Lesson 4 - Useful Standard Library Packages

March 6, 2020

Lesson 4: Useful Standard Library Packages

[Python Standard Library for Logging]

4.1 logging

`logging` - "logging facility for Python"

The Python standard library comes with a logging module that provides most of the basic logging features. By setting it up correctly, a log message can bring a lot of useful information about when and where the log is fired as well as the log context, such as the running process/thread. 1

- 4.1.1 Importing the `logging` module

```
[ ]: import logging
```

- 4.1.2 Log Levels

There are 5 standard log levels:

Level	When it's used
DEBUG	Detailed information, typically of interest only when diagnosing problems.
INFO	Confirmation that things are working as expected.
WARNING	An indication that something unexpected happened, or indicative of some problem in the near future (e.g. 'disk space low'). The software is still working as expected.
ERROR	Due to a more serious problem, the software has not been able to perform some function.
CRITICAL	A serious error, indicating that the program itself may be unable to continue running.

The logging functions are named after the level or severity of the events they are used to track. 3

- 4.1.3 The Default Logger

The logging module provides you with a default logger that allows you to get started without needing to do much configuration.

The corresponding methods for each level can be called as shown in the following example:

[2]:
```
import logging

logging.debug('This is a debug message')
logging.info('This is an info message')
logging.warning('This is a warning message')
logging.error('This is an error message')
logging.critical('This is a critical message')
```

```
WARNING:root:This is a warning message
ERROR:root:This is an error message
CRITICAL:root:This is a critical message
```

Notice that the `debug()` and `info()` messages didn't get logged. This is because, by default, the logging module logs the messages with a severity level of WARNING or above. 2

— **4.1.3.1 The `basicConfig()` method** You can use the `basicConfig()` method to configure the logging

Some of the commonly used parameters for `basicConfig()` are the following:

- level: The root logger will be set to the specified severity level.
- filename: This specifies the file.
- filemode: If filename is given, the file is opened in this mode. The default is a, which means append.
- format: This is the format of the log message. 2

—— **4.1.3.1.1 Setting the Log Level** By using the level parameter, you can set what level of log messages you want to record. This can be done by passing one of the constants available in the class, and this would enable all logging calls at or above that level to be logged. Here's an example:

[1]:
```
import logging

logging.basicConfig(level=logging.DEBUG)
logging.debug('This is a debug message')
```

```
DEBUG:root:This is a debug message
```

It should be noted that calling `basicConfig()` to configure the root logger works only if the root logger has not been configured before. Basically, this function can only be called once. 2

—— **4.1.3.1.2 Logging to a File** A very common situation is that of recording logging events in a file, so let's look at that next.

Note: Be sure to try the following in a newly-started Python interpreter, and don't just continue from the session in the previous example. 3

Let's create a new file named `lab_logging_basic.py` and add the following code:

[ ]:
```
import logging
# specify the filename we want to use for the logs
logging.basicConfig(filename='example.log', level=logging.DEBUG)
logging.debug('This message should go to the log file')
logging.info('So should this')
logging.warning('And this, too')
```

Let's run the script:

(command-line)

```
$ python3 lab_logging_basic.py
$
```

Let's check the contents of the log file:

(command-line)

```
$ cat example.log
DEBUG:root:This message should go to the log file
INFO:root:So should this
WARNING:root:And this, too
```

—— 4.1.3.1.3 Changing the format of displayed messages

Why format your log messages? Formatting your log messages can add context information to your application logs.

Let's create a new file named `lab_logging_basic_formatted.py` and add the following code:

[ ]:
```
import logging
# specify the format we want to use
logging.basicConfig(
    level=logging.DEBUG,
    format='[%(asctime)s] %(levelname)s %(module)s %(lineno)d - %(message)s'
)
logger.debug('This is a debug message')
logger.info('This is an info message')
logger.warning('This is a warning message')
```

Let's run the script:

(command-line)

```
$ python3 lab_logging_basic_formatted.py
[2020-02-18 22:27:41,121] DEBUG lab_logging_basic_formatted 7 - This is a debug message
```

```
[2020-02-18 22:27:41,121] INFO lab_logging_basic_formatted 8 - This is an info message
[2020-02-18 22:27:41,121] WARNING lab_logging_basic_formatted 9 - This is a warning message
```

As you can see, we've changed the format of the log message by specifying the format we wanted to use.

We've also placed the following in our format string:

- `%(asctime)s` - to display the date and time of an event
- `%(levelname)s` - to display the log level
- `%(module)s` - to display the module name
- `%(lineno)d` - to specify the line number
- `%(message)s` - to display the message

- 4.1.4 "Just Enough" Logging for Python Scripts

— **4.1.4.1 Creating a Custom Logger (using the `logger.getLogger(name)` method)** We can create a new logger by using the `getLogger(name)` method. It expects a name for the logger.

—— **4.1.4.1.1 Not specifying a logger name** If you don't pass/specify a name, it's going to use the root logger's name.

[1]:
```
import logging

# create a custom logger without specifying a name
logger = logging.getLogger()
logger.debug('This is a debug message')
logger.info('This is an info message')
logger.warning('This is a warning message')
```

```
This is a warning message
```

—— **4.1.4.1.2 Using `__name__` as the logger name** When naming loggers, it's good practice to use the `__name__` variable because the `__name__` variable will hold the name of the module (python file).

[ ]:
```
logger = logging.getLogger(__name__)
```

—— **4.1.4.1.3 "Just Enough" config for your Python scripts** Here's a simple example that you can use for your Python scripts.

Create a file named `lab_logging_custom.py` and add the following code:

[ ]:
```
import logging
# configure the root logger
# set the log level to info and specify the format we want to use
logging.basicConfig(
```

```
    level=logging.DEBUG,
    format='[%(asctime)s] %(levelname)s %(module)s %(lineno)d - %(message)s',
)
# create a custom logger with the name __name__
logger = logging.getLogger(__name__)
logger.debug('This is a debug message')
logger.info('This is an info message')
logger.warning('This is a warning message')
```

Let's run the script:

(command-line)

```
$ python3 lab_logging_custom.py
[2020-02-18 21:36:37,471] INFO lab_logging_custom 12 - This is an info message
[2020-02-18 21:36:37,472] WARNING lab_logging_custom 13 - This is a warning message
```

[Python Standard Libraries for Interacting with the OS, Managing Files and Manipulating dates and times]

4.2 os

`os` - "miscellaneous operating system interfaces"

The os module provides dozens of functions for interacting with the operating system. It let's us work on environment variables, files and directories.

- 4.2.1 Importing the `os` module

[2]:
```
import os
```

- 4.2.2 `os.environ` and `os.getenv()`

The `os.environ` value is known as a mapping object that returns a dictionary of the user's environmental variables. You may not know this, but every time you use your computer, some environment variables are set. These can give you valuable information, such as number of processors, type of CPU, the computer name, etc. Let's see what we can find out about our machine:

— 4.2.2.1 Accessing Environment Variables with `os.environ` and `os.getenv()` (REPL)

```
>>> import os
>>> os.environ
environ({'NVM_DIR': '/home/ubuntu/.nvm', 'SSH_CONNECTION': '34.218.119.32 24475 172.31.19.164 2
>>>
```

You can access the environmental variables using your normal dictionary methods. Here's an example:

(REPL)

```
>>> print(os.environ["C9_PROJECT"])
py_scripting
>>>
```

You could also use the `os.getenv` function to access this environmental variable:

(REPL)

```
>>> print(os.getenv("C9_PROJECT"))
py_scripting
>>>
```

The benefit of using `os.getenv()` instead of the `os.environ` dictionary is that if you happen to try to access an environmental variable that doesn't exist, the getenv function will just return None. If you did the same thing with os.environ, you would receive an error. Let's give it a try so you can see what happens:

(REPL)

```
>>> print(os.environ["STAGE"])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/usr/lib/python3.6/os.py", line 669, in __getitem__
    raise KeyError(key) from None
KeyError: 'STAGE'
>>> print(os.getenv("STAGE"))
None
>>>
```

1

- 4.2.3 `os.getcwd()`

Calling os.getcwd() displays the path you're currently in:

[3]:
```
os.getcwd()
```

[3]:
```
'/home/micaela/Projects/apper_ac2e_src/notebooks'
```

- 4.2.4 `os.remove()`

`os.remove()` removes the specified file path. If that path is a directory, it raises an OSError.

Sample usage:

(command-line)

```
>>> import os
>>> print(f"The dir files are: {os.listdir(os.getcwd())}")
The dir files are: ['sample2.txt', 'sample1.txt']
>>> os.remove("sample2.txt")
>>> print(f"The dir files are: {os.listdir(os.getcwd())}")
The dir files are: ['sample1.txt']
>>>
```

There are lots of other functionalities in the os module that are not covered here such as `os.makedir()`, `os.path()`, etc. You can check out the official documentation to see what else you can do.

4.3 shutil and glob

`shutil` - "High-level file operations"

`glob` - "Unix style pathname pattern expansion"

- 4.3.1 `shutil` module

The `shutil` module offers a number of high-level operations on files and collections of files. This module helps in automating the process of copying and removal of files and directories.

— 4.3.1.1 Importing the `shutil` module

[19]:
```
import shutil
```

— **4.3.1.2 `shutil.copy()`** `shutil.copy()` is used to copy the content of source file to destination file or directory.

Sample Usage:

Let's say we have the following directory, subfolders and files. We want to copy the `sample.txt` file from the `source` folder to the folder `destination`.

Before running the script (command-line)

```
$ pwd
/home/micaela/Projects/ac2e_pyscripting/kungfu
$ tree .
.
   destination
   source
       sample.txt

3 directories, 0 files
```

Create a file named `lab_shutil_copy.py` and add the following code (don't forget to change the directory path accordingly):

[ ]:
```python
import os
import shutil

path = '/home/micaela/Projects/ac2e_pyscripting/kungfu/'

# Source path
source  = path + 'source/'

sourcefile  = source + 'sample.txt'

# Destination path
destination = path + 'destination/' #+ 'sample(copy).txt'

# List files and directories in the source folder
print("Before copying:")
print("source:", os.listdir(source))
print("destination:", os.listdir(destination))

# Copy the content of source to destination
dest = shutil.copy(sourcefile, destination)

# List files and directories in the source folder
print("After copying:")
print("source:", os.listdir(source))
print("destination:", os.listdir(destination))

# print path of newly created file
print("Destination path:", dest)
```

Let's run the script:

(command-line)

```
$ python3 lab_shutil_copy.py
Before copying:
source: ['sample.txt']
destination: []
After copying:
source: ['sample.txt']
destination: ['sample.txt']
Destination path: /home/micaela/Projects/ac2e_pyscripting/kungfu/destination/sample.txt
```

After running the script (command-line)

```
tree .
.
```

```
   destination
       sample.txt
   source
        sample.txt

2 directories, 2 files
```

— **4.3.1.3 `shutil.move()`** `shutil.move()` method recursively moves a file or directory (source) to another location (destination) and returns the destination.

Sample Usage:

Let's say we have the following directory, subfolders and files. We want to move the `source` folder including its contents to the folder `destination`.

Before running the script (command-line)

```
$ pwd
/home/micaela/Projects/ac2e_pyscripting/kungfu
$ tree .
.
   destination
   source
        sample.txt

3 directories, 0 files
```

Create a file named `lab_shutil_move.py` and add the following code (don't forget to change the directory path accordingly):

[ ]:
```python
import os
import shutil

path = '/home/micaela/Projects/ac2e_pyscripting/kungfu/'

# List files and directories in the source folder
print("Before moving file:")
print(os.listdir(path))

# Source path
source = path + 'source/'

# Destination path
destination = path + 'destination/'

# Move source to destination
dest = shutil.move(source, destination)
```

```
# List files and directories in the source folder
print("After moving file:")
print(os.listdir(path))

# print path of newly created file
print("Destination path:", dest)
```

Let's run the script:

(command-line)

```
$ python3 lab_shutil.py
Before moving file:
['destination', 'source']
After moving file:
['destination']
Destination path: /home/micaela/Projects/ac2e_pyscripting/kungfu/destination/source
$
```

After running the script (command-line)

```
$ pwd
/home/micaela/Projects/ac2e_pyscripting/kungfu
$ tree .
.
   destination
       source
           sample.txt

3 directories, 0 files
```

- 4.3.2 `glob` module

The `glob` module finds all the pathnames matching a specified pattern according to the rules used by the Unix shell, although results are returned in arbitrary order.

— 4.3.2.1 Importing the `glob` module

[14]:
```
import glob
```

— **4.3.2.2 Wildcard** Wildcard is important glob operator for glob operations. Wildcard or asterisk (*) is used to match zero or more characters.

Let's say we have the following directory and files:

(command-line)

```
$ pwd
/home/micaela/Projects/ac2e_pyscripting/kungfu/source
$ ls -la
total 8
drwxrwxr-x 2 micaela micaela 4096 Feb 20 00:52 .
drwxrwxr-x 4 micaela micaela 4096 Feb 19 16:49 ..
-rw-rw-r-- 1 micaela micaela    0 Feb 20 00:52 sample2.txt
-rw-rw-r-- 1 micaela micaela    0 Feb 20 00:52 sample.csv
-rw-rw-r-- 1 micaela micaela    0 Feb 19 16:38 sample.txt
```

Then we want to match files that have a `.txt` extension:

[6]:
```
import glob
glob.glob("/home/micaela/Projects/ac2e_pyscripting/kungfu/source/*.txt")
```

[6]:
```
['/home/micaela/Projects/ac2e_pyscripting/kungfu/source/sample.txt',
 '/home/micaela/Projects/ac2e_pyscripting/kungfu/source/sample2.txt']
```

4.4 pathlib

- 4.4.1 Importing the main class

[15]:
```
from pathlib import Path
```

- 4.4.2 `Path().name`

`Path().name` returns a string representing the final path component, excluding the drive and root, if any:

[16]:
```
from pathlib import Path
Path("/home/micaela/Projects/ac2e_pyscripting/kungfu/source/sample.txt").name
```

[16]:
```
'sample.txt'
```

[17]:
```
from pathlib import Path
Path("/home/micaela/Projects/ac2e_pyscripting/kungfu/source").name
```

[17]:
```
'source'
```

- 4.4.3 `Path().stem`

`Path().stem` returns the final path component, without its suffix:

[23]:
```
from pathlib import Path
Path("/home/micaela/Projects/ac2e_pyscripting/kungfu/source/sample.txt").stem
```

[23]: 'sample'

[27]:
```
from pathlib import Path
Path("/home/micaela/Projects/ac2e_pyscripting/kungfu/source/sample.tar.gz").stem
```

[27]: 'sample.tar'

[26]:
```
from pathlib import Path
Path("/home/micaela/Projects/ac2e_pyscripting/kungfu/source").stem
```

[26]: 'source'

- 4.4.4 Path().suffix

`Path().suffix` returns the file extension of the final component, if any:

[24]:
```
from pathlib import Path
Path("/home/micaela/Projects/ac2e_pyscripting/kungfu/source/sample.txt").suffix
```

[24]: '.txt'

[29]:
```
from pathlib import Path
Path("/home/micaela/Projects/ac2e_pyscripting/kungfu/source").suffix
```

[29]: ''

- 4.4.5 Path().resolve

`Path().resolve` returns an absolute path, resolving any symlinks.

[51]:
```
from pathlib import Path
p = Path()
print(p)
p.resolve()
```

.

[51]: PosixPath('/home/micaela/Projects/apper_ac2e_src/notebooks')

[52]:
```
from pathlib import Path
p = Path("../")
p.resolve()
```

[52]: PosixPath('/home/micaela/Projects/apper_ac2e_src')

- 4.4.6 Path().joinpath()

Calling this method is equivalent to combining the path with each of the other arguments in turn:

[53]:
```
from pathlib import Path
Path("/home/micaela/Projects/").joinpath("ac2e_pyscripting")
```

[53]:
```
PosixPath('/home/micaela/Projects/ac2e_pyscripting')
```

- 4.4.7 Path().open()

`Path().open()` opens the file pointed to by the path

(REPL)

```
>>> from pathlib import Path
>>> p = Path('/home/micaela/Projects/ac2e_pyscripting/kungfu/source/sample.txt')
>>> with p.open() as f:
...     f.readline()
...
'hey\n'
>>>
```

- 4.4.8 Path().unlink()

`Path().unlink()` removes the file or symbolic link. If the path points to a directory, use `Path.rmdir()` instead.

[47]:
```
import os
from pathlib import Path

print("Before unlinking:")
print(os.listdir("/home/micaela/Projects/ac2e_pyscripting/kungfu/source/"))

Path("/home/micaela/Projects/ac2e_pyscripting/kungfu/source/sample.txt").
 ↪unlink()

print("After unlinking:")
print(os.listdir("/home/micaela/Projects/ac2e_pyscripting/kungfu/source/"))
```

```
Before unlinking:
['sample.txt', 'sample2.txt', 'sample.csv', 'sample.tar.gz']
After unlinking:
['sample2.txt', 'sample.csv', 'sample.tar.gz']
```

4.5 datetime

`datetime` is a module that provides necessary tools whenever you need to work with dates in Python.

- 4.5.1 Importing the `datetime` module

[11]:
```
import datetime
```

- 4.5.2 `datetime.datetime`

The most important and the commonly used object inside the inside the `datetime` module, is the `datetime` class (`datetime.datetime`)

Since both the module and the class have the same name, you need to pay attention to what object you are using.

— **4.5.2.1 `datetime.datetime.now()`** `datetime.datetime.now()` outputs a nice `datetime.datetime` object with the **current date and time in local time zone**. The output is in the following order: 'year', 'month', 'date', 'hour', 'minute', 'seconds', 'microseconds'.

[12]:
```
import datetime

datetime.datetime.now()
```

[12]:
```
datetime.datetime(2020, 3, 5, 16, 45, 42, 48029)
```

—- **4.5.2.2 `datetime.datetime.utcnow()`** `datetime.datetime.utcnow()` outputs a nice `datetime.datetime` object with the **current date and time in UTC**. The output is in the following order: 'year', 'month', 'date', 'hour', 'minute', 'seconds', 'microseconds'.

[13]:
```
import datetime

datetime.datetime.utcnow()
```

[13]:
```
datetime.datetime(2020, 3, 5, 8, 45, 49, 690698)
```

—- **4.5.2.3 Creating a datetime object using `datetime.datetime()`** How do we create a datetime object for a **specific date and time**?

Say for example for the following time: `2013-10-08 08:00:00`

You can pass the values on the same order to `datetime.datetime()`

Syntax: `datetime.datetime(year, month, day, hour, minute, seconds)`

[14]:
```
import datetime

datetime.datetime(2013, 10, 8, 8, 0, 0)
```

[14]:
```
datetime.datetime(2013, 10, 8, 8, 0)
```

- 4.5.3 Formatting a datetime object into any date format using `datetime.strftime()`

You can convert any datetime object to almost any representation of date format using its `strftime()` method.

Pass the symbol representation of the date format as an argument.

Syntax: `<datetime object>.strftime(format)`

[25]:
```
import datetime

dt = datetime.datetime(2013, 10, 8, 8, 0, 0)

print(dt.strftime('%Y-%m-%d %H-%M-%S'))
```

```
2013-10-08 08-00-00
```

- 4.5.4 Converting a string into a datetime object using `datetime.strptime()`

Syntax: `datetime.strptime(date_string, format)`

[16]:
```
from datetime import datetime

datetime_str = '2013-10-08 08:00:00'

datetime_object = datetime.strptime(datetime_str, '%Y-%m-%d %H:%M:%S')

print(type(datetime_object))
print(datetime_object)
```

```
<class 'datetime.datetime'>
2013-10-08 08:00:00
```

- 4.5.5 Getting the difference between 2 dates or times using `datetime.timedelta()`

To create a `datetime.timedelta` class you need to pass a specified duration to the class constructor. The arguments can be in weeks, days (default), hours, minutes, seconds, microseconds.

Syntax: `datetime.timedelta(<duration>)`

[21]:
```
import datetime

td = datetime.timedelta(days=30)
print(td)
```

```
30 days, 0:00:00
```

Let's try to compute the date from 30 days from now:

[22]:
```
print(datetime.date.today() + td)
```

```
2020-04-04
```

- 4.5.6 Creating a date object using `datetime.date()`

Syntax: `datetime.datetime(year, month, day)`

[24]:
```
import datetime

d = datetime.date(2013, 10, 8)
print(d)
```

```
2013-10-08
```

- 4.5.7 Creating a time object using `datetime.time()`

Syntax: `time(hour=0, minute=0, second=0)`

[29]:
```
from datetime import time

t = time()
print(t)

t = time(8, 0, 0)
print(t)

t = time(23, 59, 59)
print(t)
```

```
00:00:00
08:00:00
23:59:59
```

4.6 zipfile

The `zipfile` module can be used to manipulate ZIP archive files.

- 4.6.1 Importing the `zipfile` module

[1]:
```
import zipfile
```

- 4.6.2 Checking valid ZIP files

Let's say we have the following files in a directory:

(command-line)

```
$ pwd
/home/micaela/Projects/ac2e_pyscripting/kungfu
$ ls -la
total 1568
drwxrwxr-x 4 micaela micaela    4096 Mar  6 14:57 .
drwxrwxr-x 5 micaela micaela    4096 Mar  6 14:51 ..
drwxrwxr-x 2 micaela micaela    4096 Feb 19 17:05 destination
-rw-rw-r-- 1 micaela micaela 1584186 Mar  6 14:56 kaizend_branding.zip
-rw-rw-r-- 1 micaela micaela     207 Mar  6 14:57 lab_check_zipfile.py
drwxrwxr-x 2 micaela micaela    4096 Feb 20 03:12 source
```

We can create a script that checks if the files in our directory are valid zip files.

Create a file named `lab_check_zipfile.py` and add the following code:

[ ]:
```
import zipfile

filelist = [
    'destination', 'source', 'lab_check_zipfile.py',
    'kaizend_branding.zip', 'doesnotexist.zip'
]

for filename in filelist:
    print(filename, zipfile.is_zipfile(filename))
```

Let's run the script:

(command-line)

```
$ python lab_check_zipfile.py
destination False
source False
lab_check_zipfile.py False
kaizend_branding.zip True
doesnotexist.zip False
```

Notice that in our script, the `zipfile.is_zipfile(filename)` function returned:

- **False** - for files that aren't valid zip files as well as files that don't exist
- **True** - for valid zip files

- 4.6.3 Reading ZIP files

`zipfile.ZipFile` contains many methods like extract, get info, etc. to work the ZIP files.

Let's try reading the contents of a zip file.

Create a file named `lab_zipfile_read.py` and copy the following code (don't forget to change the directory path accordingly):

[ ]:
```
import zipfile

with zipfile.ZipFile(
    '/home/micaela/Projects/ac2e_pyscripting/kungfu/kaizend_branding.zip') as␣
↪file:

    # ZipFile.infolist() returns a list containing all the members of an␣
↪archive file
    print("\nInfo List")
    print(file.infolist())

    # ZipFile.namelist() returns a list containing all the members with names␣
↪of an archive file
    print("\nName List")
    print(file.namelist())
```

Run the script:

(command-line)

```
$ python lab_zipfile_read.py

Info List
[<ZipInfo filename='kaizend_branding/' filemode='drwxr-xr-x' external_attr=0x10>, <ZipInfo file

Name List
['kaizend_branding/', 'kaizend_branding/Logo and Wordmark/', 'kaizend_branding/Logo and Wordmar
```

- 4.6.4 Extracting ZIP files

Now let's try extracting a zip file.

Create a file named `lab_zipfile_extract.py` and add the following code (don't forget to change the directory path accordingly):

[ ]:
```
import zipfile
import os

with zipfile.ZipFile(
```

```
    '/home/micaela/Projects/ac2e_pyscripting/kungfu/kaizend_branding.zip') as␣
↪file:

    print(f"The dir files are: {os.listdir(os.getcwd())}")

    print("\nExtracting...")
    # ZipFile.extractall(path = filepath, pwd = password) extracts all the␣
↪files to current directory
    file.extractall()

    print(f"The dir files are: {os.listdir(os.getcwd())}")
```

Run the script:

(command-line)

```
python lab_zipfile_extract.py
The dir files are: ['references.txt', 'sample.txt', 'lab_logging_custom.py', 'exercise1.py', '

Extracting...
The dir files are: ['references.txt', 'sample.txt', 'lab_logging_custom.py', 'exercise1.py', '
```

Notice that the script created a new directory `kaizend_branding` after running it. It extracted the contents of file ZIP file `kaizend_branding.zip`

(command-line)

```
$ pwd
/home/micaela/Projects/ac2e_pyscripting
```

(command-line)

```
$ ls -la kaizend_branding/
total 1580
drwxrwxr-x 4 micaela micaela    4096 Mar  6 15:32  .
drwxrwxr-x 6 micaela micaela    4096 Mar  6 15:32  ..
-rw-rw-r-- 1 micaela micaela       0 Mar  6 15:32 'Icon'$'\r'
drwxrwxr-x 4 micaela micaela    4096 Mar  6 15:32 'Logo and Wordmark'
drwxrwxr-x 2 micaela micaela    4096 Mar  6 15:32  Logomark
-rw-rw-r-- 1 micaela micaela  359665 Mar  6 15:32 'PythonPH Kaizend Brand Book.pdf'
-rw-rw-r-- 1 micaela micaela 1041804 Mar  6 15:32 'PythonPH Kaizend CMYK.ai'
-rw-rw-r-- 1 micaela micaela  194712 Mar  6 15:32 'PythonPH Kaizend RGB.ai'
```

- 4.6.5 Creating ZIP files and adding files to a ZIP

To create a new archive, simple instantiate the `ZipFile` with a mode of `'w'`. Any existing file is truncated and a new archive is started. To add files, use the `write()` method. 7

Let's create a new file named `README.txt`

(command-line)

```
$ touch README.txt
```

Then a script named `lab_zipfile_create.py` and add the following code:

[ ]:
```
import zipfile

print('creating archive')
zf = zipfile.ZipFile('sample.zip', mode='w')
try:
    print('adding README.txt')
    zf.write('README.txt')
finally:
    print('closing')
    zf.close()
```

Run the script:

```
$ python lab_zipfile_create.py
creating archive
adding README.txt
closing
```

Notice that this created sa `sample.zip` file and contains the file `README.txt`

(command-line)

```
$ ls -la sample.zip
-rw-rw-r-- 1 micaela micaela 118 Mar  6 16:16 sample.zip
```

4.7 filecmp

The `filecmp` module includes functions and a class for comparing files and directories on the filesystem.

- 4.7.1 Importing the `filecmp` module

[5]:
```
import filecmp
```

- 4.7.2 Comparing Files using `filecmp.cmp()`

We can compare (2) two files on the filesystem using `filecmp.cmp()`. Below is an example:

Let's try it out! Let's say we have a files directory with the following contents:

(command-line)

```
$ pwd
/home/micaela/Projects/ac2e_pyscripting
$ files
```

```
dir1
    doe-a-deer.json
dir2
    doe-a-deer.json
    doe-a-deer.yaml

2 directories, 3 files
```

Create a file named `lab_filecmp.py` and add the following code (don't forget to change the directory path accordingly):

[ ]:
```
import filecmp

print('common_file    :', end=' ')
print(filecmp.cmp('/home/micaela/Projects/ac2e_pyscripting/files/dir1/
 ↪doe-a-deer.json',
                   '/home/micaela/Projects/ac2e_pyscripting/files/dir2/
 ↪doe-a-deer.json',
                   shallow=True),
      end=' ')
print(filecmp.cmp('/home/micaela/Projects/ac2e_pyscripting/files/dir1/
 ↪doe-a-deer.json',
                   '/home/micaela/Projects/ac2e_pyscripting/files/dir2/
 ↪doe-a-deer.json',
                   shallow=False))

print('contents_differ:', end=' ')
print(filecmp.cmp('/home/micaela/Projects/ac2e_pyscripting/files/dir1/
 ↪doe-a-deer.json',
                   '/home/micaela/Projects/ac2e_pyscripting/files/dir2/
 ↪doe-a-deer.yaml',
                   shallow=True),
      end=' ')
print(filecmp.cmp('/home/micaela/Projects/ac2e_pyscripting/files/dir1/
 ↪doe-a-deer.json',
                   '/home/micaela/Projects/ac2e_pyscripting/files/dir2/
 ↪doe-a-deer.yaml',
                   shallow=False))

print('identical      :', end=' ')
print(filecmp.cmp('/home/micaela/Projects/ac2e_pyscripting/files/dir1/
 ↪doe-a-deer.json',
                   '/home/micaela/Projects/ac2e_pyscripting/files/dir1/
 ↪doe-a-deer.json',
                   shallow=True),
      end=' ')
```

```
print(filecmp.cmp('/home/micaela/Projects/ac2e_pyscripting/files/dir1/
 ↪doe-a-deer.json',
                   '/home/micaela/Projects/ac2e_pyscripting/files/dir1/
 ↪doe-a-deer.json',
                   shallow=False))
```

Let's run the script:

```
$ python lab_filecmp.py
common_file    : True True
contents_differ: False False
identical      : True True
```

The `shallow` argument tells `cmp()` whether to look at the contents of the file, in addition to its metadata. The default is to perform a shallow comparison using the information available from `os.stat()`. If the stat results are the same, the files are considered the same. Because the stat output includes the inode on Linux, separate files are not treated as the same even if all of their other metadata (size, creation time, etc.) match. In those cases, the file contents are compared. When shallow is False, the contents of the file are always compared. 6

[Python Standard Library for String Manipulation (RegEx)]

4.8 re

The `re` module is used for string searching and manipulation. It provides regular expression matching operations similar to those found in Perl.

- 4.8.1 Importing the `re` module

[55]:
```
import re
```

- 4.8.2 `re.match()`

`re.match()` searches for some substring in a string and returns a match object if found, else it returns none.

`re.match()` finds something at the ***beginning*** of the string and returns a match object.

Format:

`re.match(<substring or pattern>, string)`

[5]:
```
import re

substring = "you"
```

```
string = """I don't think that you even realize
The joy you make me feel when I'm inside
Your universe
You hold me like I'm the one who's precious
I hate to break it to you but it's just
The other way around
You can thank your stars all you want but
I'll always be the lucky one"""

print(re.match(substring, string))
```

```
None
```

[6]:
```
import re

substring = "you"

string = """I don't think that you even realize
The joy you make me feel when I'm inside
Your universe
You hold me like I'm the one who's precious
I hate to break it to you but it's just
The other way around
You can thank your stars all you want but
I'll always be the lucky one"""

for word in string.split():
    print(re.match(substring, word))
```

```
None
None
None
None
<re.Match object; span=(0, 3), match='you'>
None
None
None
None
<re.Match object; span=(0, 3), match='you'>
None
None
None
None
None
None
None
None
None
None
```

```
None
None
None
None
None
None
None
None
None
None
None
None
None
None
<re.Match object; span=(0, 3), match='you'>
None
None
None
None
None
None
None
None
None
None
None
None
<re.Match object; span=(0, 3), match='you'>
None
None
<re.Match object; span=(0, 3), match='you'>
None
None
None
None
None
None
None
None
```

[7]:
```python
import re

shows = [
    "Stanger Things", "The Crown", "Sabrina",
    "The Witcher", "Orange is the New Black",
    "Black Mirror", "The Umbrella Academy"
]

for show in shows:
```

```
    match = re.match("(T\w+)\W", show)
    print(match)
```

```
None
<re.Match object; span=(0, 4), match='The '>
None
<re.Match object; span=(0, 4), match='The '>
None
None
<re.Match object; span=(0, 4), match='The '>
```

- 4.8.3 re.search()

`re.search()` searches for some substring in a string and returns a match object if found, else it returns none.

`re.search()` finds something ***anywhere*** in the string and returns a match object.

Format:

`re.search(<substring or pattern>, string)`

[8]:
```
import re

substring = "you"

string = """I don't think that you even realize
The joy you make me feel when I'm inside
Your universe
You hold me like I'm the one who's precious
I hate to break it to you but it's just
The other way around
You can thank your stars all you want but
I'll always be the lucky one"""

print(re.search(substring, string))
```

```
<re.Match object; span=(19, 22), match='you'>
```

- 4.8.4 re.compile()

`re.compile()` allows us to combine a regular expression pattern into pattern objects which can be used for pattern matching. It also helps to search a pattern again without rewriting it.

Format:

`re.compile(<substring or pattern>)`

Sample Use:

```
<substring or pattern>.match(string)
<substring or pattern>.search(string)
<substring or pattern>.findall(string)
...
etc.
```

[25]:
```
import re

pattern = re.compile("y\w+")

string = """I don't think that you even realize
The joy you make me feel when I'm inside
Your universe
You hold me like I'm the one who's precious
I hate to break it to you but it's just
The other way around
You can thank your stars all you want but
I'll always be the lucky one"""

pattern.match(string)
```

[27]:
```
import re

pattern = re.compile("y\w+")

string = """I don't think that you even realize
The joy you make me feel when I'm inside
Your universe
You hold me like I'm the one who's precious
I hate to break it to you but it's just
The other way around
You can thank your stars all you want but
I'll always be the lucky one"""

pattern.search(string)
```

[27]:
```
<re.Match object; span=(19, 22), match='you'>
```

[28]:
```
import re

pattern = re.compile("y\w+")

string = """I don't think that you even realize
The joy you make me feel when I'm inside
Your universe
You hold me like I'm the one who's precious
I hate to break it to you but it's just
The other way around
```

```
You can thank your stars all you want but
I'll always be the lucky one"""

pattern.findall(string)
```

[28]: ['you', 'you', 'you', 'your', 'you', 'ys']

---

1 https://www.toptal.com/python/in-depth-python-logging

2 https://realpython.com/python-logging

3 https://docs.python.org/3/howto/logging.html

4 https://python101.pythonlibrary.org/chapter16_os.html#chapter-16-the-os-module

5 https://docs.python.org/3/tutorial/stdlib.html

6 https://pymotw.com/3/filecmp/#comparing-files

7 https://pymotw.com/2/zipfile/#creating-new-archives