpy as np
>>> a = np.array([200], dtype='uint16')
>>> a + a
array([400], dtype=uint16)
```

- To change the dtype of an existing array, you can use the astype method:

```
>>> import numpy as np
>>> a = np.array([200], dtype='uint8')
>>> a.astype('uint64')
```

Files

Reading and writing files

File Processing

- A text file can be thought of as a sequence of lines

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

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
- Kind of like “File -> Open” in a Word Processor

Using open()

- handle = open(filename, mode)
- returns a handle use to manipulate the file
- filename is a string
- mode is 'r' if we are planning reading the file and 'w' if we are going to write to the file; and 'a' opens the file for appending; any data written to the file is automatically added to the end. 'r+' opens the file for both reading and writing.
- The mode argument is optional; 'r' will be assumed if it's omitted.

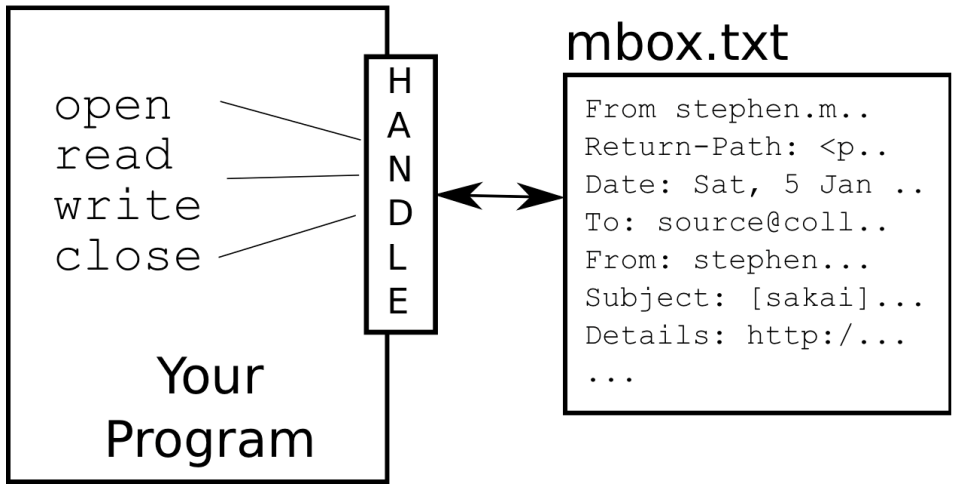
```
fhand = open('mbox.txt', 'r')
```

Using open()

- Normally, files are opened in text mode, that means, you read and write strings from and to the file, which are encoded in a specific encoding. If encoding is not specified, the default is platform dependent.
- 'b' appended to the mode opens the file in binary mode: now the data is read and written in the form of bytes objects. This mode should be used for all files that don’t contain text.

What is a File Handle?

```
>>> fhand = open('mbox.txt')
>>> print(fhand)
```



When Files are Missing

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

The newline Character

- We use a special character to indicate when a line ends called the "newline"
- We represent it as \n in strings
- Newline is still one character - not two

```
>>> stuff = 'Hello\nWorld!'
>>> print(stuff
Hello
World!

>>> stuff = 'X\nY'
>>> print(stuff)
X
Y

>>> len(stuff)
3
```

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 \nTo:
source@collab.sakaiproject.org\nFrom: stephen.marquard@uct.ac.za\nSubject:
[sakai] svn commit: r39772 - content/branches/\nDetails:
http://source.sakaiproject.org/viewsvn/?view=rev&rev=39772\n

File Handle as a Sequence

- A file handle open 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
- Remember - a sequence is an ordered set

```
xfile = open('mbox.txt')
for cheese in xfile:
    print(cheese)
```

Counting Lines in a File

- Open a file read-only
- Use a for loop to read each line
- Count the lines and print(out the number of lines

```
fhand = open('mbox.txt')
count = 0
for line in fhand:
    count = count + 1
print('Line Count:', count)
Line Count: 132045
```

Reading the *Whole* File

- We can read the whole file (newlines and all) into a single string.

```
>>> fhand = open('mbox-short.txt')
>>> inp = fhand.read()
>>> print(len(inp)
94626
>>> print(inp[:20]
From stephen.marquar
```

Searching Through a File

- We can put an if statement in our for loop to only print lines that meet some criteria

```
fhand = open('mbox-short.txt')
for line in fhand:
    if line.startswith('From:'):
        print(line)
```

OOPS!

- What are all these blank lines doing here?

From: stephen.marquard@uct.ac.za

From: louis@media.berkeley.edu

From: zqian@umich.edu

From: rjlowe@iupui.edu...

OOPS!

- The print statement adds a newline to each line.
- Each line from the file also has a newline at the end.

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

Searching Through a File (fixed)

- We can strip the whitespace from the right hand side of the string using rstrip() from the string library
- The newline is considered "white space" and is stripped

```
fhand = open('mbox-short.txt')
for line in fhand:
      line = line.rstrip()
      if line.startswith('From:') :
            print(line)
```

Skipping with continue

- We can convienently skip a line by using the continue statement

```
fhand = open('mbox-short.txt')
for line in fhand:
    line = line.rstrip()

    # Skip 'uninteresting lines'
    if not line.startswith('From:') :
        continue
    print(line) # Process our 'interesting' line
```

Using in to select lines

- We can look for a string anywhere in a line as our selection criteria

```
fhand = open('mbox-short.txt')
for line in fhand:
      line = line.rstrip()
      if not '@uct.ac.za' in line :
         continue
   print(line)
```

Prompt for File Name

```
fname = input('Enter the file name:  ')
fhand = open(fname)
count = 0
for line in fhand:
    if line.startswith('Subject:') :
        count = count + 1
print('There were', count, 'subject lines in', fname)
```

Enter the file name: mbox.txt

There were 1797 subject lines in mbox.txt

Bad File Names

```
fname = input('Enter the file name:  ')
try:
    fhand = open(fname)
except:
    print('File cannot be opened:', fname
    exit()

count = 0
for line in fhand:
    if line.startswith('Subject:') :
        count = count + 1
print('There were', count, 'subject lines in', fname)
```

Enter the file name: na na boo boo
File cannot be opened: na na boo boo

With block

- It is good practice to use the with keyword when dealing with file objects. The advantage is that the file is properly closed after its suite finishes, even if an exception is raised at some point.

```
>>> with open('workfile') as f:
        read_data = f.read()

>>> # We can check that the file has been aut
omatically closed.
>>> f.closed
True
```

- If you're not using the with keyword, then you should call `f.close()` to close the file and immediately free up any system resources used by it.

Writing files

- `f.write(string)` writes the contents of string to the file, returning the number of characters written.

```
>>> f.write('This is a test\n')
15
```

- Other types of objects need to be converted – either to a string (in text mode) or a bytes object (in binary mode) – before writing them:

```
>>> value = ('the answer', 42)
>>> s = str(value)  # convert the tuple to st
ring
>>> f.write(s)
18
```

Writing files

- `f.write(string)` writes the contents of string to the file, returning the number of characters written.

```
>>> f.write('This is a test\n')
15
```

- Other types of objects need to be converted – either to a string (in text mode) or a bytes object (in binary mode) – before writing them:

```
>>> value = ('the answer', 42)
>>> s = str(value) # convert the tuple to string
>>> f.write(s)
18
```

Writing files

- `f.tell()` returns an integer giving the file object's current position in the file represented as number of bytes from the beginning of the file when in binary mode and an opaque number when in text mode.
- To change the file object's position, use `f.seek(offset, whence)`. The position is computed from adding `offset` to a reference point; the reference point is selected by the `whence` argument.
- A `whence` value of 0 measures from the beginning of the file, 1 uses the current file position, and 2 uses the end of the file as the reference point. `whence` can be omitted and defaults to 0, using the beginning of the file as the reference point.

Writing files

```
>>> f = open('workfile', 'rb+')
>>> f.write(b'0123456789abcdef')
16
>>> f.seek(5)      # Go to the 6th byte in the file
5
>>> f.read(1)
b'5'
>>> f.seek(-3, 2)  # Go to the 3rd byte before the end
13
>>> f.read(1)
b'd'
```

Json format

The JSON format

- Strings can easily be written to and read from a file. Numbers take a bit more effort, since the read() method only returns strings, which will have to be passed to a function like int.
- When you want to save more complex data types like nested lists and dictionaries, parsing and serializing by hand becomes complicated.
- Rather than having users constantly writing and debugging code to save complicated data types to files, Python allows you to use the popular data interchange format called **JSON (JavaScript Object Notation)**.

The JSON format

- The standard module called json can take Python data hierarchies and convert them to string representations; this process is called **serializing**.
- Reconstructing the data from the string representation is called **deserializing**.
- Between serializing and deserializing, the string representing the object may have been stored in a file or data or sent over a network connection to some distant machine.

The JSON format

```
>>> import json
>>> json.dumps([1, 'simple', 'list'])
'[1, "simple", "list"]'
```

- Another variant of the dumps() function, called dump(), simply serializes the object to a text file. So if f is a text file object opened for writing, we can do this:

```
json.dump(x, f)
```

- To decode the object again, if f is a text file object which has been opened for reading:

```
x = json.load(f)
```

Pickle

Importing Pickle

- Contrary to JSON, pickle is a protocol which allows the serialization of arbitrarily complex Python objects. As such, it is specific to Python and cannot be used to communicate with applications written in other languages.
- To use pickle, you have to import it:

```
import pickle
```

- You can also use cPickle, which works exactly the same but is written in c (and is much faster).

```
import cPickle as pickle
```

Pickle module: Two Primary Methods

- `dump` = dumps an object to a file
- `load` = loads an object from a file

Using Pickle to write an object to a file

```
import pickle
a = ['luke skywalker', 'obi-
wan kenobi', 'princess leia']
fileHandle = open(“testfile”,'wb')
pickle.dump(a, fileHandle)
fileHandle.close()
```

Using Pickle to read an object from a file

```
import pickle
fileHandle = open(“testfile”,'r')
b = pickle.load( fileHandle)
print(b)
fileHandle.close()
>>>> ['luke skywalker', 'obi-
wan kenobi', 'princess leia']
```

Pickle Example:

```
import cPickle as pickle
favorite_color = {"lion": "yellow", "kitty": "red"
, "jaguar": "gold"}
pickle.dump(favorite_color, open("save.p", "wb"))

# Load the dictionary back from the pickle file.
cat_dictionary = pickle.load(open("save.p", "rb"))

# cat_dictionary is now {"kitty": "red“, "lion": "
yellow", “jaguar”: “gold”}
print(favorite_color)
print(cat_dictionary)
```

References

1. https://docs.python.org/3/tutorial/modules.html
2. https://docs.python.org/3/tutorial/inputoutput.html
3. https://docs.python.org/3/library/pickle.html#module-pickle

ĐẠI HỌC BÁCH KHOA

25 YEARS ANNIVERSARY SOICT

VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGY

Thank you for your attention!

soict.hust.edu.vn/ fb.com/groups/soict