Useful Modules

In this chapter, several useful modules are introduced. These modules include the following:

- Modules for date & time manipulation
  - time
  - datetime
- Regular expressions
  - re
- Operating system interfaces
  - os
  - os.path
- Unix style pathname pattern expansion
  - glob
- Python object serialization
  - pickle
- Python object persistence
  - shelve
- JSON encoder and decoder
  - json
- Reading and writing CSV files
  - csv

---

time and datetime

- The time module provides many time-related functions, which often call platform C library functions with the same names.
  - The epoch is the point where the time starts, and is dependent on platforms. For Unix, the epoch is January 1, 1970, 00:00:00 (UTC). We can look at time.gmtime(0) to find out the epoch on our platforms.
  - The coordinated universal time (UTC) is the primary time standard by which the world regulates clocks and times. https://en.wikipedia.org/wiki/Coordinated_Universal_Time (https://en.wikipedia.org/wiki/Coordinated_Universal_Time)
- The datetime module provides many functions to convert between several time formats and to calculate time with consideration of leap years and time zones.

---

time Module

In [1]:

```
# coding=Big5
import time
time.gmtime(0)
```

Out[1]:

```
time.struct_time(tm_year=1970, tm_mon=1, tm_mday=1, tm_hour=0, tm_min=0, tm_
sec=0, tm_wday=3, tm_yday=1, tm_isdst=0)
```

- time.struct_time https://docs.python.org/2/library/time.html (https://docs.python.org/2/library/time.html)

Attribute	Value
tm_year	(for example, 1993)
tm_mon	range [1, 12]
tm_mday	range [1,31]
tm_hour	range [0,23]
tm_min	range [0,59]
tm_sec	range [0,61]
tm_wday	range [0, 6], Monday is 0
tm_yday	range [1, 366]
tm_isdst	0, 1 or -1

In calls to mktime(), tm_isdst may be set to 1 when daylight savings time is in effect, and 0 when it is not. A value of -1 indicates that this is not known, and will usually result in the correct state being filled in.

In [2]:

```
globalt = time.gmtime()
localt = time.localtime()
print(globalt)
print(localt)
```

```
time.struct_time(tm_year=2018, tm_mon=9, tm_mday=5, tm_hour=2, tm_min=20, tm
_sec=29, tm_wday=2, tm_yday=248, tm_isdst=0)
time.struct_time(tm_year=2018, tm_mon=9, tm_mday=5, tm_hour=10, tm_min=20, t
m_sec=29, tm_wday=2, tm_yday=248, tm_isdst=0)
```

- **altzone**: the offset of the local DST timezone;
- **asctime([t])**: convert a tuple or struct_time to a string of the form 'Sun Jun 20 23:21:05 1993';
- **clock()**: on windows, clock() returns seconds since the first call to this function; on unix, clock() returns the current process time (seconds)
- **ctime([secs])**: convert a time (seconds) since the epoch to a string representing local time; (ctime(sec) is equivalent to asctime(localtime(sec)));
- daylight,
- **gmtime([secs])**: convert a time (seconds) since the epoch to a struct_time in UTC;
- **localtime([secs])**: like gmtime but convert to the local time.
- **mktime**: the inverse function of localtime()
- **monotonic()**: return the value (in fractional seconds) of a monotonic clock;
- **perf_counter()**: return the value (in fractional seconds) of a performance counter.

- **process_time()**: return the sum of the system and user CPU time of the current process;
- **sleep(secs)**: suspend the calling thread for the given number of seconds;
- **strftime(format[,t])**: convert a tuple or struct_time representing a time to a string according to the format;
- **strptime(string[,format])**: convert a string representing a time to struct_time
- **time()**: return the time (second) since the epoch;
- **get_clock_info(clock_name)**: Get information on the specified clock as a namespace object. Supported clock names and the corresponding functions to read their value are:
  - 'clock': time.clock()
  - 'monotonic': time.monotonic()
  - 'perf_counter': time.perf_counter()
  - 'process_time': time.process_time()
  - 'time': time.time()

In [21]:

```
print(time.get_clock_info('clock'))
print(time.get_clock_info('monotonic'))
print(time.get_clock_info('perf_counter'))
print(time.get_clock_info('process_time'))
print(time.get_clock_info('time'))
```

```
namespace(adjustable=False, implementation='QueryPerformanceCounter()', mono
tonic=True, resolution=2.7705650733701043e-07)
namespace(adjustable=False, implementation='GetTickCount64()', monotonic=Tru
e, resolution=0.015625)
namespace(adjustable=False, implementation='QueryPerformanceCounter()', mono
tonic=True, resolution=2.7705650733701043e-07)
namespace(adjustable=False, implementation='GetProcessTimes()', monotonic=Tr
ue, resolution=1e-07)
namespace(adjustable=True, implementation='GetSystemTimeAsFileTime()', monot
onic=False, resolution=0.015625)
```

The following functions in the time module can convert between time representations.

From	To	Function
seconds since the epoch	struct_time in UTC	time.gmtime()
seconds since the epoch	struct_time in local time	time.localtime()
struct_time in UTC	seconds since the epoch	calendar.timegm()
struct_time in local time	seconds since the epoch	time.mktime()

The following functions can convert between times and strings.

From	To	Function
string	struct_time	time.strptime()
struct_time	string	time.asctime(), time.strftime()
seconds since the epoch	string	time.ctime([sec]), time.strftime(format,localtime([sec]))

The directives can be used in the format string for strftime and strptime.

Directive	Meaning
%a	Locale's abbreviated weekday name.
%A	Locale's full weekday name.
%b	Locale's abbreviated month name.
%B	Locale's full month name.
%c	Locale's appropriate date and time representation.
%d	Day of the month as a decimal number [01,31].
%H	Hour (24-hour clock) as a decimal number [00,23].
%I	Hour (12-hour clock) as a decimal number [01,12].
%j	Day of the year as a decimal number [001,366].
%m	Month as a decimal number [01,12].
%M	Minute as a decimal number [00,59].
%p	Locale's equivalent of either AM or PM.
%S	Second as a decimal number [00,61].
%U	Week number of the year (Sunday as the first day of the week) as a decimal number [00,53]. All days in a new year preceding the first Sunday are considered to be in week 0.
%w	Weekday as a decimal number [0(Sunday),6].
%W	Week number of the year (Monday as the first day of the week) as a decimal number [00,53]. All days in a new year preceding the first Monday are considered to be in week 0.
%x	Locale's appropriate date representation.
%X	Locale's appropriate time representation.
%y	Year without century as a decimal number [00,99].
%Y	Year with century as a decimal number.
%z	Time zone offset indicating a positive or negative time difference from UTC/GMT of the form +HHMM or -HHMM, where H represents decimal hour digits and M represents decimal minute digits [-23:59, +23:59].
%Z	Time zone name (no characters if no time zone exists).
%%	A literal '%' character.

In [22]:

```
tmstr = time.strftime("%a, %d %b %Y %H:%M:%S +0000", time.gmtime())
time.strptime(tmstr,"%a, %d %b %Y %H:%M:%S +0000")
```

Out[22]:

```
time.struct_time(tm_year=2018, tm_mon=9, tm_mday=5, tm_hour=2, tm_min=46, tm
_sec=1, tm_wday=2, tm_yday=248, tm_isdst=-1)
```

---

datetime Module

https://docs.python.org/3/library/datetime.html (https://docs.python.org/3/library/datetime.html)

The datetime module includes several types which are usually used to store information about dates and times:

- datetime.date(year,month,day) is used for creating dates which have no time fields;

```
x = datetime.date.today(); # x has three data fields: x.year, x.month, x.day
```

- datetime.time(hour,minute,second) is used for creating times which are independent of a date;

```
x = datetime.time (10,30,0); # x has four data fields: x.hour, x.minute, x.se
cond, x.micorsecond
```

- datetime.datetime(year,month,day, hour,minute,second) is used to create datetime objects which have both dates and times;

```
x = datetime.datetime.today(); x = datetime.datetime.now(); x = datatime.date
time.utcnow();
```

- datetime.timedelta(days=0, seconds=0, microseconds=0, milliseconds=0, minutes=0, hours=0, weeks=0) is used to store differences between two dates or times;

```
d = datetime.timedelta(seconds=5)
```

In [30]:

```
import datetime
print(datetime.date.today())
print(datetime.time(10,30,0))
print(datetime.datetime.today(), datetime.datetime.now(), datetime.datetime.utcnow(), sep='
```

```
2018-09-05
10:30:00
2018-09-05 10:58:33.703435
2018-09-05 10:58:33.703435
2018-09-05 02:58:33.703435
```

The datetime module supports date and time arithmetic: datetime.timedelta

- Instance attributes (read-only)
  - days, seconds, microseconds
  - Only the three variables are stored internally and normalized so that the representation is unique.

    ```
    0 <= microseconds < 1000000
    0 <= seconds < 3600*24 (the number of seconds in one day)
    -999999999 <= days <= 999999999
    ```

- Class attributes
  - timedelta.min
  - timedelta.max
  - timedelta.resolution: the smallest possible difference between two different timedelta objects
- Instance methods
  - total_seconds() : return the total number of seconds

In [26]:

```
import datetime

print(datetime.date(2052,2,23)+datetime.timedelta(days=6))

print(datetime.date(2052,2,23)-datetime.date(2020,2,23))
```

```
2052-02-29
11688 days, 0:00:00
```

time arithmetic

Support Operation	Result
t1 = t2 + t3	Sum of t2 and t3. Afterwards t1-t2 == t3 and t1-t3== t2 are true.
t1 = t2 - t3	Difference of t2 and t3. Afterwards t1 == t2 - t3 and t2 == t1 + t3 are true.
t1 = t2 * i or t1 = i * t2	Delta multiplied by an integer. Afterwards t1 // i == t2 is true, provided i != 0. In general, t1 * i == t1 * (i-1) + t1 is true.
t1 = t2 * f or t1 = f * t2	Delta multiplied by a float. The result is rounded to the nearest multiple of timedelta.resolution using round-half-to-even.

Support Operation	Result
f = t2 / t3	Division of t2 by t3. Returns a float object.
t1 = t2 / f or t1 = t2 / i	Delta divided by a float or an int. The result is rounded to the nearest multiple of timedelta.resolution using round-half-to-even.
t1 = t2 // i or t1 = t2 // t3	The floor is computed and the remainder (if any) is thrown away. In the second case, an integer is returned.
t1 = t2 % t3	The remainder is computed as a timedeltaobject.
q, r = divmod(t1, t2)	Computes the quotient and the remainder: q = t1// t2 and r = t1 % t2. q is an integer and r is a timedelta object.
+t1	Returns a timedelta object with the same value.
-t1	Equivalent to timedelta(-t1.days, -t1.seconds, -t1.microseconds), and to t1* -1.
abs(t)	Equivalent to +t when t.days >= 0, and to -twhen t.days < 0.
str(t)	Returns a string in the form [D day[s],][H]H:MM:SS[.UUUUUU], where D is negative for negative t.
repr(t)	Returns a string in the formdatetime.timedelta(D[, S[, U]]), where D is negative for negative t.

In [1]:

```
from datetime import timedelta

year = timedelta(days=365)
another_year = timedelta(weeks=40, days=84, hours=23, minutes=50, seconds=600)  # adds up t

print(year.total_seconds())
print(year == another_year)

ten_years = 10 * year
print(ten_years, ten_years.days // 365)

nine_years = ten_years - year
print(nine_years, nine_years.days // 365)

three_years = nine_years // 3
print(three_years, three_years.days // 365)

print(abs(three_years - ten_years) == 2 * three_years + year)
```

```
31536000.0
True
3650 days, 0:00:00 10
3285 days, 0:00:00 9
1095 days, 0:00:00 3
True
```

Class & Instance methods of datetime.date

- Useful class methods
  - date.today(): equivalent to date.fromtimestamp(time.time());
  - date.fromtimestamp(timestamp)

```
Use the class method fromtimestamp(timestamp) to convert timestamp to datet
ime.
x=datetime.datetime.fromtimestamp(time.time())
```

  - date.fromordinal(ordinal): return the date corresponding to the proleptic Gregorian ordinal.
- Useful instance methods:

```
timetuple(), timestamp()
```

- Use the instance method timetuple() to convert to struct_time.

In [3]:

```
import datetime
x=datetime.date(1,2,3)
print(x.timetuple())
```

```
time.struct_time(tm_year=1, tm_mon=2, tm_mday=3, tm_hour=0, tm_min=0, tm_sec
=0, tm_wday=5, tm_yday=34, tm_isdst=-1)
```

Supported Operation	Result
date2 = date1 + timedelta	date2 is timedelta.days days removed from date1.
date2 = date1 - timedelta	Computes date2 such that date2 + timedelta ==date1.
timedelta = date1 - date2	
date1 < date2	date1 is considered less than date2 when date1precedes date2 in time.

A useful example:

In [16]:

```
import datetime
def time_to_birthday(m, d):
    today = datetime.date.today();
    birthday = datetime.date(today.year,m,d)
    if birthday < today:
        birthday = datetime.date(today.year+1,m,d)
    df = abs(birthday-today)
    return df.days

print(time_to_birthday(1,1))

print(time_to_birthday(datetime.date.today().month,datetime.date.today().day))

d = datetime.date.today() - datetime.timedelta(days=1)
print(time_to_birthday(d.month,d.day))

print(time_to_birthday(12,1))
```

```
118
0
364
87
```

Class & Instance methods of datetime.datetime

- Class methods
  - today(), now(), utcnow()
  - fromtimestamp(timestamp): return a datetime object corresponding to timestamp
  - combine(date,time): return a new datetime object which combines the date and time objects
  - strptime(time_string, format): return a datetime object corresponding to time_string
- Instance attributes (read-only)
  - year, day, month, hour, minute, second, microsecond
  - strftime(format)
- Instance methods
  - ctime()
  - strftime(format)
  - date(): return the date object
  - time(): return the time object
  - replace(year=self.year, month=self.month, day=self.day, hour=self.hour, minute=self.minute, second=self.second, microsecond=self.microsecond, tzinfo=self.tzinfo, * fold=0): return a datetime whose attributes specified by the keyword arguments are given with new values;
  - timestamp(): return the corresponding timestamp
  - timetuple(): return the corresponding struct_time
  - isoformat(sep='T', timespec='auto'): return a string representing the date and time in ISO 8601 format, YYYY-MM-DDTHH:MM:SS.mmmmmm or, if microsecond is 0, YYYY-MM-DDTHH:MM:SS

Support Operation
datetime2 = datetime1 + timedelta
datetime2 = datetime1 - timedelta
timedelta = datetime1 - datetime2
datetime1 < datetime2

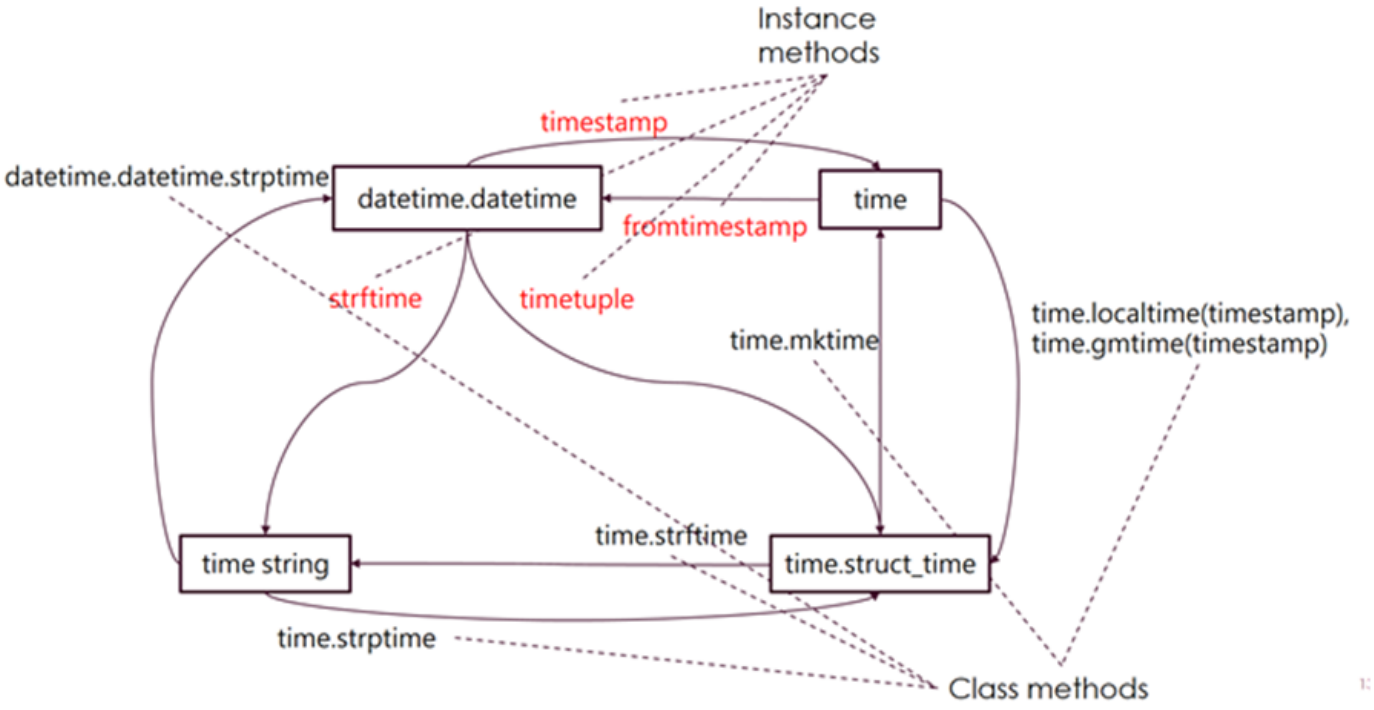
In [18]:

```
import datetime, time

z = datetime.datetime.now()
print(z.timestamp())

print(time.mktime(z.timetuple()))
```

```
1536154091.428002
1536154091.0
```



re Module

https://docs.python.org/3/library/re.html#module-re (https://docs.python.org/3/library/re.html#module-re)

- Regular expressions provide a formal way to express string search patterns.
- This module provides regular expression matching operations.
  - re.split(pattern, string , flags=0): split string by the occurrence of pattern;
  - re.search(pattern, string , flags=0): search for the first occurrence of pattern in string;

```
m=re.search(pattern,string)
if m:
    process(m)
```

  - re.match(pattern, string , flags=0): determine if zero or more characters at the beginning of string matches pattern;
  - re.findall(pattern, string , flags=0): find all non-overlapping matches of pattern in string as a list of strings;
  - re.finditer(pattern, string , flags=0): return an iterator iterating over all non-overlapping matches of pattern in string;
  - re.sub(pattern, repl, string, count=0, flags=0): return the string obtained by replacing the non-overlapping occurrences of pattern in string by repl, which can be a string or a function;
  - re.compile(pattern,flags): compile a regular expression pattern into a regular expression object;

```
import re
reg = re.compile(pattern)
reg.XXXX(string)
```

  - re.escape(string): escape all characters in string except ASCII letters, digits, and '_'.

flags

Flags	Result
re.A, re.ASCII	Make \w, \W, \b, \B, \d, \D, \s and \S perform ASCII-only matching instead of full Unicode matching.
re.I, re.IGNORECASE	Perform case-insensitive matching; expressions like [A-Z] will match lowercase letters, too.
re.M, re.MULTILINE	When specified, the pattern character '^' matches at the beginning of the string and at the beginning of each line (immediately following each newline); and the pattern character '$' matches at the end of the string and at the end of each line (immediately preceding each newline). By default, '^' matches only at the beginning of the string, and '$' only at the end of the string and immediately before the newline (if any) at the end of the string.
re.S, re.DOTALL	Make the '.' special character match any character at all, including a newline; without this flag, '.' will match anything except a newline.

In [1]:

```
import re
re.split(r",", "123,456,789")    # r=>Tell Python that this is a raw string, and Python doe
```

Out[1]:

```
['123', '456', '789']
```

Searching a specific string pattern in string by re is simple.

In [3]:

```
match = re.search(r"Orz", "One Orz, Two Orzs") # Orz is the specific search pattern.
if match:
    print(match.group(),match.start(),match.end())
```

```
Orz 4 7
```

In [4]:

```
reg = re.compile(r"Orz")
match = reg.search("One Orz, Two Orzs") # Orz is the specific search pattern.
if match:
    print(match.group(),match.start(),match.end())
```

```
Orz 4 7
```

Specifying a search pattern must follow some grammar rules.

- Special characters including ! $ ^ * ( ) + = { } [ ] | \ : . ? have special meanings in re. Therefore, to match those characters, those characters must be preceded by \.
- ^foo matches foo at the beginning of a string
- foo$ matches foo at the end of a string
- ab* matches a, ab, abb, abbb, ….
- ab+ matches ab, abb, abbb, ….
- ab? matches a and ab
- * + ? are greedy qualifiers; that is, they match as more characters as possible.

  e.g. re.search("<.*>", "<a> b <c>") finds a match <a> b <c> but does not yield the match <a>.

- *? +? ?? are non-greedy qualifiers; that is, they match as few characters as possible.

e.g. re.search("<.*?>", "<a> b <c>") finds a match <a>.

- RE1 | RE2 matches either RE1 or RE2.

Pattern	Result	Pattern	Result
.	Any characters except a newline (re.DOTALL is not set)	*? +? ??	Non-greedy versions of * + ?
^	Match the start of string	RE{m}	Match m repetitions of RE
$	Match the end of string	RE{m,}	Match at least m repetitions of RE
RE*	Match 0 or more repetitions of RE	RE{m,n}	Match m to n repetitions of RE
RE+	Match at least one repetition of RE	RE{m,n}?	A non-greedy version of {m,n}
RE?	Match 0 or 1 repetition of RE	[]	Indicate a character set

In [18]:

```
print(re.search(r"<.*>", "<a> b <c>")) # greedy qualifier
print(re.search(r"<.*?>", "<a> b <c>"))# nongreedy version of *
```

```
<_sre.SRE_Match object; span=(0, 9), match='<a> b <c>'>
<_sre.SRE_Match object; span=(0, 3), match='<a>'>
```

Patterns for indicating character sets

Pattern	Result	Pattern	Result
[abc]	a or b or c	[^abc]	All characters except a and b and c
[0-9]	0 to 9	\d	[0-9]
[^0-9]	All characters except 0 to 9	\D	[^0-9]
[0-9a-z] [0-9[a-z]]	0 to 9 or a to z	\w	[0-9a-zA-Z]
[0-9&&[^12]]	0 3 to 9	\W	[^\w]
\s	Space characters [\t\n\x0B\f\r]	\S	All characters except space characters [^\s]

Useful patterns for indicating numbers

- RE for integer numbers: [+-]?\d+
- RE for floating numbers:[+-]?(\d+(.?\d*)|.\d+)
- RE for scientific notation:[+-]?(\d+(.?\d*) |.\d+)([eE][+-]?\d+)?

In [14]:

```
# Use re to check if string is a valid integer number
string ='abc'
if re.match(r"[+-]?\d+", string) is None:
    print('invalid number')

# Use exception-handling to check if string is a valid integer number
try:
    int(string)
except ValueError:
    print('invalid number')
```

```
invalid number
invalid number
```

Match groups

Pattern	Result
\b	Match the empty string when it is at the beginning or end of a word
\B	Match the empty string when it is not at the beginning or end of a word
\number	Match the content of the group of the same number
(?P< *name* >...)	This group can be identified by name

(...) indicates the start and end of a group

In [37]:

```
m = re.match(r"(\d+)\.(\d+)", "24.1632")
print(m.group())
print(m.group(0))
print(m.group(1))
print(m.group(2))
```

```
24.1632
24.1632
24
1632
```

Example of using \b: r'\bOrz\b' matches (Orz), 'Orz', abc Orz def but does not match OrzOrz, 123Orz, Orz123.

In [38]:

```
import re
reg = re.compile(r'\bOrz\b')
for m in reg.finditer("(Orz) 'Orz' abc Orz def OrzOrz  123Orz123  456Orz123"):
    print('\\b=>',m.group(),m.start(),m.end())

reg2 = re.compile(r'\BOrz\B')
for m in reg2.finditer("(Orz) 'Orz' abc Orz def OrzOrz  123Orz123  456Orz123"):
    print('\\B=>',m.group(),m.start(),m.end())
```

```
\b=> Orz 1 4
\b=> Orz 7 10
\b=> Orz 16 19
\B=> Orz 35 38
\B=> Orz 46 49
```

Example of using \number

1. r'(\d\d)(\d\d\d)\2'

- (\d\d): group 1 requires 2 digits
- (\d\d\d): group 2 requires 3 digits
- \2 indicates this pattern must be identical to group 2

2. using \number for matching a pair of " and " or a pair of ' and '.

In [40]:

```
print(re.search(r'([\'\"]).*\1','"1234"'))
print(re.search(r'([\'\"]).*\1','"1234\''))
```

```
<_sre.SRE_Match object; span=(0, 6), match='"1234"'>
None
```

Pattern	Result
(?:...)	Matches … but the substring matched by the group cannot be retrieved or referenced.
{?!...)	Matches if a match for … does not match next
(?<=...)	Matches if the current position in the string is preceded by a match for …
(?<!...)	Matches if the current position in the string is not preceded by a match for …
(?(id/name)yes-pattern|no_pattern)	If group \id exists, apply yes-pattern; otherwise, apply no-pattern

In [43]:

```
m = re.search('(?<=abc)def', 'abcdef')
print(m.group(0) if m is not None else None)
m = re.search('(?<=abc)def', 'abddef')
print(m.group(0) if m is not None else None)
m = re.search('(?<=-)\w+', 'spam-egg')
print(m.group(0) if m is not None else None)
```

```
def
None
egg
```

Example of match groups: $(<)?(\w + @\w + (? : \.\w+)+)(?(1) > |$)$

- group 1: (<)? The first group indicates that the first character of a match can be either a '<' or empty
- group 2: $(\w + @\w + (? : \.\w+)+)$
- group 3: $(? : \.\w+)+$ group 3 cannot be referred
- $ If the first character of this match is a '<', the last character of this match must be a '>'; otherwise, this match must be at the end of string

In [70]:

```
m = re.search(r'(<)?(\w+@\w+(?:\.\w+)+)(?(1)>|$)', 'user@mail.com')
print(m,m.group(1),m.group(2))
m = re.search(r'(<)?(\w+@\w+(\.\w+)+)(?(1)>|$)', 'user@mail.com') # (group 3 returns only t
print(m,m.group(1),m.group(2),m.group(3))
m = re.search(r'(<)?(\w+@\w+(?:\.\w+)+)(?(1)>|$)', '<user@mail.com>')
print(m,m.group(1),m.group(2))
m = re.search(r'(<)?(\w+@\w+(?:\.\w+)+)(?(1)>|$)', '<user@mail.com')
print(m,m.group(1),m.group(2))
m = re.search(r'(<)?(\w+@\w+(?:\.\w+)+)(?(1)>|$)', '<user@mail.com ')
print(m)
m = re.search(r'(<)?(\w+@\w+(?:\.\w+)+)(?(1)>|$)', '<user@mail.com<')
print(m)
```

```
<_sre.SRE_Match object; span=(0, 13), match='user@mail.com'> None user@mail.
com
<_sre.SRE_Match object; span=(0, 13), match='user@mail.com'> None user@mail.
com .com
<_sre.SRE_Match object; span=(0, 15), match='<user@mail.com>'> < user@mail.c
om
<_sre.SRE_Match object; span=(1, 14), match='user@mail.com'> None user@mail.
com
None
None
```

search() vs match()

re.match() checks for a match at the beginning of string, whereas re.search() finds a match anywhere in string.

In [13]:

```
import re
mo1 = re.match("c", "abcdef")
mo2 = re.search("c", "abcdef")
print(mo1.span() if mo1 is not None else None)    # No match
print(mo2.span() if mo2 is not None else None)   # Match
```

```
None
(2, 3)
```

Use ^ to force re.search() to find the match at the beginning of string.

In [23]:

```
import re
print(re.search("^c", "abcdef")) # No match
print(re.search("^a", "abcdef")) # Match
```

```
None
<_sre.SRE_Match object; span=(0, 1), match='a'>
```

- In MULTILINE mode (flags=re.MULTILINE), match() only matches at the beginning of the string, whereas search() with a regular expression beginning with '^' will match at the beginning of each line.

In [24]:

```
import re
print(re.match('X', 'A\nB\nX', re.MULTILINE))  # No match
print(re.search('^X', 'A\nB\nX', re.MULTILINE))  # Match
```

```
None
<_sre.SRE_Match object; span=(4, 5), match='X'>
```

findall() vs finditer()

findall() finds all match groups and saves them in a list.

In [1]:

```
import re
print(re.findall(r"12","121234345656"))
print(re.findall(r"(\d\d)\1","121234345656"))
print(re.findall(r"([\'\"])[^\'\"]*\1",'''Hello "Mike", 'John', "Mary"'''))
```

```
['12', '12']
['12', '34', '56']
['"', "'", '"']
```

Use finditer() to obtain each match iteratively.

```
    reg = re.compile(pattern)
    for m in reg.finditer(string):
        print(m.group(),m.start(),m.end())
```

In [27]:

```
import re
reg = re.compile(r"12")
for m in reg.finditer("121234345656"):
    print(m.group(),m.start(),m.end())
```

```
12 0 2
12 2 4
```

In [28]:

```
reg = re.compile(r"(\d\d)\1")
for m in reg.finditer("121234345656"):
    print(m.group(),m.start(),m.end())
```

```
1212 0 4
3434 4 8
5656 8 12
```

In [29]:

```
reg = re.compile(r"([\'\"])[^\'\"]*\1")
for m in reg.finditer('''Hello "Mike", 'John', "Mary"'''):
    print(m.group(),m.start(),m.end())
```

```
"Mike" 6 12
'John' 14 20
"Mary" 22 28
```

Match Objects

m[g] is equal to m.string[m.start(g):m.end(g)]

In [69]:

```
m = re.match(r"(\w+) (\w+)", "Isaac Newton, physicist")
print(m.group(0)) #The entire match or m[0]
print(m.group(1)) # The first parenthesized subgroup. Or m[1]
print(m.group(2)) # The second parenthesized subgroup. Or m[2]
print(m.group(1,2)) # Multiple arguments give us a tuple.
```

```
Isaac Newton
Isaac
Newton
('Isaac', 'Newton')
```

In [66]:

```
m = re.match(r"(?P<first_name>\w+) (?P<last_name>\w+)", "Malcolm Reynolds")
print('first name:{}, last name:{}'.format(m.group('first_name'),m.group('last_name')))
```

```
first name:Malcolm, last name:Reynolds
```

In [71]:

```
m = re.match(r"(..)+", "a1b2c3")  # Matches 3 times.
print(m.group(1))    # Returns only the last match.
print(m.group(0))
```

```
c3
a1b2c3
```

Match Groups=>Dictionary

groupdict() is very useful to save match groups in a dictionary.

In [73]:

```
m = re.match(r"(?P<first_name>\w+) (?P<last_name>\w+)", "Malcolm Reynolds")
print(m.group('first_name'),m.group('last_name'))
d = m.groupdict()
print(d)
```

```
Malcolm Reynolds
{'first_name': 'Malcolm', 'last_name': 'Reynolds'}
```

More Advanced Examples

https://docs.python.org/3/library/re.html (https://docs.python.org/3/library/re.html)

- 6.2.5.4. Making a Phonebook
- 6.2.5.5. Text Munging
- 6.2.5.9. Writing a Tokenizer

Miscellaneous operating system interfaces: os Module

https://docs.python.org/3/library/os.html (https://docs.python.org/3/library/os.html)

- os.path is often used to manipulate the names of paths.

```
import os.path
```

- os.path.abspath(path) returns the absolute path of pathname path.
- os.path.basename(path) returns the base name of pathname path.

```
os.path.basename('C:\\Python\\Source')
```

- os.path.dirname(path)
- os.path.getatime(path) gets the last access time of pathname path.
  - getmtime, getctime
- os.path.getsize(path) gets the size in bytes of path.
- os.path.isfile(path), os.path.isdir(path)
- os.path.exists(path) returns True if path exists;
- os.path.join(path,*paths) joins one or more path components.

```
os.path.join("c:\\Python","A","B")
```

- os.path.split(path) splits pathname into (head,tail), where tail is the last pathname component.

```
os.path.split(os.path.join("c:\\Python","A","B"))
```

In [76]:

```
import os.path
print(os.path.basename('C:\\Python\\Source'))
```

```
Source
```

In [77]:

```
print(os.path.dirname('C:\\Python\\Source'))
```

```
C:\Python
```

In [83]:

```
os.path.join("c:\\Python","A","B","C")
```

Out[83]:

```
'c:\\Python\\A\\B\\C'
```

In [84]:

```
os.path.split(os.path.join("c:\\Python","A","B","C"))
```

Out[84]:

```
('c:\\Python\\A\\B', 'C')
```

- os.getcwd() gets the current working directory.
- os.listdir(directory) gets a list of the file and subdirectory in directory.

```
e.g., os.listdir(os.getcwd()) gets a list of files and subdirectories in the cu
rrent working directory.
```

- os.scandir(directory) can get the file and subdirectory in directory iteratively.

```
 for entry in os.scandir(directory):
     if entry.is_file():
         print(entry.name)
     elif entry.is_dir():
         print(entry.name)
```

    - if you want to traverse the directory tree, os.scandir() must be applied recursively.
    - os.listdir(directory) is equivalent to [entry.name for entry in os.scandir(directory)]
- os.walk() can directly walk the directory tree either top-down or bottom-up.
- os.mkdir(path) creates a directory named path.
- os.rename(oldname, newname) renames the file or directory oldname to newname.
- os.rmdir(path) and os.remove(file) delete the directory path and the file file, respectively.

In [20]:

```
import os
import datetime, time
try:
    system_path = os.environ['SystemRoot']
    for entry in os.scandir(system_path):
        name = os.path.join(system_path,entry.name)
        if entry.is_file():
            print('{:<40s} {} {:10d}'.format(entry.name,datetime.datetime.fromtimestamp(os.
        elif entry.is_dir():
            print('{:<40s} {}'.format('<{}>'.format(entry.name),datetime.datetime.fromtimes
except KeyError:
    print(os.environ.items())
```

```
<ImmersiveControlPanel>                  2018-07-11 22:01:08.304323
<INF>                                    2018-09-15 22:10:19.083570
<InfusedApps>                            2018-04-12 07:38:21.063434
<InputMethod>                            2018-04-12 07:38:21.063434
<Installer>                              2018-08-16 19:48:49.476935
<L2Schemas>                              2018-04-12 07:38:25.735695
<LanguageOverlayCache>                   2018-04-12 07:38:21.063434
<LiveKernelReports>                      2018-09-15 20:59:46.098924
<Logs>                                   2018-09-21 22:41:28.796274
lsasetup.log                             2018-06-19 00:35:09.142151
1380
<media>                                  2018-04-12 07:38:25.813826
mib.bin                                  2018-04-12 07:34:36.260256      4
3131
<Microsoft.NET>                          2018-09-21 22:41:16.850085
<Migration>                              2018-04-12 07:38:21.094687
<ModemLogs>                              2018-04-12 07:38:21.094687
notepad.exe                              2018-04-12 07:34:20.664123     24
5760
NvContainerRecovery.bat                  2018-06-19 00:42:27.520150
```

os.walk(directory, topdown=True, onerror=None, followlinks=False)

os.path https://docs.python.org/3/library/os.path.html#module-os.path (https://docs.python.org/3/library/os.path.html#module-os.path)

os.walk() returns an iterator and yields a 3-tuple (dirpath, dirnames, filenames).

- dirpath is the path to the directory;
- dirnames is a list of the names of the subdirectories in dirpath;
- filename is a list of the names of the ordinary files in dirpath.
- if topdown=False, the triple for a directory is generated after the triples for all of its subdirectories are listed before

In [7]:

```
import os
from os.path import join, getsize, getatime, getmtime, getctime

for root, dirs, files in os.walk(os.environ['SystemRoot'],topdown=True):
    print(root, "consumes", end=" ")
    print(sum(getsize(join(root, name)) for name in files), end=" ")
    print("bytes in", len(files), "non-directory files")
    for x in ['Boot', 'assembly', 'inf', 'Installer', 'System', 'System32', 'SysWOW64', 'wi
        if x in dirs:
            dirs.remove(x)  #because remove those from dirs in-place, os.walk don't visit t
```

```
C:\Windows\Chipset\Driver consumes 1204563 bytes in 3 non-directory files
C:\Windows\Chipset\Driver\Chipset consumes 3293061 bytes in 20 non-directo
ry files
C:\Windows\Chipset\Driver\Chipset\All consumes 18792749 bytes in 234 non-d
irectory files
C:\Windows\Chipset\Driver\Chipset\ia64 consumes 900736 bytes in 2 non-dire
ctory files
C:\Windows\Chipset\Driver\Chipset\Lang consumes 0 bytes in 0 non-directory
files
C:\Windows\Chipset\Driver\Chipset\Lang\CHIP consumes 0 bytes in 0 non-dire
ctory files
C:\Windows\Chipset\Driver\Chipset\Lang\CHIP\ARA consumes 126424 bytes in 2
non-directory files
C:\Windows\Chipset\Driver\Chipset\Lang\CHIP\ARB consumes 17990 bytes in 1
non-directory files
C:\Windows\Chipset\Driver\Chipset\Lang\CHIP\CHS consumes 104088 bytes in 2
non-directory files
C:\Windows\Chipset\Driver\Chipset\Lang\CHIP\CHT consumes 104300 bytes in 2
non-directory files
C:\Windows\Chipset\Driver\Chipset\Lang\CHIP\CSY consumes 140586 bytes in 2
```

Remove a directory and all of the file and subdirectory in this directory

```
import os
for root, dirs, files in os.walk(directory, topdown=False):
    for name in files:
        os.remove(os.path.join(root, name))
    for name in dirs:
        os.rmdir(os.path.join(root, name))
```

Unix style pathname pattern expansion: glob

https://docs.python.org/3/library/glob.html#module-glob (https://docs.python.org/3/library/glob.html#module-glob)

```
import glob
```

- glob.glob(pathname, recursive=False) returns a list of path names matching pathname.
- glob.iglob(pathname, recursive=False) has the same function as glob.glob but it returns an iterator instead of a list
- glob.escape(pathname) escapes all special characters ('?','*', '[').

- Pathname Pattern
  - *: match strings of arbitrary length (including 0)
  - ?: match an arbitrary single character
  - [a,b,c]: match a character in this set
  - If recursive is True, the pattern ** matches any files and zero or more directories and subdirectories.

Example

```
Current directory
A\
    1.c
    2.py
    3.cpp
B\
    1.c
    2.py
    3.cpp
1.c
2.py
3.cpp
```

- glob.glob('*.py') =>['2.py']
- glob.glob('*\.py', recursive=True) => ['2.py', 'A\2.py', 'B\2.py']
- glob.glob('**', recursive=True)=> ['1.c','2.py','3.cpp','A','A\1.c','A\2.py','A\3.cpp','B','B\1.c','B\2.py','B\3.cpp']

In [11]:

```
# this example counts the numbers of exe files in the Windows root directory and the subdir
import glob
import os.path
count = 0
for executable in glob.iglob(os.path.join(os.environ['SystemRoot'],"*.exe"),recursive=True)
    count += 1
print('there are {} executable files'.format(count))

count = 0
for executable in glob.iglob(os.path.join(os.environ['SystemRoot'],"**\*.exe"),recursive=Tr
    count += 1
print('there are {} executable files'.format(count))
```

```
there are 14 executable files
there are 3847 executable files
```

Python Object Serialization: pickle

https://docs.python.org/3/library/pickle.html (https://docs.python.org/3/library/pickle.html)

- The pickle module implements a binary protocol for serializing and de-serializing a Python object.
  - Pickling is a process converting a Python object into a byte stream (a binary file or a bytes-like object).
  - Unpickling is a process converting a byte stream into a Python object.
- There is an efficient pickle module, which is written in C and known as CPickle.

```
try:
    import CPickle
except ImportError:
    import pickle
```

- pickle.dumps(x) converts a Python object to a byte object, and pickle.loads(x) converts a byte object to a Python object.
- pickle.dump(obj, file) saves a pickled object obj into an opened binary file file, and pickle.load(file) loads a pickled object from an opened binary file file.

In [2]:

```
import pickle
x = {'A': 90, 'B':80, 'C':70}
z = pickle.dumps(x) # z is a byte object
y = pickle.loads(z)
print(z)
print(y)
```

```
b'\x80\x03}q\x00(X\x01\x00\x00\x00Bq\x01KPX\x01\x00\x00\x00Cq\x02KFX\x01\x00
\x00\x00Aq\x03KZu.'
{'B': 80, 'C': 70, 'A': 90}
```

In [4]:

```
import pickle

x = {'A': 100, 'B':80, 'C':70}

with open('x.pickle','wb') as f:
    pickle.dump(x,f)

with open('x.pickle','rb') as f:
    y = pickle.load(f)

print(y)
```

```
{'B': 80, 'C': 70, 'A': 100}
```

- The following types can be pickled:
  - None, True, and False
  - integers, floating point numbers, complex numbers
  - strings, bytes, bytearrays
  - tuples, lists, sets, and dictionaries containing only picklable objects
  - classes that are defined at the top level of a module
  - instances of such classes whose **dict** or the result of calling **getstate**() is picklable.
  - functions defined at the top level of a module (using def, not lambda) and built-in functions defined at the top level of a module
    - functions (built-in or user-defined) are pickled by “fully qualified” name reference, not by value. For example,

In [12]:

```
def f(x): return x

p = pickle.dumps(f)
print(p)
```

```
b'\x80\x03c__main__\nf\nq\x00.'
```

- pickle defines three exceptions:
  - pickle.PickleError, pickle.PickingError, pickle.UnpicklingError
- Pickling and unpickling a class instance are implemented in the following manner:

  def save(obj):

  ```
  return (obj.__class__, obj.__dict__)
  ```

  def load(cls, attributes):

  ```
  obj = cls.__new__(cls)
  obj.__dict__.update(attributes)
  return obj
  ```

The behaviors of pickling and unpickling can be changed by implementing some special methods:

- __getnewargs_ex__()
- __getnewargs__()
- __getstate__()
- __setstate__(state)

For details, refer to https://docs.python.org/3/library/pickle.html#pickling-class-instances (https://docs.python.org/3/library/pickle.html#pickling-class-instances)

Python object persistence: shelve

https://docs.python.org/3/library/shelve.html (https://docs.python.org/3/library/shelve.html)

- shelve is a persistent dictionary-like object.
- Keys must be strings and values must be picklable objects.
  - shelve.open(filename): Open a persistent dict object.
  - shelve.close(): Synchronize and close the persistent dict object.
  - shelve.sync(): Write back all entries in the cache

In [20]:

```
# conda install shelv
import shelve

sh = shelve.open('test.shelve')
sh['key1']=[1,2,3,4,5]
sh.sync()

#....

sh.close()
```

In [22]:

```
import shelve

with shelve.open('test.shelve') as sh2:
    print(sh2['key1'])
```

```
[1, 2, 3, 4, 5]
```

JSON encoder and decoder: json

https://docs.python.org/3/library/json.html (https://docs.python.org/3/library/json.html)

The json module can be used to convert between JSON objects and Python objects.

Python Objects => JSON Objects

- json.dumps()
- json.dump(obj,fp): Serialize obj as a JSON formatted stream to fp.

JSON Objects => Python Objects

- json.loads()
- json.load(fp):Deserialize fp (supporting file-like object containing a JSON document) to a Python object by following the conversion table.

Python	JSON
dict	Object
list, tuple	Array
str	String
int, float	Number
True	true
False	false
None	null

In [28]:

```python
import json

obj = {"name": "john", "age": 18, "pets" :["cat","dog"]}

#Serialize obj to a JSON formatted string
json_str = json.dumps(obj)
print(json_str)

# this string has no blanks
json_str = json.dumps(obj,separators=(',',':'))
print(json_str)

json_str = json.dumps(obj,separators=(',',':'),sort_keys=False, indent=2)
print(json_str)

json_str = json.dumps(obj,separators=(',',':'),sort_keys=True, indent=2)
print(json_str)

# Serialize obj as a JSON formatted stream to fp
with open("data.txt", "w") as fp:
    json.dump(obj, fp)
```

```
{"name": "john", "pets": ["cat", "dog"], "age": 18}
{"name":"john","pets":["cat","dog"],"age":18}
{
  "name":"john",
  "pets":[
    "cat",
    "dog"
  ],
  "age":18
}
{
  "age":18,
  "name":"john",
  "pets":[
    "cat",
    "dog"
  ]
}
```

In [71]:

```
import json
import collections

# This function will be called if obj is an object of Python Class
def convert(obj):
    return {"name": obj.name, "age":obj.age, "x": 0}

class Person:
    def __init__(self,name,age):
        self.name= name
        self.age = age

person = collections.namedtuple('person',['name','age'])

obj_P = Person("john",18)
obj_p = person("john",18)
print('class Person:',json.dumps(obj_P,default=convert))
print('named tuple person:',json.dumps(obj_p,default=convert))


x=person(*json.loads(json.dumps(obj_p)))
print(x)

# This function will be called if obj is dict
def custom_function(obj):
    return Person(obj['name'],obj['age'])

p1 = json.loads(json.dumps(obj_P,default=convert),object_hook=custom_function)
print(p1)
```

```
class Person: {"x": 0, "name": "john", "age": 18}
named tuple person: ["john", 18]
person(name='john', age=18)
<__main__.Person object at 0x0000026A1C936DA0>
```

Reading and Writing CSV Files

https://docs.python.org/3/library/csv.html#module-csv (https://docs.python.org/3/library/csv.html#module-csv)

Reading	Writing
reader() returns a reader object	writer() returns a writer object
DictReader() returns a dictionary	DictWriter() returns a writer object which can map dictionaries to rows

In [72]:

```
import csv
customers =[
{'firstname': 'Justin', 'lastname':'Lee'},
{'firstname': 'Teddy', 'lastname':'Liu'},
{'firstname': 'Terry', 'lastname':'Huang'}
]

# notice that newline=''
with open('names.csv', 'w', encoding='Big5',newline='') as csvfile: # encoding='utf-8',
    fieldnames = ['firstname', 'lastname']
    writer = csv.DictWriter(csvfile, fieldnames=fieldnames, quotechar="'")
    writer.writeheader()
    writer.writerows(customers) # writer.writerow({'firstname': 'Justin', 'lastname':'Lee'}
```

In [73]:

```
import csv

with open("names.csv", 'r',encoding='Big5',newline='') as csvfile: # encoding='utf-8'
    reader = csv.reader(csvfile,delimiter=',', quotechar="'")
    for row in reader:
        print(row)

with open("names.csv", 'r',encoding='Big5',newline='') as csvfile:
    reader = csv.DictReader(csvfile,delimiter=',', quotechar="'")
    for row in reader:
        print(row['firstname'],row['lastname'])
```

```
['firstname', 'lastname']
['Justin', 'Lee']
['Teddy', 'Liu']
['Terry', 'Huang']
Justin Lee
Teddy Liu
Terry Huang
```

In [ ]:

```

```