

An Introduction to Python

Phil Spector

Statistical Computing Facility
Department of Statistics
University of California, Berkeley

Perl vs. Python

- Perl has a much larger user base.
- Perl has more modules available.
- Facilities for multidimensional arrays and object orientation were grafted on to Perl, but are built in from the start in Python.
- Python's learning curve is far shorter than Perl's
- Python is more fun than Perl

Core Python Concepts

- small basic language
- modules (with private namespaces) for added functionality
the `import` statement provides access to modules
- true object oriented language
- strong typing
- exception handling
try/except clause can trap any error
- indentation is important

Invoking python

- Type `python` at the command line – starts an interactive session – type `control-D` to end

- Type `python programname`

- If the first line of your program is

```
#!/usr/bin/env python
```

and the program is made executable with the `chmod +x` command, you can invoke your program through its name

Getting Help in Python

- Online documentation may be installed along with Python.
- All documentation can be found at <http://python.org> .
- In an interactive Python session, the `help` function will provide documentation for most Python functions.
- To view all available methods for an object, or all of the functions defined within a module, use the `dir` function.

Functions versus Methods

Most objects you create in Python will have a set of *methods* associated with them. Instead of passing the object as an argument to a function, you “invoke” the method on the object by following the object name with a period (.) and the function call.

For example, to count the number of times the letter 'x' appears in a string, you could use the following:

```
>>> str = 'xabxyz'  
>>> str.count('x')  
2
```

You can see the names of all possible methods in an interactive session using the `dir` command.

Strings in Python

Strings are a simple form of a Python *sequence* – they are stored as a collection of individual characters.

There are several ways to initialize strings:

- single (') or double (") quotes
- triple quotes (''' or """) – allows embedded newlines
- raw strings (r'...' or r"...") – ignores special characters
- unicode strings (u'...' or u"...") – supports Unicode (multiple byte) characters

Special Characters

Sequence	Meaning	Sequence	Meaning
<code>\</code>	continuation	<code>\\</code>	literal backslash
<code>\'</code>	single quote	<code>\"</code>	double quote
<code>\a</code>	bell	<code>\b</code>	backspace
<code>\e</code>	escape character	<code>\0</code>	null terminator
<code>\n</code>	newline	<code>\t</code>	horizontal tab
<code>\f</code>	form feed	<code>\r</code>	carriage return
<code>\0XX</code>	octal character XX	<code>\xXX</code>	hexadecimal value XX

Use raw strings to treat these characters literally.

String Operators and Functions

- `+` – concatenation
- `*` – repetition
- `[i]` – subscripting (zero-based; negative subscripts count from the end)
- `[i:j]` – slice from *i*-th character to one before the *j*-th
(length of the slice is $j - i$)
- `[i:]` – slice from *i*-th character to the end
- `[:i]` – slice from the first character to one before the *i*-th
- `len(string)` – returns number of characters in *string*

String Methods

Name	Purpose
<code>join</code>	Insert a string between each element of a sequence
<code>split</code>	Create a list from “words” in a string
<code>splitlines</code>	Create a list from lines in a string
<code>count</code>	Count the number of occurrences of substring
<code>find</code>	Return the lowest index where substring is found
<code>index</code>	Like <code>find</code> , but raises <code>ValueError</code> if not found
<code>rfind</code>	Return the highest index where substring if found
<code>rindex</code>	Like <code>rfind</code> , but raises <code>ValueError</code> if not found
<code>center</code>	Centers a string in a given width

String Methods (continued)

Name	Purpose
<code>ljust</code>	Left justifies a string
<code>rstrip</code>	Removes leading whitespace
<code>rjust</code>	Right justifies a string
<code>rstrip</code>	Removes trailing whitespace
<code>strip</code>	Removes leading and trailing whitespace
<code>capitalize</code>	Capitalize the first letter of the string
<code>lower</code>	Make all characters lower case
<code>swapcase</code>	Change upper to lower and lower to upper
<code>title</code>	Capitalize the first letter of each word in the string
<code>upper</code>	Make all characters upper case

Numbers in Python

- integers - ordinary integers with the default range of the computer
initialize without decimal point
- longs - “infinite” precision integers – immune to overflow
initialize without decimal point and a trailing “L”
- float - double precision floating point numbers
initialize using decimal point
- complex - double precision complex numbers
initialize as $a + bj$

Hexadecimal constants can be entered by using a leading 0X or 0x;
octal constants use a leading 0.

Operators and Functions for Numbers

- Usual math operators: + - * / %
- Exponentiation: **
- Bit shift: << >>
- Core functions: `abs`, `round`, `divmod`
many more in `math` module

If both operands of the division operator are integers, Python uses integer arithmetic. To insure floating point arithmetic, use a decimal point or the `float` function on at least one of the operands.

Type Conversion

When Python encounters an object which is not of the appropriate type for an operation, it raises a `TypeError` exception. The usual solution is to convert the object in question with one of the following conversion functions:

- `int` - integer
- `long` - long
- `float` - float
- `complex` - complex
- `str` - converts anything to a string

Sequence Types

We've already seen the simplest sequence type, strings. The other builtin sequence types in Python are lists, tuples and dictionaries. Python makes a distinction between mutable sequences (which can be modified in place) and immutable sequences (which can only be modified by replacement): lists and dictionaries are mutable, while strings and tuples are immutable.

Lists are an all-purpose “container” object which can contain any other object (including other sequence objects). Tuples are like lists, but immutable. Dictionaries are like lists, but are indexed by arbitrary objects, instead of consecutive integers.

The subscripting and slicing operations presented for strings also work for other sequence objects, as does the `len` function.

Sequence Elements

- Lists - use square brackets (`[]`)

Empty list: `x = []`

List with elements: `x = [1,2,"dog","cat",abs]`

Access using square brackets: `print x[2]`

- Tuples - use parentheses (`()`)

Empty tuple: `x = ()`

Tuple with elements: `x = (1,2,"dog","cat",abs)`

Tuple with a single element: `x = (7,)`

Access using square brackets: `print x[2]`

- Dictionary - use curly braces (`{ }`)

Empty dictionary: `x = {}`

Dictionary with elements: `x = {"dog":"Fido","cat":"Mittens"}`

Access using square brackets: `print x["cat"]`

Nesting of Sequence Types

Sequence types can be as deeply nested as necessary. This makes it very easy to store complex data structures in basic Python objects:

```
nestlist = [1,2,"dog","cat", (20,30,40),  
            {"one":("uno",1), "two":("dos",2), "three":("tres",3)}]  
print nestlist[5]["one"][0]      #prints uno  
nestlist[1] = 14                  #ok - lists are mutable  
nestlist[4][2] = "something"     #fails - tuples are immutable  
nestlist[4] = "something"        #ok to replace whole tuple
```

The individual elements of lists and dictionaries can be modified in place, but this is not true for strings and tuples.

Indexing and Slicing

In addition to extracting parts of lists through subscripting and slicing, you can also modify parts of lists (in place), by referring to a list slice on the left hand side of the equal sign:

```
>>> x = [1,2,3,4,5,6,7,8,9,10]
>>> x[3:5]
[4, 5]
>>> x[3:5] = [40,50,60]
>>> x
[1, 2, 3, 40, 50, 60, 6, 7, 8, 9, 10]
```

Note that the replacement slice can be of a different size.

Insert arbitrary elements into a list using a slice of size zero:

```
>>> x = [1,2,3,4,5,6,7,8,9,10]
>>> x[4:4] = [10,20,30]
>>> x
[1, 2, 3, 4, 10, 20, 30, 5, 6, 7, 8, 9, 10]
```

List Operators

Lists support concatenation and repetition like strings, but to concatenate an element to the end of a list, that element must be made into a list.

`[1,2,3] + 4` results in a `TypeError`, but

`[1,2,3] + [4]` yields a list with four elements.

Similarly for repetition

`0 * 10` results in the integer 0, but

`[0] * 10` results in a list containing ten zeroes.

The `in` operator provides a fast way to tell if something is an element of a list. For example, to find unique values in a list:

```
unique = []
for e in thelist:
    if e not in unique:
        unique = unique + [e]
```

List Methods

Name	Purpose
<code>append</code>	Adds a single element to a list
<code>count</code>	Counts how many times an element appears
<code>extend</code>	Adds multiple elements to a list
<code>index</code>	Returns lowest index of an element in a list
<code>insert</code>	Inserts an object into a list
<code>pop</code>	Returns and removes first element of a list
<code>remove</code>	Removes first occurrence of an element from a list
<code>reverse</code>	Reverses a list in place
<code>sort</code>	Sorts a list in place

Notice that joining together the elements of a list into a string is done with the `join` method for strings.

Sorting Lists in Python

The `sort` method for lists accepts an optional function argument which defines how you want the elements of the list sorted. This function should accept two arguments and return -1 if the first is less than the second, 0 if they are equal, and 1 if the first is greater than the second.

Suppose we wish to sort words disregarding case. We could define the following function, and pass it to `sort`:

```
>>> def cmpcase(a,b):
...     return cmp(a.lower(),b.lower())
...
>>> names = ['Bill','fred','Tom','susie']
>>> names.sort()
>>> names
['Bill', 'Tom', 'fred', 'susie']
>>> names.sort(cmpcase)
>>> names
['Bill', 'fred', 'susie', 'Tom']
```

Dictionaries

Dictionaries are very convenient because it's often easier to associate a string with a piece of data than remember its position in an array. In addition, the keys of a Python dictionary can be any immutable Python object, not just strings.

The following methods are provided for dictionaries:

Name	Purpose
<code>clear</code>	remove all keys and values
<code>get</code>	access values through key with default
<code>has_key</code>	tests for presence of key
<code>keys</code>	returns all keys
<code>values</code>	returns all values

Using Dictionaries for Counting

Since it is an error to refer to a non-existent key, care must be taken when creating a dictionary. Suppose we wish to count the number of times different words appear in a document.

1. Use exceptions

```
try:
    counts[word] = counts[word] + 1
except KeyError:
    counts[word] = 1
```

2. Check with `has_key`

```
if counts.has_key(word):
    counts[word] = counts[word] + 1
else:
    counts[word] = 1
```

3. Use `get`

```
counts[word] = counts.get(word,0) + 1
```

Printing

While the `print` statement accepts any Python object, more control over printed output can be achieved by using formatting strings combined with the “%” operator.

A formatting string contains one or more %-codes, indicating how corresponding elements (in a tuple on the right hand side of the % operator) will be printed. This table shows the possible codes:

Code	Meaning	Code	Meaning
<code>d</code> or <code>i</code>	Decimal Integer	<code>e</code> or <code>E</code>	Exponential Notation
<code>u</code>	Unsigned Integer	<code>g</code> or <code>G</code>	“Optimal” Notation
<code>o</code>	Octal Integer	<code>s</code>	Display as string
<code>h</code> or <code>H</code>	Hexadecimal Integer	<code>c</code>	Single character
<code>f</code>	Floating Point Number	<code>%</code>	Literal percent sign

Examples of Formatting

Field widths can be specified after the % sign of a code:

```
>>> animal = 'chicken'
>>> print '%20s' % animal
           chicken
```

With floating point arguments, the number of decimal places can be specified:

```
>>> x = 7. / 3.
>>> print x
2.333333333333
>>> print '%5.2f' % x
  2.33
```

When formatting more than one item, use a tuple, not a list.

```
>>> print 'Animal name: %s  Number: %5.2f' % (animal,x)
Animal name: chicken  Number:  2.33
```

The result of these operations is a string

```
>>> msg = 'Animal name: %s  Number: %5.2f' % (animal,x)
>>> msg
'Animal name: chicken  Number:  2.33'
```

File Objects

The `open` function returns a file object, which can later be manipulated by a variety of methods. This function takes two arguments: the name of the file to be opened, and a string representing the mode. The possible modes are:

String	Meaning
<code>r</code>	Open file for reading; file must exist.
<code>w</code>	Open file for writing; will be created if it doesn't exist
<code>a</code>	Open file for appending; will be created if it doesn't exist
<code>r+</code>	Open file for reading and writing; contents are not destroyed
<code>w+</code>	Open file for reading and writing; contents are destroyed
<code>a+</code>	Open file for reading and writing; contents are not destroyed

By default, files are opened with mode `"r"`. A `'b'` can be appended to the mode to indicate a binary file.

Using File Objects: Reading

Suppose we wish to read the contents of a file called "mydata". First, create the appropriate file object.

```
try:
    f = open('mydata', 'r')
except IOError:
    print "Couldn't open mydata"
    sys.exit(1)
```

Note the `try/except` block; every call to `open` should be in such a block. Once the file is opened, you can iterate over each line in the file with a for loop, or use one of the following methods: available:

- `readline` - reads the next line of the file
- `readlines` - reads an entire file into a list - one line per element
- `read` - reads a file into a string.

Reading from a File: Example

Suppose we have a file with one number on each line, and we want to add together all the numbers:

```
try:
```

```
    f = open('numbers', 'r')
```

```
except IOError:
```

```
    print "Couldn't open numbers"
```

```
    sys.exit(1)
```

```
total = 0                                # initialize
```

```
for line in f:
```

```
    line = line[:-1]                      # removes newline
```

```
    total = total + int(line)             # type conversion!
```

```
print 'total=%d' % total
```

Using File Objects: Writing

If a file is opened with a mode of 'w' or 'a' the following methods can be used to write to the file:

- `write` - writes its argument to the specified file
- `writelines` - writes each element of a list to the specified file

These methods do not automatically add a newline to the file. The `print` statement automatically adds a newline, and can be used with file objects using the syntax:

```
print >> fileobject, string-to-be-printed
```

This makes it especially easy to change a program that writes to standard output to one that writes to a file.

Standard File Objects

Each time you invoke Python, it automatically creates three file objects, found in the `sys` module, representing standard input (`sys.stdin`), standard output (`sys.stdout`) and standard error (`sys.stderr`).

These can be used like any other file object.

For example, to write an error message to standard error, you could use:

```
print >> sys.stderr, 'Here is an error message'
```

File Objects and Object Oriented Programming

Although they are referred to as file objects, any object which provides the appropriate methods can be treated as a file, making it very easy to modify programs to use different sources. Some of the functions in Python which can provide file-like objects include

- `os.popen` – pipes (shell command input and output)
- `urllib.urlopen` – remote files specified as URLs
- `StringIO.StringIO` – treats a string like a file
- `gzip.GzipFile` – reads compressed files directly

Assignment Statements

To assign a value to a variable, put the name of the variable on the left hand side of an equals sign (=), and the value to be assigned on the right hand side:

```
x = 7
names = ['joe', 'fred', 'sam']
y = x
```

Python allows multiple objects to be set to the same value with a chained assignment statement:

```
i = j = k = 0
```

Furthermore, multiple objects can be assigned in one statement using unrolling:

```
name = ['john', 'smith']
first, last = name
x, y, z = 10, 20, 30
```

A Caution About List Assignments

When you perform an assignment, Python doesn't copy values – it just makes one variable a reference to another. It only does the actual copy when the original variable is overwritten or destroyed. For immutable objects, this creates no surprises. But notice what happens when we change part of a mutable object that's been assigned to another variable:

```
>>> breakfast = ['spam', 'spam', 'sausage', 'spam']
>>> meal = breakfast
>>> breakfast[1] = 'beans'
>>> breakfast
['spam', 'beans', 'sausage', 'spam']
>>> meal
['spam', 'beans', 'sausage', 'spam']
```

Even though we didn't explicitly reference `meal`, some of its values were modified.

True Copy for List Assignments

To avoid this behaviour either assign a complete slice of the list:

```
meal = breakfast[:]
```

or use the `copy` function of the `copy` module:

```
import copy
meal = copy.copy(breakfast)
```

If the original variable is overwritten, a true copy is made:

```
>>> breakfast = ['spam', 'spam', 'sausage', 'spam']
>>> meal = breakfast
>>> breakfast = ['eggs', 'bacon', 'beans', 'spam']
>>> meal
['spam', 'spam', 'sausage', 'spam']
```

You can use the `is` operator to test if two things are actually references to the same object.

Comparison Operators

Python provides the following comparison operators for constructing logical tests:

Operator	Tests for	Operator	Tests for
<code>==</code>	Equality	<code>!=</code>	Non-equality
<code>></code>	Greater than	<code><</code>	Less than
<code>>=</code>	Greater than or equal	<code><=</code>	Less than or equal
<code>in</code>	Membership in sequence	<code>is</code>	Equivalence
<code>not in</code>	Lack of membership	<code>not is</code>	Non-equivalence

Logical expressions can be combined using `and` or `or`.

You can treat logical expressions as integers (`True = 1`, `False = 0`).

Truth and Falsehood

- Logical comparisons – boolean values `True` or `False`
- Numeric values – false if 0, true otherwise.
- Sequence objects – false if they contain no items, true otherwise.
- Special values:
 - `None` – “null value” always false
 - `True` and `False` – boolean values with obvious meanings

Indentation

People who start programming in Python are often surprised that indentation, which is mostly cosmetic in most languages, actually determines the structure of your Python program. Indentation is very useful for several reasons:

1. Just looking at your program gives you an excellent idea of what its structure is, and you're never deceived by improper indentation, since it will generate a syntax error.
2. Since everyone has to indent, other people's programs are generally easier to read and understand.
3. Most experienced programmers agree that good indentation is useful, but requires too much discipline. In Python, you're guaranteed to develop good indentation practices.

Many editors provide facilities for automatic and consistent indentation (`emacs`, `vim`, `bbedit`, etc.). The majority of indentation problems arise from using more than one editor to edit the same program.

if statement

The `if/elif/else` statement is the basic tool for conditional execution in Python. The form is:

```
if expression:
    statement(s)
elif expression:
    statement(s)
elif expression:
    statement(s)
    . . .
else:
    statements
```

The `elif` and `else` clauses are optional.

The colon (`:`) after the expression is required, and the `statement(s)` following the `if` statement must be consistently indented.

You must have at least one statement after an `if` statement; you can use the keyword `pass` to indicate “do nothing”.

The for loop

The `for` loop is the basic construct in Python for iterating over an object such as any sequence or a file object. The syntax is:

```
for var in sequence:
    statements
else:
    statements
```

The colon (`:`) is required, as is proper indentation.

`var` represents a copy of each element of `sequence` in turn, and is local to the `for` loop. The `else` is optional; the statements associated with it are executed if the `for` loop runs to completion.

When you iterate over a dictionary, you're actually iterating over that dictionary's keys.

The range function

Although the `for` loop is useful for iterating over each element of a sequence, there are many tasks which involve more than one sequence, or which need to change the elements of the sequence, and the `for` loop can not handle these tasks. In cases like this, the `range` function can be used to generate a set of indices for subscripting the sequence object(s).

The `range` function accepts between one and three arguments.

With one argument it returns a sequence from zero to one less than its argument. With two arguments, it returns a sequence from the first argument to one less than the second; an optional third argument specifies an increment for the sequence.

If the sequence which would be generated is very long, the `xrange` function can be used. Instead of generating the list in memory, it returns the next element in the sequence each time it is called.

Examples of the range function

Suppose we have two lists, `prices` and `taxes`, and we wish to create a list called `total` which contains the sum of each element in the two arrays.

```
total = len(prices) * [0]
for i in range(len(prices)):
    total[i] = prices[i] + taxes[i]
```

The `range` function can be used when you need to modify elements of a mutable sequence:

```
for i in range(len(x)):
    if x[i] < 0:
        x[i] = 0
```

List Comprehensions

As an alternative to loops, python provides list comprehensions, which have the following basic form:

```
[expr for var in seq]
```

which would return a list, the same length as `seq`, with the result of evaluating `expr` with each value in `seq`. For example, to create a sequence of squares of the numbers from 1 to 5, we could use

```
>>> [i*i for i in range(0,6)]  
[0, 1, 4, 9, 16, 25]
```

List comprehensions also allow `if` statements:

```
>>> [i*i for i in range(0,6) if i % 2 == 1]  
[1, 9, 25]
```

List comprehensions also support multiple `for` statements, returning expressions with all combinations of multiple sequences:

```
>>> [x + y for x in range(0,3) for y in range(0,3)]  
[0, 1, 2, 1, 2, 3, 2, 3, 4]
```

The `while` loop

The `while` loop is useful when you need to iterate over a non-sequence. The basic syntax of the `while` loop is:

```
while expression:  
    statements  
else:  
    statements
```

The statements following the `else` clause are executed if the `expression` is initially false.

As always, remember the colon (`:`) and proper indentation.

Control inside Loops

If you wish to stop the execution of a loop before it's completed, you can use the `break` statement. Control will be transferred to the next statement after the loop (including any `else` and `elif` clauses). The `break` statement is especially useful in Python, because you can not use an assignment statement as the expression to be tested in a `while` loop.

If you wish to skip the remaining statements for a single iteration of a loop, and immediately begin the next iteration, you can use the `continue` statement.

while loop example

Suppose we wish to select ten numbers from the integers from 1 to 100, without any number appearing twice. We can use the `randint` function of the `random` module to generate the numbers, and a `while` loop to stop when we've found our ten numbers.

```
import random

got = 0
nums = []

while got < 10:
    i = random.randint(1,100)
    if i not in nums:
        nums.append(i)
        got = got + 1
```

Example: Counting Fields in a File

```
import sys

filename = sys.argv[1]

try:
    f = open(filename, 'r')
except IOError:
    print >>sys.stderr, "Couldn't open %s" % filename
    sys.exit(1)

counts = {}
while 1:
    line = f.readline()
    if not line:
        break
    line = line[:-1]
    fields = line.split(',')
    l = len(fields)
    counts[l] = counts.get(l,0) + 1

keys = counts.keys()
keys.sort()

for k in keys:
    print '%d %d' % (k, counts[k])
```

Writing Functions

The `def` statement creates functions in Python. Follow the statement with the name of the function and a parenthesized list of arguments. The arguments you use in your function are local to that function. While you can access objects outside of the function which are not in the argument list, you can not change them.

The function definition should end with a colon (`:`) and the function body should be properly indented.

If you want your function to return a value, you must use a `return` statement.

You can embed a short description of the function (accessed through the interactive `help` command) by including a quoted string immediately after the function definition statement.

Example: Writing a Function

```
def merge(list1, list2):  
    """merge(list1,list2) returns a list consisting of the  
    original list1, along with any elements in list2 which  
    were not included in list1"""  
  
    newlist = list1[:]  
    for i in list2:  
        if i not in newlist:  
            newlist.append(i)  
  
    return newlist
```

Variable number of Arguments

To handle variable numbers of arguments, put an asterisk before an argument name, and it will create a list of that name (without the asterisk) inside your function. Similarly, to collect named arguments to your function, use two asterisks before an argument name, and a corresponding dictionary will be created.

```
>>> def funny(a,b,*unn,**nm):
...     for u in unn:
...         print u
...     print '-' * 10
...     for n in nm:
...         print n,nm[n]
...
>>> funny(7,3,12,19,22,first=1,last=10)
12
19
22
-----
last 10
first 1
```

Named Arguments and Default Values

If you want to set defaults for some arguments, so that you don't have to specify all the arguments when you call a function, simply place an equal sign (=) and the desired default after the variable's name in the function definition.

You can use a similar syntax when calling a function to specify the arguments in an arbitrary order. (By default, arguments must be passed to a function in the order in which they appear in the function definition.)

Functional Programming: `map` and `filter`

Python provides two functions which accept a function as one of its arguments.

`map` takes a function and a list, and applies the function to each member of the list, returning the results in a second list. As an example of the `map` function, suppose we have a list of strings which need to be converted to floating point numbers. We could use:

```
values = map(float, values)
```

`filter` takes a function which returns `True` or `False` as its first argument and a list as its second. It returns a list containing only those elements for which the provided function returned `True`. The `path.isdir` function of the `os` module returns a value of 1 (`True`) if its argument is a directory. To extract a list of directories from a list of files, we could use:

```
dirs = filter(os.path.isdir, files)
```

Functional Programming (cont'd)

You can provide your own functions to `map` or `filter`. While you can always define these functions in the usual way, for simple one-line expressions a `lambda` expression can be used to define a function in place. To illustrate the syntax, consider the following statement, which removes the last character of each element of the list `lines` and stores the result back into the list:

```
lines = map(lambda x:x[:-1],lines)
```

The following statements would have equivalent results:

```
def rmlast(x):  
    return x[:-1]  
lines = map(rmlast,lines)
```

Using Modules

There are three basic ways to use the `import` statement to make functions and other objects from modules available in your program.

1. `import module`

Objects from `module` need to be referred to as `module.objectname`. Only the module name is actually imported into the namespace.

2. `from module import function`

The name `function` will represent the object of that name from `module`. No other symbols are imported from the module (including the module's name).

3. `from module import *`

The names of all of the objects in `module` are imported into the namespace. This form of the `import` statement should only be used if the module author explicitly says so.

Some Useful Python Modules

- `re` – Perl-style regular expressions
- `os` – Operating System Interface (`system`, `environ`, etc.)
- `os.path` – automatically imported with `os`
- `sys` – access Python's current environment (`argv`, `exit`, etc.)
- `copy` – true copies of list (`copy`, `deepcopy`)
- `pickle` – serialize Python objects for later retrieval
- `cgi` – access variables from CGI scripts on web server
- `urllib` – access URLs as if they were local files

A Brief Introduction to Regular expressions

The `re` module provides perl-compliant regular expression support in python. Regular expressions allow us to search and modify character strings based on patterns that occur in the text. They are constructed of

1. literal characters - matched if the literal character is in the string
2. character classes - matched if any of the characters within square brackets (`[]`) are in the string
3. negated character classes - if the first character in a character class is the caret (`^`), it is matched by any character **not** within the square brackets.
4. special characters - which are shortcuts for special patterns or which act as modifiers for other patterns

Special Characters in Regular Expressions

Character	Use and Meaning
.	Any character
^	Beginning of string
\$	End of string
*	Zero or more of preceding entity
+	One or more of preceding entity
()	Grouping
	Alternation

If you need to use any of these special characters literally, they must be preceded by a backslash \.

In addition, the following escape sequences can be used:

Symbol	Matches	Symbol	Matches
\w	Alphanumerics and _	\W	Non-alphanumerics
\d	Digits	\D	Non-digits
\s	Whitespace	\S	Non-whitespace

Some Simple Examples

Find email addresses in text:

```
r'[-A-Za-z0-9_\.%]+@[[-A-Za-z0-9_\.\\%]+\.[A-Za-z]+'
```

Find HTML lines with links:

```
r'<a +href *='
```

Find lines of text with numbers:

```
r'[0-9.]+'
```

Find lines containing dog or cat:

```
r'dog|cat'
```

Find file names of mp3 files:

```
r'\.mp3$'
```

Using Regular Expressions in python

Before a regular expression can be used, it must be compiled. For convenience, when the following methods are called as functions (*e.g.* `re.search`, `re.findall`), they accept an uncompiled regular expression.

The function to compile a regular expression is `re.compile`. You pass it a regular expression, and it returns the compiled form. It also accepts the following options, which can be joined with a logical or (`|`)

Short Name	Long Name	Purpose
I	IGNORECASE	Non-case-sensitive match
M	MULTILINE	Make <code>^</code> and <code>\$</code> match beginning and end of lines within the string, not just the beginning and end of the string
S	DOTALL	allow <code>.</code> to match newline, as well as any other character
X	VERBOSE	ignore comments and unescaped whitespace

For example, to find the word `cat` at the beginning of lines in a multiline string, ignoring case, use `re.search('^cat',txt,re.I|re.M)`.

Regular Expression functions/methods

- `re.compile` accepts a text regular expression, and returns a compiled regular expression

Example: `mypat = re.compile(r'<a +href *=')`

- `re.search` accepts a regular expression, and a string, and returns a match object or `None`

Example: `m = re.search(r'<a +href +=', txt)`

`m = mypat.search(txt)`

`m` could then be tested or examined.

- `re.findall` accepts a regular expression and a string, and returns an empty list, or a list containing the matched string

`mypat.findall('some text <a href="something"')` returns
`['<a href=']`

- `re.sub` substitutes text in a string based on regular expressions

Example: `new = re.sub(r'<a +href +=', '<img src=', txt)`

`new = mypat.sub('<img src=', txt)`

Tagging of Regular Expressions

Just finding a line with a regular expression doesn't completely solve text manipulation problems. To focus on specific portions of text, you can use tagging, *i.e.* surrounding text of interest with (unescaped) parentheses. Tagged groups can then be extracted from match objects, or used in substitutions, referring to tagged groups as `\1`, `\2`, etc.

Examples:

Delete text from comment character to the end of a line:

```
re.sub(r'^(.*)#.*$',r'\1',txt)
```

Reverse words and numbers:

```
re.sub(r'(\w+) (\d+)',r'\2 \1',txt)
```

When a regular expression contains tagged expressions, `findall` returns those expressions in a tuple:

```
>>> mypat = re.compile('\s*(\w+)\s*(\d+)')
>>> txt = ' hello 720 '
>>> mypat.findall(txt)
[('hello', '720')]
```

Match Objects

In addition to other methods of working with regular expressions, you can operate on the match object returned by `re.search` directly.

```
>>> imgpat = re.compile(''< *img +src *= *["](.*)["]'')
>>> txt = "Here's an image <img src='photo.jpg'> More text"
>>> m = imgpat.search(txt)
>>> m.group()          # returns entire match
"<img src='photo.jpg'"
>>> m.group(1)        # returns tagged expression
'photo.jpg'
>>> m.groups(1)       # same, but as a tuple
('photo.jpg',)
>>> m.start()         # position of entire match
16
>>> m.end()           # end of entire match
36
>>> m.span()          # start and end of entire match
(16, 36)
>>> m.span(1)         # start and end of tagged expression
(26, 35)
>>> m.expand('Image name: \\1')  # like the sub command
'Image name: photo.jpg'
```

Greediness of Regular Expressions

Following a previous example, suppose we wanted to extract image names from the following:

```
>>> txt = 'Some text'
>>> imgpat = re.compile(''< *img +src *= *['"](.*)['"]''')
>>> imgpat.findall(txt)
['one.gif">Some text>> imgpat.findall(txt)
['one.gif', 'two.gif']
```

2. Use the ? modifier to the * and + operators for a non-greedy search:

```
>>> imgpat = re.compile(''< *img +src *= *['"](..*?)['"]''')
>>> imgpat.findall(txt)
```

The cPickle module

The `cPickle` module serializes any python object into a binary form that can be stored on disk, and easily restored into a python session. Suppose we've created a python object called `mydb`, and we wish to save it in a file called `datastore`.

```
import cPickle,sys
try:
    f = open('datastore','w')
except IOError:
    print "Couldn't open datastore for writing"
    sys.exit()

p = cPickle.pickler(f,1)
p.dump(mydb)
f.close()
```

Restoring a cPickled object

To restore the object, use code like the following

```
import cPickle
try:
    f = open('datastore', 'r')
except IOError:
    print "Couldn't open datastore for reading"

mydb = cPickle.load(f)
f.close()
```

Note that you can assign the object to any name that you choose, not just the one under which it was stored.

Basics of Object-Oriented Programming in Python

To create a new class in python, you provide the name of the class after the `class` keyword, followed by a colon with the various methods which are defined for the class indented in the usual way. Each class must have an `__init__` method, which will be called when the class is first invoked. Within the methods, objects whose names are preceded by `self`. will be stored as attributes in the object, while other variables will be local to the defined methods. All the methods must take the object `self` as their first argument. In addition to methods which you provide names, you can write special methods (shown on the next slide) that allow operator overloading.

Operator Overloading

The following table shows the methods that implement operator overloading in python:

Method	Use
<code>__init__(object)</code>	called when class constructor is invoked
<code>__repr__(object)</code>	also called when object name typed in interpreter
<code>__del__(object)</code>	called when an object is destroyed
<code>__str__(object)</code>	called by <code>print(object)</code>
<code>__len__(object)</code>	called by <code>len(object)</code>
<code>__getitem__(object,key)</code>	allows you to intercept subscripting requests
<code>__setitem__(object,key,value)</code>	allows you to set values of subscripted items
<code>__getslice__(object,start,fin)</code>	allows you to intercept slice requests
<code>__setslice__(object,start,fin,value)</code>	allows you to set slices
<code>__add__(object,other)</code>	called by <code>object + other</code>
<code>__radd__(object,other)</code>	called by <code>other + object</code>
<code>__sub__(object,other)</code>	called by <code>object - other</code>
<code>__mul__(object,other)</code>	called by <code>object * other</code>
<code>__mod__(object,other)</code>	called by <code>object % other</code>

Object-Oriented Programming

As an example of a simple object-oriented program, consider writing a program that will allow us to store and retrieve names, phone numbers and email addresses from our friends. First, we'll create the `__init__` method. This is the method that is invoked when we create a new object using the class name as a constructor, so it contains the arguments that we'll use to create an object:

```
class Friend:
    def __init__(self, name, phone='', email=''):
        self.name = name
        self.phone = phone
        self.email = email
```

Now define methods to display the object:

```
def __str__(self):
    return 'Name: %s\nPhone: %s\nEmail: %s' % \
           (self.name, self.phone, self.email)
def __repr__(self):
    return self.__str__()
```

Object-Oriented Programming (contd.)

We can now try creating and displaying “Friend” objects:

```
>>> x = Friend('Joe Smith', '555-1234', 'joe.smith@notmail.com')
```

```
>>> y = Friend('Harry Jones', '515-2995', 'harry@who.net')
```

```
>>> print x
```

```
Name: Joe Smith
```

```
Phone: 555-1234
```

```
Email: joe.smith@notmail.com
```

```
>>> y
```

```
Name: Harry Jones
```

```
Phone: 515-2995
```

```
Email: harry@who.net
```

Next, we create a `Frienddb` object, which will use the `cPickle` module to store the `Friend` objects. We’ll initialize the object by passing it a file name: if the file name exists, it will use it as the data base, and if not, we’ll create the data base.

Object-Oriented Programming (contd.)

```
import sys
class Frienddb:
    def __init__(self,file=None):
        if file == None:
            print 'Must provide a filename'
            return
        self.file = file
        if os.path.isfile(file):
            try:
                f = open(file,'r')
            except IOError:
                sys.stderr.write('Problem opening file %s\n' % file)
                return
            try:
                self.db = cPickle.load(f)
                return
            except cPickle.UnpicklingError:
                print >>sys.stderr, '%s is not a pickled database.' % file
                return
            f.close()
        else:
            self.db = []
```

Object-Oriented Programming (contd.)

Now we can write methods to add objects to our database, and to search for the names of our friends. (In practice, these would all be indented under the `class` definition.)

```
def add(self, name, phone='', email=''):
    self.db.append(Friend(name, phone, email))
def search(self, pattern):
    srch = re.compile(pattern, re.I)
    found = []
    for item in self.db:
        if srch.search(item.name):
            found.append(item)
    return found
```

We can simply append records to the empty list that was initialized in the `__init__` method.

Object-Oriented Programming (contd.)

Next, we'll provide a method to store the database through pickling, and to insure that we don't lose data, a `__del__` method which saves our data:

```
def store(self):
    import cPickle
    try:
        f = open(self.file,w)
    except IOError:
        print >>sys.stderr, \
            Problem opening file %s for write\n % self.file
        return
    p = cPickle.Pickler(f,1)
    p.dump(self.db)
    f.close()
def __del__(self):
    if self.db:
```

```
self.store()
```

Testing Our Class

The following program allows us to interactively test the methods that were implemented.

```
import frienddb
file = raw_input(File? )
fdb = Frienddb(file)
while 1:
    line = raw_input(> )
    if line[0] == a:
        name = raw_input(Name? )
        phone = raw_input(Phone number? )
        email = raw_input(Email? )
        fdb.add(name,phone,email)
    elif line[0] == ?:
        line = string.strip(line[1:])
        fdb.search(line)
    elif line[0] == q:
        fdb.store()
        break
```

Testing Our Class (cont'd)

Running the program:

```
springer.spector$ python frienddb.py
```

```
File? friend.db
```

```
> a
```

```
Adding to database...
```

```
Name? John Smith
```

```
Phone number? 555-1212
```

```
Email? jsmith@nowhere.com
```

```
> a
```

```
Adding to database...
```

```
Name? Fred Jones
```

```
Phone number? 555-3211
```

```
Email? fredj@somewhere.com
```

```
> ? Jones
```

```
Name: Sue Jones
```

```
Phone: 332-1991
```

```
Email: sue@sue.net
```

```
-----
```

```
Name: Fred Jones
```

```
Phone: 555-3211
```

```
Email: fredj@somewhere.com
```

```
-----
```

Inheritance

Suppose we found our friend data base to be so useful that we decided to create one for business. We'll create a new object, **Contact**, that will hold information about our business contacts:

```
class Contact:
    def __init__(self,name,company,phone='',email='',product=''):
        self.name = name
        self.company = company
        self.phone = phone
        self.email = email
        self.product = product
    def __str__(self):
        return Name:%s\nCompany:%s\nPhone:%s\nEmail:%s\nProduct:%s % \
            (self.name,self.company,self.phone,self.email,self.product)
    def __repr__(self):
        return self.__str__()
```

Inheritance (cont'd)

Having created the `Contact` class, we now need to create a `Contactdb` class to hold the records. But the basics of opening files, pickling and searching are identical to those for the `Frienddb`. All that's changed is what we'll be adding to our database, so we write a new `add` method, and allow the other methods to be inherited:

```
from frienddb import Frienddb
class Contactdb(Frienddb):
    def add(self,name,company,phone='',email='',product=''):
        self.db.append(Contact(name,company,\
                               phone,email,product))
```

Once we create a `Contactdb` object, we can use the `store` and `search` methods with no modification.

Extending Methods

Now suppose we want to extend the `search` method for the `Contactdb` class to allow searching for things other than names. Note how the `search` method for the `Frienddb` class can be called directly, when we wish to search for a name:

```
def search(self,name='',company='',product=''):
    results = []
    if name:
        found = Frienddb.search(self,name)
        results.extend(found)
    if company:
        srch = re.compile(company,re.I)
        found = []
        for item in self.db:
            if srch.search(item.company):
                found.append(item)
        results.extend(found)
    if product:
        srch = re.compile(product,re.I)
        found = []
        for item in self.db:
            if srch.search(item.product):
                found.append(item)
        results.extend(found)

    return results
```