# Errors, Exceptions and Formatting

Lecture 5

CSCI 3351 & CSCI 6651
Dr. Frank Breitinger

# Syntax Errors

- Syntax errors (a.k.a. parsing errors) are perhaps the most common kind of error one encounters.
- Parser repeats the offending line and displays a error message

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

  - File name and line number are printed so you know where to look in case the input came from a script

# Exceptions

- Exceptions are errors that occur during runtime.
  - For example:

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

# Handling Exceptions

- Possible to write programs that handle selected exceptions which is done using Try-Except

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

- First the 'try' clause is executed
- If error occurs, the program switches to the except part
  - Has to match the exception, e.g., ValueError.
    - It is allowed to only write except which then handles all exceptions.
  - If an exception occurs which does not match the exception named, it is passed on to outer try statements;
  - if no handler is found, it is an unhandled exception and execution stops.

# Try-except-else

- The try-except statement has an optional else clause, which must follow all except clauses.
  - It is useful for code that must be executed if the try clause does not raise an exception.

```
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()
```

# Throwing exceptions

- It is also possible to throw own exceptions but we will not address this for now.

# PRINT & FORMAT STRINGS

# Breaking Long Statements into Multiple Lines

- Long statements cannot be viewed on screen without scrolling and cannot be printed without cutting off
- Multiline continuation character (\): Allows to break a statement into multiple lines
  - Example:
    - `print('my first name is',\`
      `first_name)`

# More About Data Output

- print function displays line of output
  - Newline character at end of printed data
  - Special argument end='delimiter' causes print to place delimiter at end of data instead of newline character
- Example:
  - `print("I will have a dollar sign at the end", end='$')`

# More About Data Output

- print function uses space as item separator
  - Special argument sep='delimiter' causes print to use delimiter as item separator
- Example:
  - `print("that", "is", "cool", sep=' -> ')`
  - `print("that", "is", "cool", sep='\t')`

https://docs.python.org/2.0/ref/strings.html

# Special characters

- Special characters appearing in string literal
  - Preceded by backslash (\)
    - Examples: newline (\n), horizontal tab (\t)
  - Treated as commands embedded in string

| Escape Sequence | Meaning |
|---|---|
| *newline* | Ignored |
| \\ | Backslash (\) |
| \' | Single quote (') |
| \" | Double quote (") |
| \a | ASCII Bell (BEL) |
| \b | ASCII Backspace (BS) |
| \f | ASCII Formfeed (FF) |
| \n | ASCII Linefeed (LF) |
| \r | ASCII Carriage Return (CR) |
| \t | ASCII Horizontal Tab (TAB) |
| \v | ASCII Vertical Tab (VT) |
| *ooo* | ASCII character with octal value *ooo* |
| \x*hh*... | ASCII character with hex value *hh*... |

# Formatting numbers overview

- Two ways to format numbers:
  - `Old: '%s %s' % ('one', 'two')`
  - `New: '{} {}'.format('one', 'two')`
- `.format` was introduced in Python2.6
  - If you need backward compatibility with earlier Python, you should use `%`
  - For Python3 and newer you should use `.format`
  - `.format` is more powerful than `%`.
    - Porting `% to .format` is easy but the other way round can be non trivia

# format function

- Can format display of numbers on screen
  - Two arguments:
    - Numeric value to be formatted
    - Format specifier
- Returns string containing formatted number
- Format specifier typically includes precision and data type
  - Can be used to indicate scientific notation, comma separators, and the minimum field width used to display the value
- Example:

```
'0x{:x}'.format(20)
0x14
```

https://pyformat.info

# format function cont'd

- Many ways to format variables; we will only discuss a few possibilities. For more infos see website.

# Formatting Text Examples

```
>>> '{} {} {{hello}}'.format('one', '20')
one 20 {hello}
>>> '{1} {0}'.format('one', '20')
20 one
>>> '{:10}, {:_<10}, {:^10} end'.format('one', '20',
'hello')
one       , 20________,   hello    end
>>> '{:_<10.5}'.format('xylophone')
xylop_____
```

# Formatting Numbers Examples

```
>>> '{:d} {:f}'.format(20, 3.1415926535)
20 3.141593
>>> '{:4d} {:06.2f} {:04d}'.format(20, 3.141592, 14)
 20 003.14 0014
>>> '0x{:02x}, {:+d}, {: d} end'.format(20, 20, -23)
0x14, +20, -23 end
>>> '{1:b}'.format(16, 10)
1010
```

# Strings and Integers

- `ord(char)` - Converts a character into a number
  - `ord('a') is 97`
  - `ord('b') is 98`
  - Uses standard mapping such as ASCII and Unicode
- `char(number)` - Converts a number into a char (string)
  - `chr(99) is 'c'`
  - `chr(100) is 'd'`

# Assignment 3

- Write a program that counts the word and letter frequencies for text files. All details are provided on repl.it