What are Exceptions:

* Exceptions occur whenever syntactically correct code results in an error.
* This is usually a result of something unexpected happening during the code’s execution.
* Base Exceptions – used mostly as base classes for other exceptions:
  + BaseException – base class for all built-in exceptions.
  + Exception – all built-in, non-system-exiting exceptions are derived from this. User-defined exceptions should inherit from this.
  + ArithmeticError – OverflowError, ZeroDivisionError, FloatingPointError.
  + BufferError – when a buffer operation fails.
  + LookupError – IndexError, KeyError.
* Concrete Exceptions – exceptions that are normally raised:
  + AssertionError – raised when ‘assert’ statement fails.
  + AttributeError – when an attribute reference or assignment fails.
  + EOFError – when an input hits the end of file without reading any data.
  + GeneratorExit
  + ImportError – when the ‘import’ has trouble loading in a module.
    - ModuleNotFoundError – subclass of ImportError when the imported module could not be found.
  + IndexError – when a sequence subscript is out of range.
  + KeyError – when a dictionary key is not found in the set of existing keys.
  + KeyboardInterrupt – when the user inputs a code-ending keyboard command. Inherits from BaseException instead of Exception so the code doesn’t continue running.
  + MemoryError – when an operation runs out of memory, but the situation may still be possibly rescued by deleting some objects from the stack.
  + NameError – when local or global name is not found.
  + NotImplementedError – raised by abstract methods when a subclass does not override the method.
  + OSError – when a system function returns a system-related error.
  + OverflowError – when the result of an arithmetic expression is too large to be represented.
  + RecursionError – raised when maximum recursion depth is exceeded.
  + ReferenceError – when a weak reference proxy is used to access an attribute of a referent after it has been garbage collected.
  + RuntimeError – when an error occurs that doesn’t fall into any of the other categories.
  + StopIteration – raised by next() and \_\_next\_\_() to signal that there are no further items produced by the iterator.
  + StopAsyncIteration – raised by \_\_anext\_\_() of an asynchronous iterator object to stop the iteration.
  + SyntaxError – raised when the parser encounters a syntax error.
  + IndentationError – subclass of SyntaxError when there is an error in indentation.
  + TabError – subclass of IndentationError when indents are inconsistent.
  + SystemError – when the interpreter finds an internal error.
  + SystemExit – inherits from BaseException, raised by sys.exit() function.
  + TypeError – when an operation or function is applied to an object of inappropriate type.
  + UnboundLocalError – when reference is made to local variable in a function but no value has been bound to that variable.
  + UnicodeError – inherits from ValueError, when Unicode-related encoding or decoding error occurs.
  + UnicodeEncodeError – Unicode-related error occurs during encoding.
  + UnicodeDecodeError – Unicode-related error occurs during decoding.
  + UnicodetranslateError – Unicode-related error occurs during translating.
  + ValueError – when an operation or function receives an argument that has the right type but inappropriate value.
  + ZeroDivisionError – raised when the second argument of a division or modulo operator is 0.
  + EnvironmentError, IOError, and WindowsError are aliases of OSError starting in Python 3.3.
* OS Exceptions – subclasses of OSError:
  + BlockingIOError – when an operation would block an object set for non-blocking operation.
    - Errno: EAGAIN, EALREADY, EWOULDBLOCK, and EINPROGRESS
  + ChildProcessError – when an operation on a child process failed.
    - Errno: ECHILD
  + ConnectionError – base class for connection-related issues.
  + BrokenPipeError – subclass of ConnectionError, raised when trying to write on a pipe while the other end has been closed.
    - Errno: EPIPE, ESHUTDOWN
  + ConnectionAbortedError – subclass of ConnectionError, raised when a connection attempt is aborted by the peer.
    - Errno: ECONNABORTED
  + ConnectionRefusedError – subclass of ConnectionError, raised when a connection attempt is refused by the peer.
    - Errno: ECONNREFUSED
  + ConnectionResetError – subclass of ConnectionError, raised when a connection nis reset by the peer.
    - Errno: ECONNRESET
  + FileExistsError – raised when trying to create a file/directory which already exists.
    - Errno: EEXIST
  + FileNotFoundError – raised when a file/directory is required but doesn’t exist.
    - Errno: ENOENT
  + InterruptedError – raised when a system call is interrupted by an incoming signal.
    - Errno: EINTR
    - Changed in 3.5, Python will retry the system call when interrupted.
  + IsADirectoryError – raised when a file operation is requested on a directory.
    - Errno: EISDIR
  + NotADirectoryError – raised when a directory operation is requested on something that is not a directory.
    - Errno: ENOTDIR
  + PermissionError – raised when trying to run an operation without the adequate access rights, e.g. filesystem permissions.
    - Errno: EACCES, EPERM, and ENOTCAPABLE
  + ProcessLookupError – raised when a given process doesn’t exist.
    - Errno: ESRCH
  + TimeoutError – raised when a system function timed out at the system level.
    - Errno: ETIMEDOUT

Handle errors using Python-defined Exceptions:

* When exceptions occur, you want your program to be written in such a way that the exception can be dealt with without stopping the execution of your code.
* Try-except blocks are used to make sure exceptions don’t stop your code from running.
* Try:
  + Risky code that might raise exceptions will go here.
* Except:
  + Code that runs when an exception is raised in the Try-block.
* Else:
  + Code that runs when no exceptions are raised.
* Finally:
  + Code that runs whether or not an exception is raised.
* You can create different Except blocks to handle specific errors, but only the first matched exception handler will execute:
  + Except NameError as E:
    - Print(E)
  + Except (NameError, TypeError, ValueError):
    - Print(“Something went wrong”)
* Raise keyword:
  + Use to raise exceptions or errors.
  + raise Exception(“text”)
  + Using the ‘raise’ keyword without giving an Exception class will reraise the exception that last occurred.
  + Raise is useful for input validation.
* Assert keyword:
  + Used when debugging code.
  + Allows you to test if a condition in your code returns True, and if not it will raise an AssertionError.
  + assert [condition], [message (optional)]
  + Ex.
    - x = “hello”
    - assert x == “goodbye”, “X should be ‘hello’”

Exception Hierarchy:

* Base Exception
  + Exception
    - AttributeError
    - ArithmeticError
      * ZeroDivisonError
      * FloatingPointError
      * OverflowError
    - EOFError
    - NameError
    - LookupError
      * IndexError
      * KeyError
    - OSError
      * ConnectionError
      * FileNotFoundError
      * InterruptedError
      * PermissionError
      * TimeoutError
    - TypeError
    - ValueError
  + SystemExit
  + GeneratorExit
  + KeyboardInterrupt

Event Classes:

from threading import Event

* Event objects are used to manage communication between threads.
* Event.set() – set an internal flag to True or False.
* Event.clear() – resets the flag to False.
* Event.wait() – blocks the thread until the flag is set to True.

Self-Defined Exceptions:

* User-defined exceptions extend from ‘Exception’ class.
  + class CustomError(Exception)
* You can then use the ‘raise’ keyword to raise your newly defined exceptions.