



# PROGRAMMING IN PYTHON

Gavrilit Dragos  
Course 9

# TIME MODULE

Implements function that allows one to work with time:

- Get current time
- Format time
- Sleep
- Time zone information

Details about **time** module in Python:

- Python 3: <https://docs.python.org/3/library/time.html#module-time>

# TIME MODULE

## Usage:

Python 3.x

```
import time
w = ["Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun"]
print ("Time in seconds:",time.time())
print ("Today          :",time.ctime())
tmobj = time.localtime()
print ("Year           :",tmobj.tm_year)
print ("Month          :",tmobj.tm_mon)
print ("Day            :",tmobj.tm_mday)
print ("Day of week      :",w[tmobj.tm_wday])
print ("Day from year    :",tmobj.tm_yday)
print ("Hour            :",tmobj.tm_hour)
print ("Min             :",tmobj.tm_min)
print ("Sec             :",tmobj.tm_sec)
```

## Output

```
Time in seconds: 1604739661.403554
Today          : Sat Nov  7 11:01:01 2020
Year           : 2020
Month          : 11
Day            : 7
Day of week     : Sat
Day from year   : 312
Hour           : 11
Min            : 1
Sec            : 1
```

# TIME MODULE

Both **localtime** and **gmtime** have one parameter (the number of seconds from 1970). If this parameter is provided the time object will be the time computed based on that number. Otherwise the time object will be the time based on `time.time ()` (current time) value.

Python 3.x	Output
<code>import time</code>	
<code>print (time.localtime())</code>	<code>time.struct_time(tm_year=2019, tm_mon=11, tm_mday=10, tm_hour=14, tm_min=30, tm_sec=34, tm_wday=6, tm_yday=314, tm_isdst=0)</code>
<code>print (time.gmtime())</code>	<code>time.struct_time(tm_year=2019, tm_mon=11, tm_mday=10, tm_hour=12, tm_min=30, tm_sec=34, tm_wday=6, tm_yday=314, tm_isdst=0)</code>
<code>print (time.gmtime(100))</code>	<code>time.struct_time(tm_year=1970, tm_mon=1, tm_mday=1, tm_hour=0, tm_min=1, tm_sec=40, tm_wday=3, tm_yday=1, tm_isdst=0)</code>
<code>print (time.gmtime(time.time()))</code>	<code>time.struct_time(tm_year=2019, tm_mon=11, tm_mday=10, tm_hour=12, tm_min=30, tm_sec=34, tm_wday=6, tm_yday=314, tm_isdst=0)</code>

# TIME MODULE

Use **mktime** to convert from a time object struct to a float number.

Use **asctime** member to convert from a time object to a readable representation of the time (string format).

Python 3.x

```
import time

t = time.time()
tobj = time.localtime()
tm = time.mktime(tobj)
print (tm)
print (t)
print (time.asctime(tobj))
```

**Output**

```
1604739710.0
1604739710.098777
Sat Nov  7 11:01:50 2020
```

# TIME MODULE

Use **strftime** to time object to a specified string representation:

Abbreviation	Description	Abbreviation	Description
%H	Hour in 24 hour format	%M	Minute
%I	Hour in 12 hour format	%S	Seconds
%Y	Year (4 digits)	%A	Day of week (name)
%m	Month (decimal)	%d	Day of month (decimal)
%B	Month (name)	%p	AM or PM

Python 3.x

```
import time
tobj = time.localtime()
print (time.strftime("%H:%M:%S - %Y-%m-%d",tobj))
print (time.strftime("%I%p:%M:%S - %B",tobj))
print (time.strftime("%B,%A %d %Y",tobj))
```

## Output

```
14:17:25 - 2019-11-10
02PM:17:25 - November
November,Sunday 10 2019
```

# TIME MODULE

**strftime** if used without a time object applies the string format to the current time.

Time module also has a function **sleep** that receives one parameter (the number of seconds the current script has to wait until it continues its execution).

Python 3.x

```
import time

for i in range(0,8):
    print (time.strftime("%H:%M:%S"))
    time.sleep(2) #sleep 2 seconds
```

If no time object structure is provided, **strftime** function will use current time to apply the specified format.

## Output

```
11:46:15
11:46:17
11:46:19
11:46:21
11:46:23
11:46:25
11:46:27
11:46:29
```

# HASHLIB MODULE

Implements function that allows one to compute different cryptographic functions:

- MD5
- SHA-1
- SHA-224
- SHA-384
- SHA-512
- Shake-128
- Shake-256

Details about **hashlib** module in Python:

- Python 3: <https://docs.python.org/3/library/hashlib.html>



# HASHLIB MODULE

Each hashlib object has an **update** function (to update the value of the hash) and a **digest** or **hexdigest** function(s) to compute the final hash.

Python 3.x

```
import hashlib

m = hashlib.md5()
m.update(b"Today")
m.update(b" I'm having")
m.update(b" a Python ")
m.update(b"course")
print (m.hexdigest())
```

Output

```
9dea650a4eab481ec0f4b5ba28e3e0b8
```

```
import hashlib

print (hashlib.md5(b"Today I'm having a Python course").hexdigest())
```

# HASHLIB MODULE

Each hashlib object has an **update** function (to update the value of the hash) and a **digest** or **hexdigest** function(s) to compute the final hash.

Python 3.x

```
import hashlib

m = hashlib.md5()
m.update(b'Today ')
m.update(b" I'm having")
m.update(b" a Python ")
m.update(b"course")
print (m.hexdigest())
```

The **<b>** prefix in front of a string is ignored in Python 2. In Python 3 means that the string is a byte list. **update** method requires a list of bytes (not a string). However, in Python 2 it can be used without the prefix **<b>**

```
import hashlib

print (hashlib.md5(b"Today I'm having a Python course").hexdigest())
```

# HASHLIB MODULE

Hashes are often use on files (to associate the content of a file to a specific hash).

Python 3.x

```
import hashlib

def GetFileSHA1(filePath):
    m = hashlib.sha1()
    m.update(open(filePath, "rb").read())
    return m.hexdigest()

print (GetFileSHA1("< a file path >"))
```

Output

cad7a796be26149218a76661d316685d7de2d56d

While this example is ok, keep in mind that it loads the entire file content in memory !!!

# HASHLIB MODULE

The correct way to do this (having a support for large files is as follows):

Python 3.x

```
import hashlib
def GetFileSHA1(filePath):
    try:
        m = hashlib.sha1()
        f = open(filePath, "rb")
        while True:
            data = f.read(4096)
            if len(data)==0: break
            m.update(data)
        f.close()
        return m.hexdigest()
    except:
        return ""
```

# DATA SERIALIZATION

Python has several implementations for data serialization.

- JSON
- Pickle
- Marshal

Documentation for JSON:

- Python 3: <https://docs.python.org/3/library/json.html>

Documentation for PICKLE :

- Python 3: <https://docs.python.org/3/library/pickle.html>

Documentation for Marshal :

- Python 3: <https://docs.python.org/3/library/marshal.html#module-marshal>

# JSON MODULE

## JSON functions:

- **json.dump** (obj, fp, skipkeys=False, ensure\_ascii=True, check\_circular=True, allow\_nan=True, cls=None, indent=None, separators=None, default=None, sort\_keys=False, \*\*kw)
- **json.dumps**(obj, skipkeys=False, ensure\_ascii=True, check\_circular=True, allow\_nan=True, cls=None, indent=None, separators=None, default=None, sort\_keys=False, \*\*kw) → to obtain the string representation of the obj in JSON format
- **json.load**(fp, cls=None, object\_hook=None, parse\_float=None, parse\_int=None, parse\_constant=None, object\_pairs\_hook=None, \*\*kw)
- **json.loads**(s, encoding=None, cls=None, object\_hook=None, parse\_float=None, parse\_int=None, parse\_constant=None, object\_pairs\_hook=None, \*\*kw)

# JSON MODULE

Usage (serialization):

Python 3.x

```
import json
```

```
d = { "a": [1, 2, 3],  
      "b": 100,  
      "c": True  
}
```

```
s = json.dumps(d)
```

```
open("serialization.json", "wt").write(s)
```

```
print (s)
```

## serialization.json

FileAddr	000	001	002	003	004	005	006	007	Text
000000000	123	034	099	034	058	032	116	114	{"c": true, "a": [1, 2, 3], "b": 100}
000000008	117	101	044	032	034	097	034	058	
000000016	032	091	049	044	032	050	044	032	
000000024	051	093	044	032	034	098	034	058	
000000032	032	049	048	048	125				

## Output

```
{"a": [1, 2, 3], "b": 100, "c": true}
```

# JSON MODULE

Usage (de-serialization):

Python 3.x

```
import json
```

```
data = open("serialization.json", "rt").read()  
d = json.loads(data)  
print (d)
```

```
import json
```

```
d = json.load(open("serialization.json", "rt"))  
print (d)
```

**Output**

```
{"a": [1, 2, 3], "b": 100, "c": true}
```



# PICKLE MODULE

Pickle is another way to serialize objects in Python. The serialization is done in a binary mode.

Pickle can also serialize:

- Functions (defined using **def** and not lambda)
- Classes
- Functions from modules

Documentation for PICKLE :

- Python 3: <https://docs.python.org/3/library/pickle.html>

# PICKLE MODULE

PICKLE functions:

- `pickle.dump` (obj, file, *protocol=None, \*, fix\_imports=True*)
- `pickle.dumps`(obj, *protocol=None, \*, fix\_imports=True*) → to obtain the buffer representation of the obj in pickle format
- `pickle.load`(file, *\*, fix\_imports=True, encoding="ASCII", errors="strict"*)
- `pickle.loads`(byte\_object, *\*, fix\_imports=True, encoding="ASCII", errors="strict"*)

PICKLE support multiple version. Be careful when you serialize with Python 2 and try to de-serialize with Python 3 (not all version supported by Python 3 are also supported by Python 2).

If you are planning to switch between versions, either check `pickle.HIGHEST_PROTOCOL` to see if the highest protocol is compatible or use **0** as the protocol value.

# PICKLE MODULE

Python 3.x

```
import pickle
```

```
d = {  
    "a": [1, 2, 3],  
    "b": 100,  
    "c": True  
}
```

## serialization.pickle

FileAddr	000	001	002	003	004	005	006	007	Text
000000000	128	003	125	113	000	040	088	001	Ç♥}q (X☺
000000008	000	000	000	099	113	001	136	088	cq☺êX
000000016	001	000	000	000	097	113	002	093	☺ aq☺]
000000024	113	003	040	075	001	075	002	075	q♥(K☺K☺K
000000032	003	101	088	001	000	000	000	098	♥eX☺ b
000000040	113	004	075	100	117	046			q♦Kdu.

```
buffer = pickle.dumps(d) #buffer = pickle.dumps(d, 0) (safety)
```

```
open("serialization.pickle", "wb").write(buffer)
```

Pickle need a file to be open in binary mode !

# PICKLE MODULE

Usage (de-serialization):

Python 3.x

```
import pickle
```

```
data = open("serialization.pickle", "rb").read()  
d = pickle.loads(data)  
print (d)
```

```
import pickle
```

```
d = pickle.load(open("serialization.pickle", "rb"))  
print (d)
```

**Output**

```
{"a": [1, 2, 3], "b": 100, "c": true}
```

# MARSHAL MODULE

Marshal is another way to serialize objects in Python. The serialization is done in a binary mode. Designed for python compiled code (pyc). The binary result is platform-dependent !!!

Marshal functions:

- **marshal.dump** (value, file, [version]) marshal
- **marshal.dumps**(value, [version]) ➔ to obtain the binary representation of the obj in marshal format
- **marshal.load**(file)
- **marshal.loads**(string/buffer)

Documentation for Marshal :

- Python 3: <https://docs.python.org/3/library/marshal.html#module-marshal>

# MARSHAL MODULE

Usage (serialization):

Python 3.x

```
import marshal
```

```
d = {  
    "a": [1, 2, 3],  
    "b": 100,  
    "c": True  
}
```

```
buffer = marshal.dumps(d)  
open("serialization.marshal", "wb").write(buffer)
```

## serialization.marshal

FileAddr	000	001	002	003	004	005	006	007	Text
000000000	251	218	001	099	084	218	001	097	√ √ ☹ c T √ ☹ a
000000008	091	003	000	000	000	233	001	000	[ ♥      ☹ ☹
000000016	000	000	233	002	000	000	000	233	☹ ☹      ☹
000000024	003	000	000	000	218	001	098	233	♥      √ ☹ b ☹
000000032	100	000	000	000	048				d    0

# MARSHAL MODULE

Usage (de-serialization):

Python 3.x

```
import marshal
```

```
data = open("serialization.marshal","rb").read()  
d = marshal.loads(data)  
print (d)
```

```
import marshal
```

```
d = marshal.load(open("serialization.marshal","rb"))  
print (d)
```

Marshal serialization has a different format in Python 2 and Python 3 (these two are not compatible).

**Output**

```
{"a": [1, 2, 3], "b": 100, "c": true}
```

# RANDOM MODULE

Implements different random base functions:

- `random.random()` → a random float number between 0 and 1
- `random.randint(min,max)` → a random integer number between [min ... max]
- `random.choice(list)` → selects a random element from a list
- `random.shuffle(list)` → shuffles the list
- `random.sample(list,count)` → creates another list from the current one containing **count** elements

Details about **random** module in Python:

- Python 3: <https://docs.python.org/3/library/random.html>



# RANDOM MODULE

## Usage:

Python 3.x

```
import random

print (random.random())
print (random.randint(5,10))

l = [2,3,5,7,11,13,17,19]
print (random.choice(l))
print (random.sample(l,3))

random.shuffle(l)
print (l)
```

### Output

```
0.9410874890940395
9
5
[19, 17, 11]
[13, 17, 11, 5, 2, 19, 7, 3]
```

# ZIPFILE MODULE

Implements different functions to work with a zip archive:

- List all elements from a zip archive
- Extract files
- Add files to archive
- Get file information
- etc

Details about **zipfile** module in Python:

- Python 3: <https://docs.python.org/3/library/zipfile.html>

# ZIPFILE MODULE

Listing the content of a zip archive:

Python 3.x

```
import zipfile

z = zipfile.ZipFile("archive.zip")
for i in z.infolist():
    print (i.filename,
           i.file_size,
           i.compress_size)

z.close()
```

## Output

```
MathOps/ 0 0
MathOps/Complex/ 0 0
MathOps/Complex/Series.py 117 79
MathOps/Complex/__init__.py 38 38
MathOps/Simple/ 0 0
MathOps/Simple/Arithmetic.py 54 52
MathOps/Simple/Bits.py 60 55
MathOps/Simple/__init__.py 87 84
MathOps/__init__.py 30 30
a.py 43 43
all.csv 62330588 8176706
```

# ZIPFILE MODULE

To extract a file from an archive:

Python 3.x

```
import zipfile
z = zipfile.ZipFile("archive.zip")
z.extract("MathOps/Simple/Arithmetic.py", "MyFolder")
z.close()
```

Arithmetic.py will be extracted to “MyFolder/MathOps/Simple/Arithmetic.py”

To extract all files:

Python 3.x

```
import zipfile
z = zipfile.ZipFile("archive.zip")
z.extractall("MyFolder")
z.close()
```

# ZIPFILE MODULE

A file can also be opened directly from an archive. This is usually required if one wants to extract the content somewhere else or if the content needs to be analyzed in memory.

Python 3.x

```
import zipfile

z = zipfile.ZipFile("archive.zip")
f = z.open("MathOps/Simple/Arithmetic.py")
data = f.read()
f.close()
open("my_ar.py", "wb").write(data)
z.close()
```

Method **open** from zipfile returns a file-like object. You can also specify a password:  
Format: ZipFile.**open**(name, *mode='r', pwd=None*)

# ZIPFILE MODULE

The following script creates a zip archive and add files to it:

Python 3.x

```
import zipfile

z = zipfile.ZipFile("new_archive.zip", "w", zipfile.ZIP_DEFLATED)
z.writestr("test.txt", "some texts ...")
z.write("serialization.json")
z.write("serialization.json", "/dir/a.json")
z.writestr("/dir/a.txt", "another text ...")
z.close()
```

**writestr** method writes the content of a string into a zip file.

**write** methods add a file to the archive.

When creating an archive one can specify a desire compression: ZIP\_DEFLATED, ZIP\_STORED, ZIP\_BZIP2 or ZIP\_LZMA.