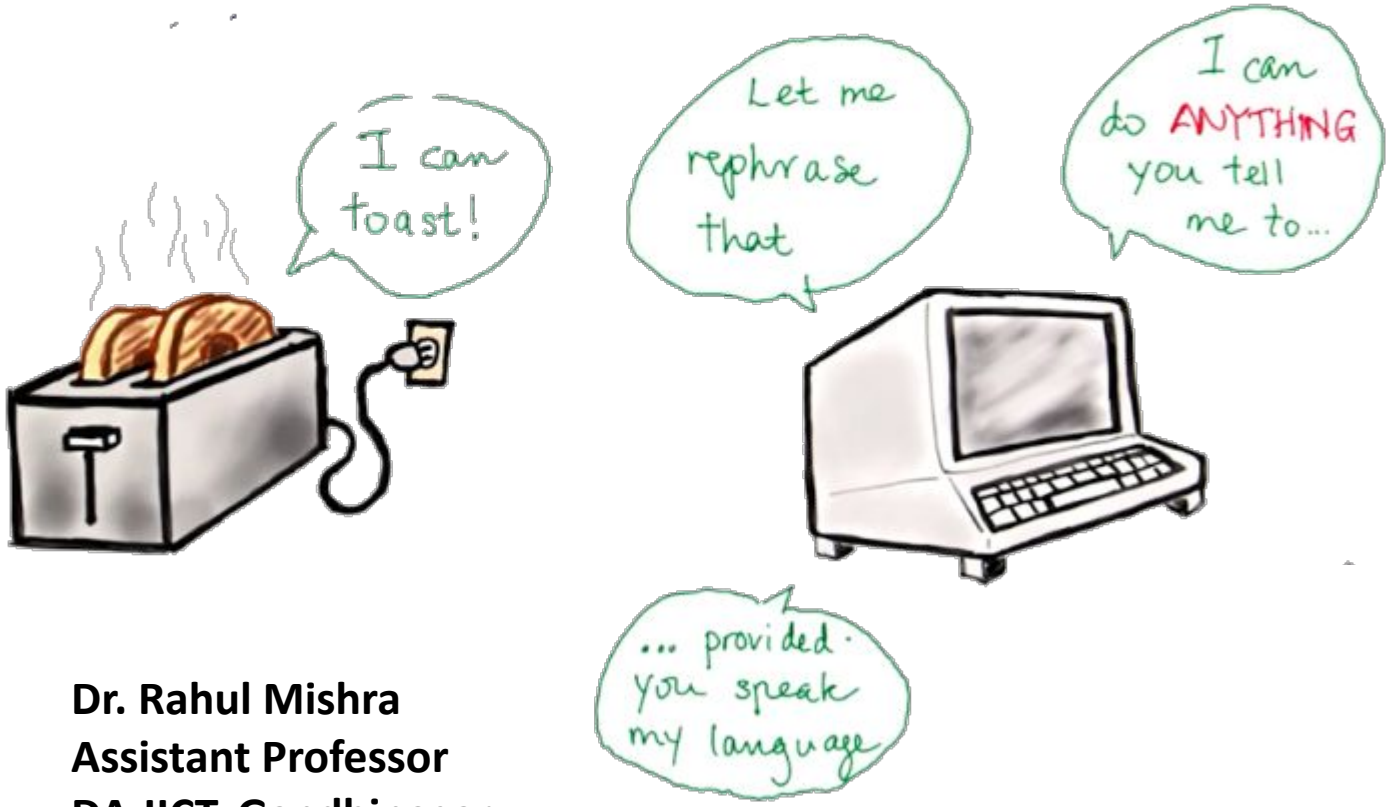
# Programming Lab

## Autumn Semester

**Course code: PC503**





**Dr. Rahul Mishra**
**Assistant Professor**
**DA-IICT, Gandhinagar**

# Lecture 16

## Exception Handling

# Errors and Exceptions

## *3. Handling Exceptions*

- The `try` … `except` statement has an optional *else clause*, which, when present, must follow all *except clauses*.

- It is useful for code that must be executed if the *try clause* does not raise an exception. For example:

```
>>> for arg in sys.argv[1:]:
...     try:
...         f = open(arg, 'r')
...     except OSError:
...         print('cannot open', arg)
...     else:
...         print(arg, 'has', len(f.readlines()), 'lines')
...         f.close()
...
>>>
```

# Errors and Exceptions

## *3. Handling Exceptions*

The use of the `else` clause is better than adding additional code to the `try` clause because it avoids accidentally catching an exception that wasn't raised by the code being protected by the `try` … `except` statement.

Exception handlers do not handle only exceptions that occur immediately in the *try clause*, but also those that occur inside functions that are called (even indirectly) in the *try clause*. For example:

```
>>>
```

```
>>>
```

# Errors and Exceptions

## *3. Handling Exceptions*

```
>>> try:
...     this_fails()
... except ZeroDivisionError as err:
...     print('Handling run-time error:', err)
...
Handling run-time error: division by zero
>>>
```

# Errors and Exceptions

## *4. Raising Exceptions*

The raise statement allows the programmer to force a specified exception to occur. For example:

```
>>> raise NameError('HiThere')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: HiThere
>>>
```

- The sole argument to raise indicates the exception to be raised.
- This must be either an exception instance or an exception class (a class that derives from **Base Exception**, such as **Exception** or one of its subclasses).
- If an exception class is passed, it will be implicitly instantiated by calling its constructor with no arguments:

# Errors and Exceptions

## *4. Raising Exceptions*

```
>>> raise ValueError  # shorthand for 'raise
ValueError()'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError
>>>
```

If you need to determine whether an exception was raised but don't intend to handle it, a simpler form of the raise statement allows you to re-raise the exception:

```
>>> try:
...     raise NameError('HiThere')
... except NameError:
...     print('An exception flew by!')
...     raise
...
An exception flew by!
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
NameError: HiThere
```

# Errors and Exceptions

## *5. Exception Chaining*

If an unhandled exception occurs inside an except section, it will have the exception being handled attached to it and included in the error message:

```
>>> try:
...     open("database.sqlite")
... except OSError:
...     raise RuntimeError("unable to handle error")
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
FileNotFoundError: [Errno 2] No such file or
directory: 'database.sqlite'

During the handling of the above exception, another
exception occurred:

Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
RuntimeError: unable to handle error
>>>
```

# Errors and Exceptions

## *5. Exception Chaining*

To indicate that an exception is a direct consequence of another, the raise statement allows an optional from clause:

```
# exc must be an exception instance or None.
>>> raise RuntimeError from exc
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'exc' is not defined
>>>
```

# Errors and Exceptions

## *5. Exception Chaining*

This can be useful when you are transforming exceptions. For example:

```
>>> def func():
...     raise ConnectionError
...
>>> try:
...     func()
... except ConnectionError as exc:
...     raise RuntimeError('Failed to open database') from
exc
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
  File "<stdin>", line 2, in func
ConnectionError

The above exception was the direct cause of the following
exception:

Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
RuntimeError: Failed to open database
>>>
```

# Errors and Exceptions

## *5. Exception Chaining*

It also allows disabling automatic exception chaining using the from None idiom:

```
>>> try:
...     open('database.sqlite')
... except OSError:
...     raise RuntimeError from None
...
Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
RuntimeError
>>>
```

# Errors and Exceptions

## *6. User-defined Exceptions*

- *Programs may name their own exceptions by creating a new exception class.* Exceptions should typically be derived from the Exception class, either directly or indirectly.

- Exception classes can be defined that do anything any other class can do, but are usually kept simple, often only offering a number of attributes that allow information about the error to be extracted by handlers for the exception.

- Most exceptions are defined with names that end in “Error”, similar to the naming of the standard exceptions.

- Many standard modules define their own exceptions to report errors that may occur in functions they define

# Errors and Exceptions

## *7. Defining Clean-up Actions*

The try statement has another optional clause which is intended to define clean-up actions that must be executed under all circumstances. For example:

```
>>> try:
...     raise KeyboardInterrupt
... finally:
...     print('Goodbye, world!')
...
Goodbye, world!
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
KeyboardInterrupt
>>>
```

# Errors and Exceptions

## *7. Defining Clean-up Actions*

- If a ***finally*** clause is present, the ***finally*** clause will execute as the last task before the try statement completes. ***The finally*** clause runs whether or not the try statement produces an exception. The following points discuss more complex cases when an exception occurs:

- If an exception occurs during the execution of the try clause, the exception may be handled by an except clause. If the exception is not handled by an except clause, the exception is re-raised after the ***finally*** clause has been executed.

- An exception could occur during the execution of an except or else clause. Again, the exception is re-raised after the **finally** clause has been executed.

# Errors and Exceptions

## *7. Defining Clean-up Actions*

- If the **finally** clause executes a break, continue, or return statement, exceptions are not re-raised.

- If the try statement reaches a break, continue, or return statement, **the finally** clause will execute just prior to the break, continue or return statement's execution.

- If a finally clause includes a return statement, the returned value will be the one from the finally clause's return statement, not the value from the try clause's return statement.

# Errors and Exceptions

## *7. Defining Clean-up Actions*

```
>>> def bool_return():
...     try:
...         return True
...     finally:
...         return False
...
>>> bool_return()
False
```

# Errors and Exceptions

## *7. Defining Clean-up Actions*

```
>>> def divide(x, y):
...     try:
...         result = x / y
...     except ZeroDivisionError:
...         print("division by zero!")
...     else:
...         print("result is", result)
...     finally:
...         print("executing finally clause")
...
>>> divide(2, 1)
result is 2.0
executing finally clause
>>> divide(2, 0)
division by zero!
executing finally clause
>>> divide("2", "1")
executing finally clause
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in divide
TypeError: unsupported operand type(s) for /: 'str' and 'str'
```

# Classes

- Classes provide a means of bundling data and functionality together.
- Creating a new class creates a *new type of object*, allowing new *instances* of that type to be made.
- Each class instance can have *attributes attached to it to maintain its state*.
- Class instances can also have methods (defined by their class) for modifying their state.
- Compared with other programming languages, Python's class mechanism adds classes with a minimum of new syntax and semantics. It is a mixture of the class mechanisms found in C++ and Modula-3.

# Classes

- Python classes provide all the standard features of **Object Oriented Programming**: the class inheritance mechanism allows *multiple base classes, a derived class can override any methods of its base class or classes, and a method can call the method of a base class with the same name*.

- Objects can contain arbitrary amounts and kinds of data. As is true for modules, classes partake of the dynamic nature of Python: they are created at runtime and can be modified further after creation.

- In C++ terminology, normally class members (including the data members) are public (except see below Private Variables), and all member functions are virtual.

- Unlike C++, built-in types can be used as base classes for extension by the user. Also, like in C++, most built-in operators with special syntax (arithmetic operators, subscripting, etc.) can be redefined for class instances.

# Classes

## Python Scopes and Namespaces

- A namespace is a mapping from names to objects.
- Most namespaces are currently implemented as Python dictionaries, but that's normally not noticeable in any way (except for performance), and it may change in the future.
- Examples of namespaces are the ***set of built-in names*** (containing functions such as ***abs(),*** and ***built-in exception names***); the ***global names in a module; and the local names in a function invocation***.
- In a sense, the set of attributes of an object also forms a namespace.
- The important thing to know about namespaces is that there is absolutely no relation between names in different namespaces; for instance, two different modules may both define a function maximize without confusion — users of the modules must prefix it with the module name.

# Classes

## Python Scopes and Namespaces

- The word attribute for any name following a dot — for example, in the expression ***z.real, real is an attribute of the object z.***

- Strictly speaking, references to names in modules are attribute references: in the expression ***modname.funcname***, modname is a module object and funcname is an attribute of it.

- In this case, there happens to be a straightforward mapping between the module's attributes and the global names defined in the module: they share the same namespace!

# Classes

## Python Scopes and Namespaces

- A **scope** is a textual region of a Python program where a namespace is directly accessible. "**Directly accessible**" here means that an unqualified reference to a name attempts to find the name in the namespace.

- Although scopes are determined statically, they are used dynamically. At any time during execution, there are 3 or 4 nested scopes whose namespaces are directly accessible:

- *the **innermost scope**, which is searched first, contains the local names*

- *the scopes of any enclosing functions, which are searched starting with the nearest **enclosing scope**, contain non-local, but also non-global names*

- *the **next-to-last scope** contains the current module's global names*

- *the **outermost scope** (searched last) is the namespace containing built-in names*

```
def scope_test():
    def do_local():
        spam = "local spam"

    def do_nonlocal():
        nonlocal spam
        spam = "nonlocal spam"

    def do_global():
        global spam
        spam = "global spam"

    spam = "test spam"
    do_local()
    print("After local assignment:", spam)
    do_nonlocal()
    print("After nonlocal assignment:", spam)
    do_global()
    print("After global assignment:", spam)

scope_test()
print("In global scope:", spam)
```

The output of the example code is:

```
After local assignment: test spam
After nonlocal assignment: nonlocal spam
After global assignment: nonlocal spam
In global scope: global spam
```