Date and time

Python's time and calendar modules help track of dates and times.

What is Tick?

Time intervals are floating-point numbers in units of seconds. Particular instants in time are expressed in senconds since 12:00am January 1,1970(epoch). There is a popular **time** module available in python which provides function for working with times, and for converting between representations. The function time.time() returns the current system time in ticks since epoch.

In [4]:
```
import time

ticks = time.time()
print("Number of ticks in 12:00am January 1,1970:",ticks)
```

```
Number of ticks in 12:00am January 1,1970: 1579233491.4303014
```

Dates before the epoch can not be represented in this form. Dates in far future also cannot be represented this way. The cutoff point is sometime in 2038 for UNIX and Windows.

Getting current time

In [5]:
```
print("Local time is: ",time.localtime(time.time()))
```

```
Local time is:  time.struct_time(tm_year=2020, tm_mon=1, tm_mday=16, tm_hour=23, tm_min=3, tm_sec=
4, tm_wday=3, tm_yday=16, tm_isdst=0)
```

Getting formatted time

In [6]:
```
print("Local time is: ", time.asctime(time.localtime(time.time())))
```

```
Local time is:  Thu Jan 16 23:04:46 2020
```

Getting calendar for a month

In [8]:
```
import calendar

cal = calendar.month(2008,1)
print(cal)
```

```
    January 2008
Mo Tu We Th Fr Sa Su
    1  2  3  4  5  6
 7  8  9 10 11 12 13
14 15 16 17 18 19 20
21 22 23 24 25 26 27
28 29 30 31
```

Is a leap year ?

In [11]:
```
print(calendar.isleap(2008))
```

```
True
```

** Which day is this?**

In [13]:
```
print(calendar.weekday(2008,1,3))
```

```
3
```

0-Monday to 6-Sunday

Exceptions Handling

List of standard Exceptions:

- Exception : Base class for all exceptions
- ZeroDivisionError : Raised when division or modulo by zero place for all numeric types.
- IndexError : Raised when an index is not found in a sequence.
- KeyError : Raised when the specific key is not found in the dictionary.

- NameError : Raised when an idenfifier is not found in the local or global namespace.
- SyntaxError : Raised when there is an error in Python syntax.
- IndentationError : Raised when identation is not specified properly.
- TypeError : Raised when an operation or function is attempted that is invalid for the specific data type.
- ValueError : Raised when the build-in function for a data type has the valid type of arguments, but the arguments have invalid vlues specified.
- RuntimeError : Raised when a generated error does not fall into any category.

What is Exception ? An exception is an event, which occurs during the execution of a program that disrupts normal flow of the program's instructions.

When a python script raises an exception, it must either handle the exception immediately otherwise it terminates and quits.

Handling an exception If you have some suspicious code that may raise an exception, you can defend your program by placing the suspicious code in a **try:** block.

```
try:
   You do your operations here;
   ......................
except ExceptionI:
   If there is ExceptionI, then execute this block.
except ExceptionII:
   If there is ExceptionII, then execute this block.
   ......................
else:
   If there is no exception then execute this block.
```

In [20]:
```
# Example 1
try:
    fh = open("testfile", "w")
    fh.write("This is my test file for handling exception.")
except IOError:
    print("Error: can't find the file or read data")
else:
    print("Written content in the file successfully")
    fh.close()
```

```
Written content in the file successfully
```

In [21]:
```
# Example 2
try:
   fh = open("testfile", "r")
   fh.write("This is my test file for exception handling!!")
except IOError:
   print ("Error: can\'t find file or read data")
else:
   print ("Written content in the file successfully")
```

```
Error: can't find file or read data
```

the except clause with no exceptions

```
the except clause with no exceptions
try:
   You do your operations here;
   ......................
except:
   If there is any exception, then execute this block.
   ......................
else:
   If there is no exception then execute this block.
```

This type of try-except statements catches all the exceptions that occur. Using this kind of try-except statement is not considered a good programming practice though, because it catches ll exceptions but does not make the programmer identify the root cause of the problem that my occur.

Try-finally Clause

You can use **finally:** block along with **try:** block. The finally block is a place to put any code that must execute, whether the try-block raise an exception or not.

```
try:
   You do your operations here;
   ......................
   Due to any exception, this may be skipped.
finally:
   This would always be executed.
   ......................
```

In [24]:
```
# Example 1
try:
   fh = open("testfile", "r")
   fh.write("This is my test file for exception handling!!")
except IOError:
   print ("Error: can\'t find file or read data")
finally:
   print ("must run this block whether the try block raise an exception or not")
```

```
Error: can't find file or read data
must run this block whether the try block raise an exception or not
```

In [28]:
```
# Example 2
try:
   fh = open("testfile", "w")
   try:
      fh.write("This is my test file for exception handling!!")
   finally:
      print ("Going to close the file")
      fh.close()
except IOError:
   print ("Error: can\'t find file or read data")
```

```
Going to close the file
```

In [29]:
```
# Example 3

try:
   fh = open("testfile", "r")
   try:
      fh.write("This is my test file for exception handling!!")
   finally:
      print ("Going to close the file")
      fh.close()
except IOError:
   print ("Error: can\'t find file or read data")
```

```
Going to close the file
Error: can't find file or read data
```

Argument of an Exception

An exception can have an argument, which is a value that gives additional information about the problem. The contens of the argument vary by exception. You can capture an exception's argument by supplying a variable in the except clause as follows

```
try:
   You do your operations here;
   ......................
except ExceptionType as Argument:
   You can print value of Argument here...
```

If you write the code to handle a single exception, you can have a variable follow the name of the exception in the except statement. If you are trapping multiple exceptions, you can have a variable follow the tuple of exception.

This variable receives the value of the exception mostly containing the cause of the exception. The variable can receive a single value or multiple values in the form of a tuple. This tuple usually contians the error string, the error number and an error location.

In [46]:
```
def temp_convert(var):
   try:
      return int(var)
   except Exception as Argument:
      print ("The argument does not contain numbers\n",Argument)

# Call above function here.
temp_convert("xyz");
```

```
The argument does not contain numbers
 invalid literal for int() with base 10: 'xyz'
```

Raising exceptions

You can raise exceptions using **raise** statement as follows:

```
raise [Exception [, args [, traceback]]]
```

Here,Exception is the type of exception (for example, NameError) and arg is a value for the exception argument (optional,if not supplied exception argument is None)

The traceback is also optional (rarely used) and if present, is the traceback object used for the exception.

In [69]:
```
# Example 1
def function_name(level):
    if level < 1:
        raise Exception("Invalid layer",level)
    return level

try:
    function_name(0)
except Exception as e:
    print("level is less than 1 \n Error: \n",e)
else:
    print("Level is valid")
```

```
level is less than 1
 Error:
 ('Invalid layer', 0)
```

User defined Exceptions

Here is an example related to RuntimeError. A class is created that is subclassed from RuntimeError. This is useful when you need to display more specific information when exception is caught.

In the **try:** block user-defined exception is raised and caught in the **except:** block. The variable e is used to create an instance of class NetworkError.

In [91]:
```
class NetwrokError(RuntimeError):
    def __init__(self,*args):
        self.args = args

try:
    raise NetwrokError("no connection")
except NetwrokError as e:Files
    print(e.args)
```

```
('no connection',)
```

Files I/O

Opening and closing files Python provides basic functions and methods necessary to manipulate files by default. You can do most of the file manipulation using a file object.

Open Before you can read or write a file, you have to open it using build in **Open** function.

This function creates a file object, which would be utilized to call other methods associated with it.

```
syntax:
file_object = open(file_name[, access_mode][, buffering])
```

- file_name : string value that contains the name of the file that you want to access.
- access_mode: Determines the mode in which the file has to be opened. i.e. read,write, append,etc.
- buffereing:

-- If set to 0 => No buffering takes place.

-- If set to 1 => line buffering is perfomed while buffering a file.

-- If set to greater than 1 => buffering action is performed with the indicated buffer size.

-- If negative => the buffer size is the system default

os module for working with files

Renaming

```
os.rename(current_file_name, new_file_name)
```

In [94]:
```
import os
os.rename('testfile','mytestfile')
```

In [96]:
```
print(os.listdir())
```

```
['pics', '.ipynb_checkpoints', 'Data Structures.ipynb', 'Basic python.ipynb', 'mytestfile']
```

Removing

```
os.remove(filename)
```

Make new directory

```
os.mkdir("New dir")
```

Change directiory

```
os.chdir("newdir")
```

Current working dir

```
os.getcwd()
```

delete current directory

```
os.rmdir('dirname')
```