SNS COLLEGE OF ENGINEERING
Kurumbapalayam (Po), Coimbatore – 641 107
An Autonomous Institution
Accredited by NBA – AICTE and Accredited by NAAC – UGC with 'A' Grade
Approved by AICTE, New Delhi & Affiliated to Anna University, Chennai

DEPARTMENT OF CSE (IoT & CYBER SECURITY INCLUDING BLOCKCHAIN TECHNOLOGY)

19IT103 – COMPUTATIONAL THINKING AND PYTHON PROGRAMMING

- **A readable, dynamic, pleasant, flexible, fast and powerful language**

Objective

Files and exception: text files, reading and writing files, format operator; command line arguments, errors and exceptions, handling exceptions, modules, packages; Illustrative programs: word count, copy file, Voter's age validation, Marks range validation (0-100).

.

RECAP

- Python uses C-style string formatting to create new, formatted strings. The "%" operator is used to format a set of variables enclosed in a "tuple" (a fixed size list), together with a format string, which contains normal text together with "argument specifiers", special symbols like "%s" and "%d".
- Command-line arguments are a common way to parameterize execution of programs.
- We can pass the parameters while running the program.
- sys is module that helps to parse the arguments

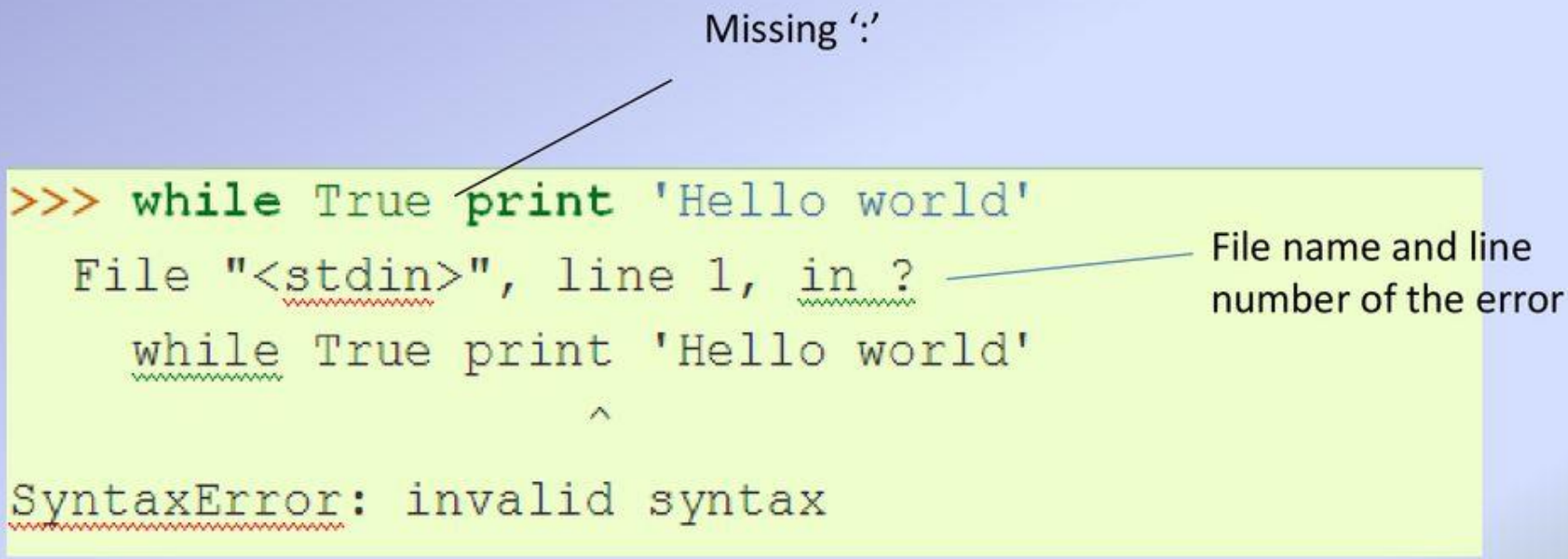
Errors and Exception

We can make certain mistakes while writing a program that lead to errors when we try to run it. A python program terminates as soon as it encounters an unhandled error. These errors can be broadly classified into two classes:

- Syntax errors
- Logical errors (Exceptions)

Syntax Errors

- An error of language resulting from code that does not conform to the syntax of the programming language.



Syntax Error

main.py

```
1   a = 20
2   if (a%2==0)
3     print('even')
4   else:
5     print('odd')
```

Missing ':'

https://cmd-args.kiteit.repl.run

```
> python main.py
  File "main.py", line 2
    if (a%2==0)
              ^
SyntaxError: invalid syntax
>
```

Exceptions

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

```
>>> 10 * (1/0)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ZeroDivisionError: integer division or modulo by zero
>>> 4 + spam*3
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
NameError: name 'spam' is not defined
>>> '2' + 2
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: cannot concatenate 'str' and 'int' objects
```

The types of exceptions in the example are:

- **ZeroDivisionError**
- **NameError**
- **TypeError**

❖ *Built-in Exceptions* lists the built-in exceptions and their meanings.

Exceptions (Logical Errors)

main.py

```
1   import sys
2   cmdargs=sys.argv;
3   value1 = 10
4   value2 = 0
5   result = value1 / value2;
6   print("result:",result)
```

program syntactically correct
But (logically wrong)

https://cmd-args.kiteit.repl.run

```
> python main.py
Traceback (most recent call last):
  File "main.py", line 5, in <module>
    result = value1 / value2;
ZeroDivisionError: division by zero
>
```

Handling Exceptions

- To write programs that handle selected exceptions(try statement) .

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

- The try statement works as follows:
  - the *try clause* (the statement(s) between the try and except) is executed.
  - When no exception occurs in the try clause, no exception handler is executed.
  - When an exception occurs in the try clause, a search for an exception handler is started.

Handling Exceptions(Cont'd)

- A try statement may have more than one except clause to specify handlers for different exceptions.
- An except clause may name multiple exceptions as a parenthesized tuple, for example:

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

Handling Exceptions(Cont'd)

- The try ... except statement has an optional *else clause*.
- Else clause must follow all except clauses.

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

Exception argument

- When an exception occurs, it may have an associated value--exception's *argument*.
- The except clause may specify a variable after the exception name. The two arguments stored in instance.args.
- The exception instance defines ___str__() so the arguments can be printed directly.

```
>>> try:
...     raise Exception('spam', 'eggs')
... except Exception as inst:
...     print type(inst)     # the exception instance
...     print inst.args      # arguments stored in .args
...     print inst           # __str__ allows args to printed directly
...     x, y = inst          # __getitem__ allows args to be unpacked directly
...     print 'x =', x
...     print 'y =', y
...
<type 'exceptions.Exception'>
('spam', 'eggs')
('spam', 'eggs')
x = spam
y = eggs
```

arguments

Handling Exceptions(Cont'd)

- An exception can occur inside functions that are called in the try clause. For example:

```
>>> def this_fails():
...     x = 1/0
...
>>> try:
...     this_fails()
... except ZeroDivisionError as detail:
...     print 'Handling run-time error:', detail
...
Handling run-time error: integer division or modulo by zero
```

Raising Exceptions

- The raise statement allows the programmer to force a specified exception to occur.

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

Raising Exceptions(Cont'd)

- 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 ?
NameError: HiThere
```

User-defined Exceptions

- To create a new exception class to have own exceptions.
- Exceptions should typically be derived from the **Exception** class, either directly or indirectly.

```
>>> class MyError(Exception):
...     def __init__(self, value):
...         self.value = value
...     def __str__(self):
...         return repr(self.value)
...
>>> try:
...     raise MyError(2*2)
... except MyError as e:
...     print 'My exception occurred, value:', e.value
...
My exception occurred, value: 4
>>> raise MyError('oops!')
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
__main__.MyError: 'oops!'
```

User-defined Exceptions(Cont'd)

```
class Error(Exception):
    """Base class for exceptions in this module."""
    pass

class InputError(Error):
    """Exception raised for errors in the input.

    Attributes:
        expr -- input expression in which the error occurred
        msg  -- explanation of the error
    """

    def __init__(self, expr, msg):
        self.expr = expr
        self.msg = msg

class TransitionError(Error):
    """Raised when an operation attempts a state transition that's not
    allowed.

    Attributes:
        prev -- state at beginning of transition
        next -- attempted new state
        msg  -- explanation of why the specific transition is not allowed
    """

    def __init__(self, prev, next, msg):
        self.prev = prev
        self.next = next
        self.msg = msg
```

- Offering a number of attributes that allow different exceptions.

Defining Clean-up Actions

- The try statement has another optional clause, *finally clause*
- A *finally* clause is intended to define 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
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 ?
  File "<stdin>", line 3, in divide
TypeError: unsupported operand type(s) for /: 'str' and 'str'
```

The two exceptions are handled by the except clause

❖A finally clause is executed in any event.
❖A finally *clause* is always executed before leaving the try statement

The TypeError raised by dividing two strings and therefore re-raised after the finally clause has been executed

Predefined Clean-up Actions

- Some objects define standard clean-up actions to be undertaken when the object is no longer needed.

```
for line in open("myfile.txt"):
    print line
```

❖ The problem with this code is that it leaves the file open after the code has finished executing. This is not an issue in simple scripts, but can be a problem for larger applications.

Predefined Clean-up Actions(Cont'd)

- The with statement allows objects like files to be used in a way that ensures they are always cleaned up promptly and correctly.

```
with open("myfile.txt") as f:
    for line in f:
        print line
```

❖ After the statement is executed, the file f is always closed.

Python Built in exceptions

Exception	Cause of Error
`AssertionError`	Raised when an `assert` statement fails.
`AttributeError`	Raised when attribute assignment or reference fails.
`EOFError`	Raised when the `input()` function hits end-of-file condition.
`FloatingPointError`	Raised when a floating point operation fails.
`GeneratorExit`	Raise when a generator's `close()` method is called.
`ImportError`	Raised when the imported module is not found.
`IndexError`	Raised when the index of a sequence is out of range.
`KeyError`	Raised when a key is not found in a dictionary.

Python Built in exceptions

`KeyboardInterrupt`	Raised when the user hits the interrupt key (`Ctrl+C` or `Delete`).
`MemoryError`	Raised when an operation runs out of memory.
`NameError`	Raised when a variable is not found in local or global scope.
`NotImplementedError`	Raised by abstract methods.
`OSError`	Raised when system operation causes system related error.
`OverflowError`	Raised when the result of an arithmetic operation is too large to be represented.
`ReferenceError`	Raised when a weak reference proxy is used to access a garbage collected referent.
`RuntimeError`	Raised when an error does not fall under any other category.

Python Built in exceptions

`StopIteration`	Raised by `next()` function to indicate that there is no further item to be returned by iterator.
`SyntaxError`	Raised by parser when syntax error is encountered.
`IndentationError`	Raised when there is incorrect indentation.
`TabError`	Raised when indentation consists of inconsistent tabs and spaces.
`SystemError`	Raised when interpreter detects internal error.
`SystemExit`	Raised by `sys.exit()` function.
`TypeError`	Raised when a function or operation is applied to an object of incorrect type.

Python Built in exceptions

`UnboundLocalError`	Raised when a reference is made to a local variable in a function or method, but no value has been bound to that variable.
`UnicodeError`	Raised when a Unicode-related encoding or decoding error occurs.
`UnicodeEncodeError`	Raised when a Unicode-related error occurs during encoding.
`UnicodeDecodeError`	Raised when a Unicode-related error occurs during decoding.
`UnicodeTranslateError`	Raised when a Unicode-related error occurs during translating.
`ValueError`	Raised when a function gets an argument of correct type but improper value.
`ZeroDivisionError`	Raised when the second operand of division or modulo operation is zero.

SUMMARY

A python program terminates as soon as it encounters an unhandled error. These errors can be broadly classified into two classes:

- Syntax errors
- Logical errors (Exceptions)


Thank
you