Python Scripting

Samarjit Adhikari
(Samarjit.adhikari@gmail.com)

Which of the following languages, you know?

- C or C++
- Java
- Perl
- Scheme
- Fortran
- Python
- Matlab

Overview

- Running Python and Output
- Data Types
- Input and File I/O
- Control Flow
- Functions
- Modules

Hello World!

- Open a terminal window and type “python”
- If on Windows open a Python IDE like IDLE
- At the prompt type ‘hello world!’

```
>>> 'hello world!'

'hello world!'
```

Python Overview

From *Learning Python, 5th Ed*Mark Lutz

- Programs are composed of modules
- Modules contain statements
- Statements contain expressions
- Expressions create and process objects

The Python Interpreter

- Python is an interpreted language
- The interpreter provides an interactive environment to play with the language
- Results of expressions are printed on the screen

```
>>> 3 + 7
10
>>> 3 < 15
True
>>> 'print me'
'print me'
>>> print 'print me'
print me
>>>
```

The ‘print’ Statement

- Elements separated by commas print with a space between them
- A comma at the end of the statement (print ‘hello’,) will not print a newline character

```
>>> print 'hello'
hello
>>> print 'hello', 'there'
hello there
```

Documentation

The ‘#’ starts a line comment

```
>>> 'this will print'
'this will print'
>>> #'this will not'
>>>
```

Variables

- Are not declared, just assigned
- The variable is created the first time you assign it a value
- Are references to objects
- Type information is with the object, not the reference
- Everything in Python is an object

Everything is an object

- Everything means everything, including functions and classes (more on this later!)
- Data type is a property of the object and not of the variable

```
>>> x = 7
>>> x
7
>>> x = 'hello'
>>> x
'hello'
>>>
```

Numbers: Integers

- Integer – the equivalent of a C long
- Long Integer – an unbounded integer value.

```
>>> 132224
132224
>>> 132323 ** 2
17509376329L
>>>
```

Numbers: Floating Point

- int(x) converts x to an integer
- float(x) converts x to a floating point
- The interpreter shows a lot of digits

```
>>> 1.23232
1.2323200000000001
>>> print 1.23232
1.23232
>>> 1.3E7
13000000.0
>>> int(2.0)
2
>>> float(2)
2.0
```

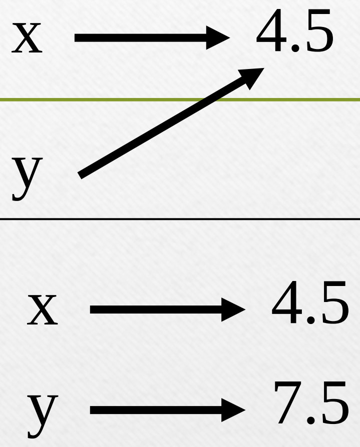
Numbers: Complex

- Built into Python
- Same operations are supported as integer and float

```
>>> x = 3 + 2j
>>> y = -1j
>>> x + y
(3+1j)
>>> x * y
(2-3j)
```

Numbers are immutable

```
>>> x = 4.5
>>> y = x
>>> y += 3
>>> x
4.5
>>> y
7.5
```



x → 4.5

y → 4.5

x → 4.5

y → 7.5

String Literals

- Strings are *immutable*
- There is no char type like in C++ or Java
- + is overloaded to do concatenation

```
>>> x = 'hello'
>>> x = x + ' there'
>>> x
'hello there'
```

String Literals: Many Kinds

- Can use single or double quotes, and three double quotes for a multi-line string

```
>>> 'I am a string'
'I am a string'
>>> "So am I!"
'So am I!'
>>> s = """And me too!
though I am much longer
than the others :)"""
'And me too!\nthough I am much longer\nthan the others :)‘
>>> print s
And me too!
though I am much longer
than the others :)‘
```

String Literals: Many Kinds

- Then why do we require to support both?

(single quote a quote)

String Literals: Many Kinds

Escape	**Meaning**
newline	Ignored (continuation line)
\\	Backslash (stores one \)
\'	Single quote (stores ')
\"	Double quote (stores ")
\a	Bell
\b	Backspace
\f	Form feed
\n	Newline (linefeed)
\r	Carriage return
\t	Horizontal tab
\v	Vertical tab
\x*hh*	Character with hex value *hh* (exactly 2 digits)
ooo	Character with octal value *ooo* (up to 3 digits)
\0	Null: binary 0 character (doesn’t end string)
\N{ id }	Unicode database ID
\u*hhhh*	Unicode character with 16-bit hex value
\U*hhhhhhhh*	Unicode character with 32-bit hex valuea
other	Not an escape (keeps both \ and *other*)

```
>>> 'knight"s', "knight's"
('knight"s', "knight's")
```

```
>>> s = 'a\nb\tc’
>>> s
'a\nb\tc'
>>> print(s)
a
b c
```

String Literals: Raw String

- Sometimes, the special treatment of backslashes for introducing escapes can lead to trouble.
- Instead use raw string

```
>>>myfile = open('C:\new\text.dat', 'w')
-- Wrong
```

```
>>>myfile = open(r'C:\new\text.dat', 'w')
-- Right
```

```
>>> path = r'C:\new\text.dat'
>>> path # Show as Python code
'C:\\new\\text.dat'
>>> print(path) # User-friendly format
C:\new\text.dat
```

Substrings and Methods

```
>>> s = '012345'
>>> s[3]
'3'
>>> s[1:4]
'123'
>>> s[2:]
'2345'
>>> s[:4]
'0123'
>>> s[-2]
'4'
```

- **len**(String) – returns the number of characters in the String

- **str**(Object) – returns a String representation of the Object

```
>>> len(x)
6
>>> str(10.3)
'10.3'
```

String Formatting

- Similar to C’s printf
- <formatted string> % <elements to insert>
- Can usually just use %s for everything, it will convert the object to its String representation.

```
>>> "One, %d, three" % 2
'One, 2, three'
>>> "%d, two, %s" % (1,3)
'1, two, 3'
>>> "%s two %s" % (1, 'three')
'1 two three'
>>>
```

Quick Revisit

- What gets printed?

```
print r"\nwoow"
```

- What gets printed? Assuming python version 2.x

```
print (type(1/2))
```

- What gets printed?

```
print "hello" 'world'
```

```
print "\x48\x49!"
```

```
name = "snow storm"
print "%s" % name[6:8]
```

Lists

- Ordered collection of data
- Data can be of different types
- Lists are *mutable*
- Issues with shared references and mutability
- Same subset operations as Strings

```
>>> x = [1,'hello', (3 + 2j)]
>>> x
[1, 'hello', (3+2j)]
>>> x[2]
(3+2j)
>>> x[0:2]
[1, 'hello']
```

Lists: Modifying Content

- **x[i] = a** reassigns the ith element to the value a
- Since x and y point to the same list object, *both* are changed
- The method **append** also modifies the list

```
>>> x = [1,2,3]
>>> y = x
>>> x[1] = 15
>>> x
[1, 15, 3]
>>> y
[1, 15, 3]
>>> x.append(12)
>>> y
[1, 15, 3, 12]
```

Lists: Modifying Contents

- The method **append** modifies the list and returns **None**
- List addition (**+**) returns a new list

```
>>> x = [1,2,3]
>>> y = x
>>> z = x.append(12)
>>> z == None
True
>>> y
[1, 2, 3, 12]
>>> x = x + [9,10]
>>> x
[1, 2, 3, 12, 9, 10]
>>> y
[1, 2, 3, 12]
>>>
```

Tuples

- Tuples are *immutable* versions of lists
- One strange point is the format to make a tuple with one element:

  ‘,’ is needed to differentiate from the mathematical expression (2)

```
>>> x = (1,2,3)
>>> x[1:]
(2, 3)
>>> y = (2,)
>>> y
(2,)
>>>
```

Dictionaries

- A set of key-value pairs
- Dictionaries are *mutable*

```
>>> d = {1 : 'hello', 'two' : 42, 'blah' : [1,2,3]}
>>> d
{1: 'hello', 'two': 42, 'blah': [1, 2, 3]}
>>> d['blah']
[1, 2, 3]
```

Dictionaries: Add/Modify

- Entries can be changed by assigning to that entry

```
>>> d
{1: 'hello', 'two': 42, 'blah': [1, 2, 3]}
>>> d['two'] = 99
>>> d
{1: 'hello', 'two': 99, 'blah': [1, 2, 3]}
```

- Assigning to a key that does not exist adds an entry

```
>>> d[7] = 'new entry'
>>> d
{1: 'hello', 7: 'new entry', 'two': 99, 'blah': [1, 2, 3]}
```

Dictionaries: Deleting Elements

- The **del** method deletes an element from a dictionary

```
>>> d
{1: 'hello', 2: 'there', 10: 'world'}
>>> del(d[2])
>>> d
{1: 'hello', 10: 'world'}
```

Copying Dictionaries and Lists

- The built-in **list** function will copy a list
- The dictionary has a method called **copy**

```
>>> l1 = [1]
>>> l2 = list(l1)
>>> l1[0] = 22
>>> l1
[22]
>>> l2
[1]
```

```
>>> d = {1 : 10}
>>> d2 = d.copy()
>>> d[1] = 22
>>> d
{1: 22}
>>> d2
{1: 10}
```

Data Type Summary

- Lists, Tuples, and Dictionaries can store any type (including other lists, tuples, and dictionaries!)
- Only lists and dictionaries are mutable
- All variables are references

Data Type Summary

- Integers: 2323, 3234L
- Floating Point: 32.3, 3.1E2
- Complex: 3 + 2j, 1j
- Lists: l = [1,2,3]
- Tuples: t = (1,2,3)
- Dictionaries: d = {‘hello’ : ‘there’, 2 : 15}

Quick Revisit

- What gets printed?

```
kvps  = {"user","bill", "password","hillary"}
print kvps['password']
```

In order to store values in terms of key and value we use what core datatype.
a) List
b) tuple
c) class
d) dictionary

Select all options that print
hello-how-are-you
a) print(‘hello’, ‘how’, ‘are’, ‘you’)
b) print(‘hello’, ‘how’, ‘are’, ‘you’ + ‘-‘ * 4)
c) print(‘hello-‘ + ‘how-are-you’)
d) print(‘hello’ + ‘-‘ + ‘how’ + ‘-‘ + ‘are’ + ‘-‘ + ‘you’)

Input

- The **raw_input**(string) method returns a line of user input as a string
- The parameter is used as a prompt
- The string can be converted by using the conversion methods **int**(string), **float**(string), etc.

Input: Example

```
print "What's your name?"
name = raw_input("> ")

print "What year were you born?"
birthyear = int(raw_input("> "))

print "Hi %s! You are %d years old!" % (name, 2011 - birthyear)
```

```
~: python input.py
What's your name?
> Michael
What year were you born?
>1980
Hi Michael! You are 31 years old!
```

Files: Input

inflobj = open(‘data’, ‘r’)	Open the file ‘data’ for input
S = inflobj.read()	Read whole file into one String
S = inflobj.read(N)	Reads N bytes (N >= 1)
L = inflobj.readlines()	Returns a list of line strings

Files: Output

outflobj = open(‘data’, ‘w’)	Open the file ‘data’ for writing
outflobj.write(S)	Writes the string S to file
outflobj.writelines(L)	Writes each of the strings in list L to file
outflobj.close()	Closes the file

Reading and Writing Files

```
>>> f = open('workfile', 'w')
>>> print f
<open file 'workfile', mode 'w' at 80a0960>
```

mode can be 'r' when the file will only be read, 'w' for only writing (an existing file with the same name will be erased), and 'a' opens the file for appending; any data written to the file is automatically added to the end. 'r+' opens the file for both reading and writing. The mode argument is optional; 'r' will be assumed if it’s omitted.

On Windows, 'b' appended to the mode opens the file in binary mode, so there are also modes like 'rb', 'wb', and 'r+b'. Python on Windows makes a distinction between text and binary files; the end-of-line characters in text files are automatically altered slightly when data is read or written.

Booleans

- 0 and None are false
- Everything else is true
- True and False are aliases for 1 and 0 respectively

Boolean Expressions

- Compound boolean expressions short circuit
- and and or return one of the elements in the expression
- Note that when None is returned the interpreter does not print anything

```
>>> True and False
False
>>> False or True
True
>>> 7 and 14
14
>>> None and 2
>>> None or 2
2
```

Moving to Files

- The interpreter is a good place to try out some code, but what you type is not reusable
- Python code files can be read into the interpreter using the **import** statement

Moving to Files

- In order to be able to find a module called myscripts.py, the interpreter scans the list sys.path of directory names.
- The module must be in one of those directories.

```
>>> import sys
>>> sys.path
['C:\\Python26\\Lib\\idlelib', 'C:\\WINDOWS\\system32\\python26.zip',
'C:\\Python26\\DLLs', 'C:\\Python26\\lib', 'C:\\Python26\\lib\\plat-win',
'C:\\Python26\\lib\\lib-tk', 'C:\\Python26', 'C:\\Python26\\lib\\site-packages']
>>> import myscripts
Traceback (most recent call last):
  File "<pyshell#2>", line 1, in <module>
    import myscripts.py
ImportError: No module named myscripts.py
```

No Braces

- Python uses ***indentation*** instead of braces to determine the scope of expressions
- All lines must be indented the same amount to be part of the scope (or indented more if part of an inner scope)
- This **forces** the programmer to use proper indentation since the indenting is part of the program!

If Statements

```
import math
x = 30
if x <= 15 :
    y = x + 15
elif x <= 30  :
    y = x + 30
else :
    y = x
print ‘y = ‘,
print math.sin(y)
```

In file ifstatement.py

```
>>> import ifstatement
y =  0.999911860107
>>>
```

In interpreter

While Loops

```
x = 1
while x < 10 :
    print x
    x = x + 1
```

In whileloop.py

```
>>> import whileloop
1
2
3
4
5
6
7
8
9
>>>
```

In interpreter

Loop Control Statements

break	Jumps out of the closest enclosing loop
continue	Jumps to the top of the closest enclosing loop
pass	Does nothing, empty statement placeholder

The Loop Else Clause

- The optional **else** clause runs only if the loop exits normally (not by break)

```
x = 1

while x < 3 :
    print x
    x = x + 1
else:
    print 'hello'
```

In whileelse.py

```
~: python whileelse.py
1
2
hello
```

Run from the command line

The Loop Else Clause

```
x = 1
while x < 5 :
    print x
    x = x + 1
    break
else :
    print 'i got here'
```

```
~: python whileelse2.py
1
```

whileelse2.py

For Loops

- Similar to perl for loops, iterating through a list of values

forloop1.py

```
for x in [1,7,13,2] :
    print x
```

```
~: python forloop1.py
1
7
13
2
```

forloop2.py

```
for x in range(5) :
    print x
```

```
~: python forloop2.py
0
1
2
3
4
```

range(N) generates a list of numbers [0,1, …, n-1]

Quick Revisit

- What gets printed?

```
nums = set([1,1,2,3,3,3,4])
print len(nums)
```

```
x = ['ab', 'cd']
for i in x:
    i.upper()
print(x)
```

```
x = ['ab', 'cd']
for i in x:
    x.append(i.upper())
print(x)
```

```
i = 1
while True:
    if i%3 == 0:
        break
    print(i)
    i + = 1
```

```
x = 123
for i in x:
    print(i)
```

```
d = {0: 'a', 1: 'b', 2: 'c'}
for i in d:
    print(i)
```

```
d = {0: 'a', 1: 'b', 2: 'c'}
for x, y in d.items():
    print(x, y)
```

Function Basics

```
def max(x,y) :
    if x < y :
        return x
    else :
        return y
```

functionbasics.py

```
>>> import functionbasics
>>> max(3,5)
5
>>> max('hello', 'there')
'there'
>>> max(3, 'hello')
'hello'
```

Functions are first class objects

- Can be assigned to a variable
- Can be passed as a parameter
- Can be returned from a function
- Functions are treated like any other variable in Python, the **def** statement simply assigns a function to a variable

Function names are like any variable

- Functions are objects
- The same reference rules hold for them as for other objects

```
>>> x = 10
>>> x
10
>>> def x () :
...     print 'hello'
>>> x
<function x at 0x619f0>
>>> x()
hello
>>> x = 'blah'
>>> x
'blah'
```

Functions as Parameters

```
def foo(f, a) :
    return f(a)

def bar(x) :
    return x * x
```

funcasparam.py

```
>>> from funcasparam import *
>>> foo(bar, 3)
9
```

Note that the function foo takes two parameters and applies the first as a function with the second as its parameter

Functional Programming?

• **Object-oriented** programs manipulate collections of objects. Objects have internal state and support methods that query or modify this internal state in some way. Smalltalk and Java are object-oriented languages.

C++ and Python are languages that support object-oriented programming, but don’t force the use of object-oriented features.

• **Functional programming** decomposes a problem into a set of functions. Ideally, functions only take inputs and produce outputs, and don’t have any internal state that affects the output produced for a given input.

Well-known functional languages include the ML family (Standard ML, OCaml, and other variants) and
Haskell

Functional Programming…

Generator expressions and list comprehensions

Two common operations on an iterator’s output are
1) performing some operation for every element, 2) selecting a subset of elements that meet some condition.
For example, given a list of strings, you might want to strip off trailing whitespace from each line or extract all the strings containing a given substring.

List comprehensions and generator expressions (short form: “listcomps” and “genexps”) are a concise notation for such operations, borrowed from the functional programming language Haskell.

```
line_list = [' line 1\n', 'line 2 \n', ...]
# Generator expression -- returns iterator
stripped_iter = (line.strip() for line in line_list)

# List comprehension -- returns list
stripped_list = [line.strip() for line in line_list]
```

Built-in functions

map(func,seq) – for all i, applies func(seq[i]) and returns the corresponding sequence of the calculated results.

```
def double(x):
    return 2*x
```

highorder.py

```
>>> from highorder import *
>>> lst = range(10)
>>> lst
[0,1,2,3,4,5,6,7,8,9]
>>> map(double,lst)
[0,2,4,6,8,10,12,14,16,18]
```

Built-in functions

filter(boolfunc,seq) – returns a sequence containing all those items in seq for which boolfunc is True.

```
def even(x):
    return ((x%2 == 0)
```

highorder.py

```
>>> from highorder import *
>>> lst = range(10)
>>> lst
[0,1,2,3,4,5,6,7,8,9]
>>> filter(even,lst)
[0,2,4,6,8]
```

Built-in functions

reduce(func,seq) – applies func to the items of seq, from left to right, two-at-time, to reduce the seq to a single value.

```
def plus(x,y):
    return (x + y)
```

highorder.py

```
>>> from highorder import *
>>> lst = [‘h’,’e’,’l’,’l’,’o’]
>>> reduce(plus,lst)
‘hello’
```

Functions Inside Functions

- Since they are like any other object, you can have functions inside functions

```
def foo (x,y) :
    def bar (z) :
        return z * 2
    return bar(x) + y
```

```
>>> from funcinfunc import *
>>> foo(2,3)
7
```

funcinfunc.py

Functions Returning Functions

```
def foo (x) :
    def bar(y) :
        return x + y
    return bar
# main
f = foo(3)
print f
print f(2)
```

```
~: python funcreturnfunc.py
<function bar at 0x612b0>
5
```

funcreturnfunc.py

Parameters: Defaults

- Parameters can be assigned default values
- They are overridden if a parameter is given for them
- The type of the default doesn’t limit the type of a parameter

```
>>> def foo(x = 3) :
...     print x
...
>>> foo()
3
>>> foo(10)
10
>>> foo('hello')
hello
```

Parameters: Named

- Call by name
- Any positional arguments must come before named ones in a call

```
>>> def foo (a,b,c) :
...     print a, b, c
...
>>> foo(c = 10, a = 2, b = 14)
2 14 10
>>> foo(3, c = 2, b = 19)
3 19 2
```

Anonymous Functions

- A lambda expression returns a function object
- The body can only be a simple expression, not complex statements

```
>>> f = lambda x,y : x + y
>>> f(2,3)
5
>>> lst = ['one', lambda x : x * x, 3]
>>> lst[1](4)
16
```

Modules

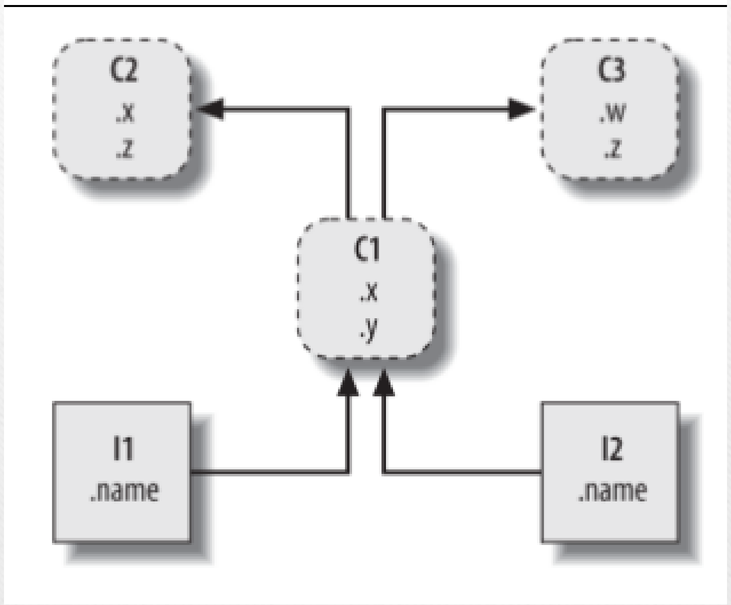
- The highest level structure of Python
- Each file with the py suffix is a module
- Each module has its own namespace

Modules: Imports

import mymodule	Brings all elements of mymodule in, but must refer to as mymodule.<elem>
from mymodule import x	Imports x from mymodule right into this namespace
from mymodule import *	Imports all elements of mymodule into this namespace

Inheritance

Resolving I2.w : it will search the linked objects in order of I2, C1, C2, C3 and stop at the first attached w it finds (or raise an error if w isn't found at all). In this case, w won't be found until C3 is searched because it appears only in that object. In other words, I2.w resolves to C3.w by virtue of the automatic search



- I1.x and I2.x both find x in C1 and stop because C1 is lower than C2
- I1.y and I2.y both find y in C1 because that's the only place y appears.
- I1.z and I2.z both find z in C2 because C2 is further to the left than C3 .
- I2.name finds name in I2 without climbing the tree at all.

Error and Exceptions

Handling Exceptions

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

- 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 statement.
- 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.

Error and Exceptions

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

An except clause may name multiple exceptions as a parenthesized tuple

Note that the parentheses around this tuple are required,because except ValueError, e: was the syntax used for what is normally written as except ValueError as e: in modern Python .

The old syntax is still supported for backwards compatibility. This means exceptRuntimeError, TypeError is not equivalent to except (RuntimeError, TypeError): but to except RuntimeError as TypeError: which is not definitely what we want.

Error and Exceptions

The last except clause may omit the exception name(s), to serve as a wildcard

```
import sys
try:
    f = open('myfile.txt')
    s = f.readline()
    i = int(s.strip())
except IOError as e:
    print "I/O error({0}): {1}".format(e.errno, e.strerror)
except ValueError:
    print "Could not convert data to an integer."
except:
    print "Unexpected error:", sys.exc_info()[0]
    raise
```

Error and 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 IOError:
        print 'cannot open', arg
    else:
        print arg, 'has', len(f.readlines()), 'lines'
        f.close()
```

Error and Exceptions

The except clause may specify a variable after the exception name (or tuple). The variable is bound to an exception instance with the arguments stored in instance.args. For convenience, the exception instance defines __str__() so the arguments can be printed directly without having to reference .args.

One may also instantiate an exception first before raising it and add any attributes to it as desired.

```
>>> 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 be printed
directly
...     x, y = inst.args
...     print 'x =', x
...     print 'y =', y
...
<type 'exceptions.Exception'>
('spam', 'eggs')
('spam', 'eggs')
x = spam
y = eggs
```

Error and Exceptions

User-defined Exceptions

Exceptions should typically be derived from the Exception class, either directly or indirectly. For example:

```
>>> 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 <module>
__main__.MyError: 'oops!'
```

Error and Exceptions

Defining Clean-up Actions and predefined clean up action

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!
KeyboardInterrupt
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
```

A finally clause is always executed before leaving the try statement, whether an exception has occurred or not

Error and Exceptions

predefined clean up action

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

The problem with this code is that it leaves the file open. The with statement mentioned below 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, even if a problem was encountered while processing the lines.

Regular Expression

^	Matches the beginning of a line
$	Matches the end of the line
.	Matches any character
\s	Matches whitespace
\S	Matches any non-whitespace character
*	Repeats a character zero or more times
*?	Repeats a character zero or more times (non-greedy)
+	Repeats a chracter one or more times
+?	Repeats a character one or more times (non-greedy)
[aeiou]	Matches a single character in the listed set
[^XYZ]	Matches a single character not in the listed set
[a-z0-9]	The set of characters can include a range
(	Indicates where string extraction is to start
)	Indicates where string extraction is to end

Regular Expression

- Before you can use regular expressions in your program, you must import the library using "import re"
- You can use re.search() to see if a string matches a regular expression similar to using the find() method for strings
- You can use re.findall() to extract portions of a string that match your regular expression similar to a combination of find() and slicing: var[5:10]

Regular Expression

Using re.search( ) like find( )

```
hand = open('mbox-short.txt')
for line in hand:
    line = line.rstrip()
    if line.find('From:') >= 0:
        print line
```

```
import re

hand = open('mbox-short.txt')
for line in hand:
    line = line.rstrip()
    if re.search('From:', line) :
        print line
```

Regular Expression

Using re.search() like startswith()

```
hand = open('mbox-short.txt')
for line in hand:
    line = line.rstrip()
    if line.startswith('From:') >= 0:
        print line
```

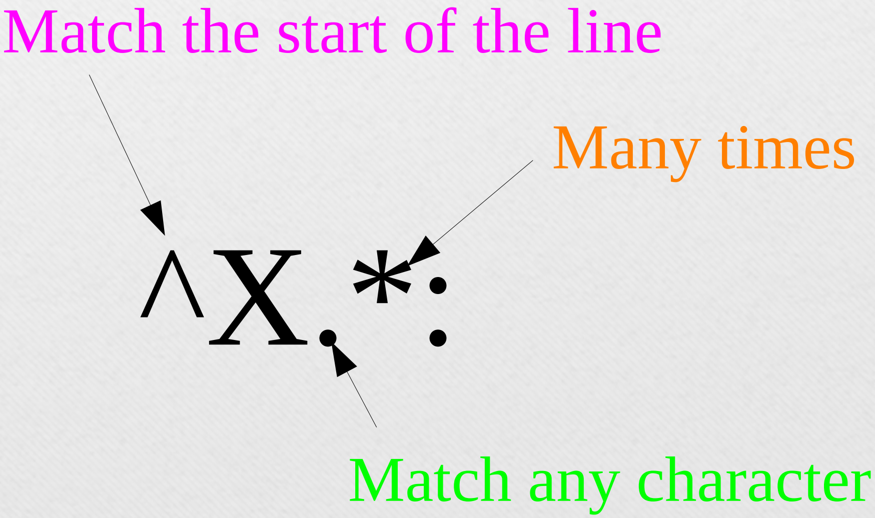
```
import re

hand = open('mbox-short.txt')
for line in hand:
    line = line.rstrip()
    if re.search('^From:', line) :
        print line
```

Regular Expression

Wild-Card Characters (*)

- The dot character matches any character
- If you add the asterisk character, the character is "any number of times"

```
X-Sieve: CMU Sieve 2.3
X-DSPAM-Result: Innocent
X-DSPAM-Confidence: 0.8475
X-Content-Type-Message-Body:
text/plain
```



Match the start of the line

Many times

^X.*:

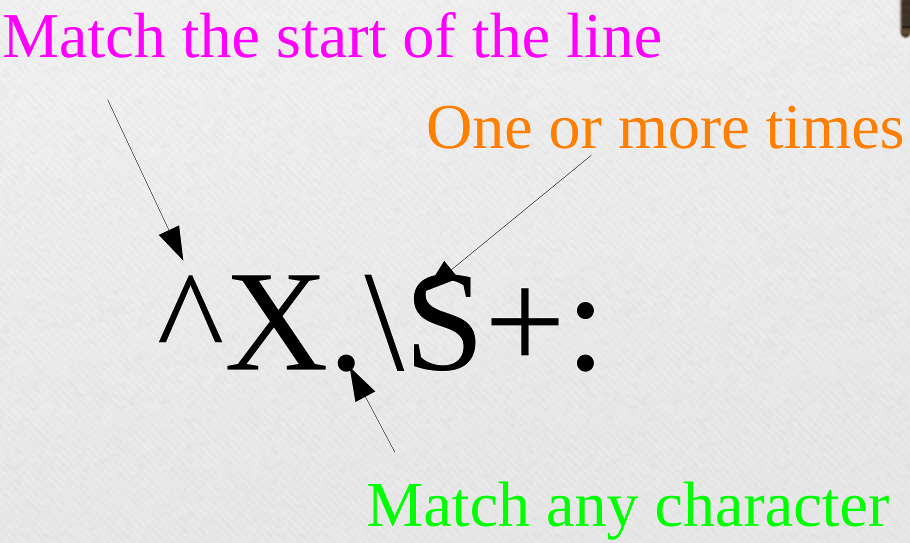
Match any character

Regular Expression

Wild-Card Characters (+)

- The + character matches one or more characters

```
X-Sieve: CMU Sieve 2.3
X-DSPAM-Result: Innocent
X-DSPAM-Confidence: 0.8475
X-Content-Type-Message-Body:
text/plain
```



Match the start of the line

One or more times

^X.\S+:

Match any character

Regular Expression

Discuss

- match() Determine if the RE matches at the beginning of the string.
- search() Scan through a string, looking for any location where this RE matches.
- findall() Find all substrings where the RE matches, and returns them as a list.
- finditer() Find all substrings where the RE matches, and returns them as an iterator.
- group() Return the string matched by the RE
- start() Return the starting position of the match
- end() Return the ending position of the match
- span() Return a tuple containing the (start, end) positions of the match

Regular Expression

Discuss

- >>> m = p.match('tempo')
- >>> m.group()
- 'tempo'
- >>> m.start(), m.end()
- (0, 5)
- >>> m.span()
- (0, 5)

```
>>> import re
>>> p = re.compile('[a-z]+')
>>> print p.match('::: message')
None
>>> m = p.search('::: message'); print m
<_sre.SRE_Match object at 0x...>
>>> m.group()
'message'
>>> m.span()
(4, 11)
```

THANK YOU