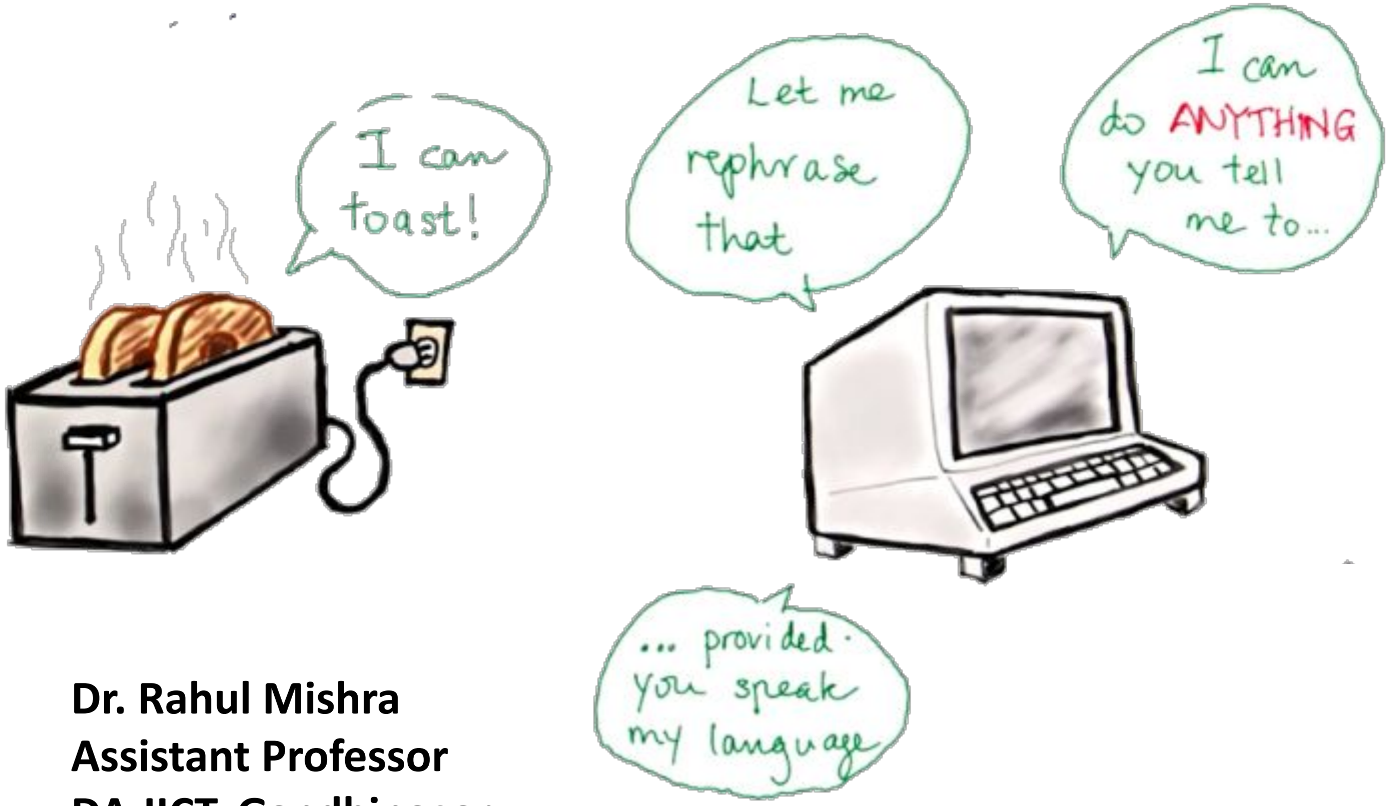
ENGINEERS WITH
SOCIAL RESPONSIBILITY

Programming Lab

Autumn Semester

Course code: PC503


I can toast!
Let me rephrase that
I can do ANYTHING you tell me to...
... provided you speak my language

Dr. Rahul Mishra
Assistant Professor
DA-IICT, Gandhinagar

Images are collected from the web and may be subject to copyright

Lecture 16

Exception Handling

Errors and Exceptions

1. *Syntax Errors*

Syntax errors, also known as parsing errors, are perhaps the most common kind of complaint you get while you are still learning Python:

```
>>> while True print('Hello world')
  File "<stdin>", line 1
    while True print('Hello world')
               ^
SyntaxError: invalid syntax
```

- The parser repeats the offending line and displays a little *‘arrow’ pointing at the earliest point* in the line where the error was detected.
- The error is caused by (or at least detected at) the token preceding the arrow: in the example, the error is detected at the function *print(), since a colon (':') is missing before it.*
- File name and *line number are printed so you know where to look in case the input came from a script.*

Errors and Exceptions

2. Exceptions

- Even if a statement or expression is syntactically correct, it may cause an error when an attempt is made to execute it.

- Errors detected during execution are called exceptions and are not unconditionally fatal: you will soon learn how to handle them in Python programs.

- Most exceptions are not handled by programs, however, and result in error messages as shown here:

```
>>> 10 * (1/0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
>>> 4 + spam*3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'spam' is not defined
>>> '2' + 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate str
(not "int") to str
>>>
```

Errors and Exceptions

2. Exceptions

- The last line of the error message indicates what happened.

- Exceptions come in different types, and the type is printed as part of the message: the types in the example are ***ZeroDivisionError, NameError and TypeError.***

- The string printed as the exception type is the name of the built-in exception that occurred.

- This is true for all built-in exceptions, but need not be true for user-defined exceptions (although it is a useful convention).

- Standard exception names are built-in identifiers (not reserved keywords).

Errors and Exceptions

2. Exceptions

```
>>> 10 * (1/0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
>>> 4 + spam*3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'spam' is not defined
>>> '2' + 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate str (not "int") to str
>>>
```

Errors and Exceptions

2. Exceptions

- The last line of the error message indicates what happened.

- Exceptions come in different types, and the type is printed as part of the message: the types in the example are **ZeroDivisionError, NameError and TypeError.**

- The string printed as the exception type is the name of the built-in exception that occurred. This is true for all built-in exceptions, but need not be true for user-defined exceptions (although it is a useful convention).

- Standard exception names are built-in identifiers (not reserved keywords).

Errors and Exceptions

3. Handling Exceptions

- It is possible to write programs that handle selected exceptions.
- Look at the following example, which asks the user for input until a valid integer has been entered, but allows the user to interrupt the program
- (using Control-C or whatever the operating system supports);
- note that a user-generated interrupt is signalled by raising the KeyboardInterrupt exception.

Errors and Exceptions

3. Handling Exceptions

```
>>> while True:
...     try:
...         x = int(input("Please enter a number: "))
...         break
...     except ValueError:
...         print("Oops!  That was no valid number.  Try again...")
...
Please enter a number: q
Oops!  That was no valid number.  Try again...
Please enter a number: 12
>>>
```

Errors and Exceptions

3. Handling Exceptions

The try statement works as follows.

- First, the *try clause* (the statement(s) between the try and except keywords) is executed.
- If no exception occurs, the *except clause* is skipped and execution of the try statement is finished.
- If an exception occurs during execution of the try clause, the rest of the clause is skipped. Then, if its type matches the exception named after the except keyword, the *except clause* is executed, and then execution continues after the try/except block.
- If an exception occurs which does not match the exception named in the *except clause*, **it is passed on to outer try statements; if no handler is found, it is an *unhandled exception* and execution stops with a message as shown above.**

Errors and Exceptions

3. Handling Exceptions

- A try statement may have more than one *except clause*, to specify handlers for different exceptions. At most one handler will be executed.

- Handlers only handle exceptions that occur in the corresponding *try clause*, not in other handlers of the same try statement.

- An *except clause* may name multiple exceptions as a parenthesized tuple, for example:

```
... except (RuntimeError, TypeError, NameError):
...     pass
```

Errors and Exceptions

3. Handling Exceptions

- A class in an except clause is compatible with an exception if it is the same class or a base class thereof (but not the other way around

- — an except clause listing a derived class is not compatible with a base class).

- For example, the following code will **print B, C, D** in that order:

Errors and Exceptions

3. Handling Exceptions

```
>>> class B(Exception):
...     pass
...
>>> class C(B):
...     pass
...
>>> class D(C):
...     pass
...
>>> for cls in [B, C, D]:
...     try:
...         raise cls()
...     except D:
...         print("D")
...     except C:
...         print("C")
...     except B:
...         print("B")
...
B
C
D
```

Errors and Exceptions

3. Handling Exceptions

- When an exception occurs, it may have associated values, also known as the *exception's arguments.*

- The presence and types of the arguments depend on the *exception type.*

- The except clause may specify a variable after the exception name.

- The variable is bound to the exception instance which typically has an args attribute that stores the arguments.

- For convenience, builtin exception types define __str__() to print all the arguments without explicitly accessing .args.

Errors and Exceptions

3. Handling Exceptions

```
>>> try:
...     raise Exception('spam', 'eggs')
... except Exception as inst:
...     print(type(inst))    # the exception type
...     print(inst.args)     # arguments stored in .args
...     print(inst)          # __str__ allows args to be printed directly,
...                          # but may be overridden in exception subclasses
...     x, y = inst.args     # unpack args
...     print('x =', x)
...     print('y =', y)
...
<class 'Exception'>
('spam', 'eggs')
('spam', 'eggs')
x = spam
y = eggs
>>>
```

Errors and Exceptions

3. Handling Exceptions

- The exception's `__str__()` output is printed as the last part ('detail') of the message for unhandled exceptions.
- `BaseException` is the common base class of all exceptions. One of its subclasses, `Exception`, is the base class of all the non-fatal exceptions. Exceptions which are not subclasses of `Exception` are not typically handled, because they are used to indicate that the program should terminate. They include `SystemExit` which is raised by `sys.exit()` and `KeyboardInterrupt` which is raised when a user wishes to interrupt the program.
- `Exception` can be used as a wildcard that catches (almost) everything. However, it is good practice to be as specific as possible with the types of exceptions that we intend to handle, and to allow any unexpected exceptions to propagate on.
- The most common pattern for handling `Exception` is to print or log the exception and then re-raise it (allowing a caller to handle the exception as well):

Errors and Exceptions

3. Handling Exceptions

```
>>> import sys
>>>
>>> try:
...     f = open('myfile.txt')
...     s = f.readline()
...     i = int(s.strip())
... except OSError as err:
...     print("OS error:", err)
... except ValueError:
...     print("Could not convert data to an integer.")
... except Exception as err:
...     print(f"Unexpected {err=}, {type(err)=}")
...     raise
```

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
>>>
```