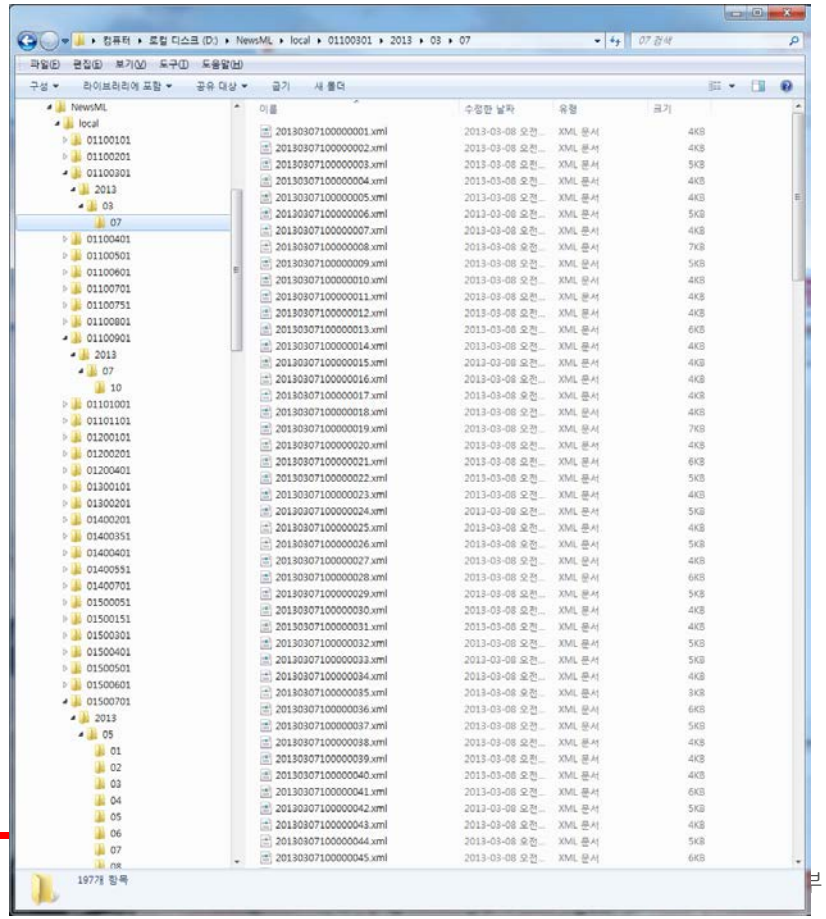
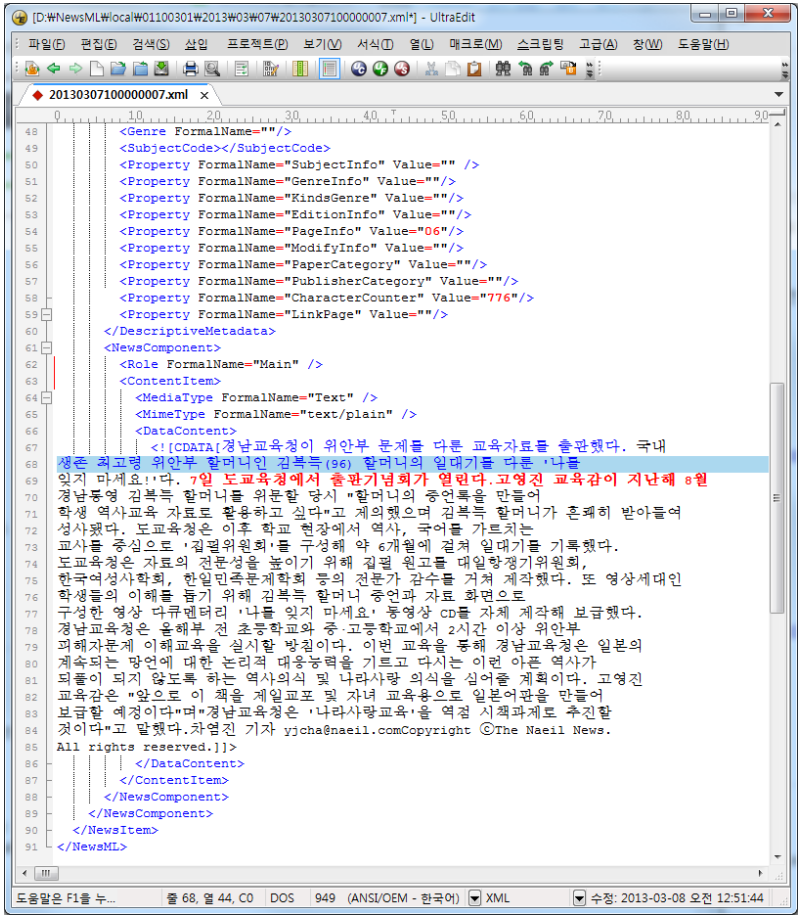
Python 주요 내용

국민대학교 자연어처리 연구실
강승식

<실제 상황> 내가 해야 할 일은...

- **수많은 폴더의 수많은 파일: 3,665 폴더 632,023 파일(압축 14G)**
  - Text만 추출 -- 1개의 파일로 저장, 단어 빈도 조사 등...





Python 설치

- Python 인터프리터 -- http://www.python.org/
- 리눅스 -- Python 내장
- 윈도우
  - 설치 후 환경변수 Path에 Python 설치된 절대경로 추가
    - **제어판-시스템-고급시스템설정-환경변수-Path**
- 참고자료
  - **점프 투 파이썬 -- http://www.wikidocs.net/1**

Index

- Python 소개
- 자료형
- 제어문
- 입출력
- 함수
- 클래스
- 모듈
- 라이브러리
- 예제 프로그램

Python 이란?

- 1990년 귀도 반 로섬 (Guido van Rossum) 개발
- 기원 -- “Monty Python’s Flying Circus”(네덜란드 어린이 프로그램)
- 사전적 의미
  - Python : 그리스 신화 속, 파르나수스 산의 동굴에 사는 큰 뱀
  - Python 서적의 표지, Python 아이콘에 뱀이 그려지는 주된 이유
- 활용
  - 아나콘다
    - 레드햇 리눅스 설치 프로그램
  - 구글(Google), 인포시크(Infoseek)
    - 검색 프로그램

Python의 특징

- 인간다운 언어
  - 인간의 대화 방식에 가까운 문법
  - if 4 in [1,2,3,4]: print “4가 있습니다.”
- 문법이 쉽다
  - 빠른 학습
- 강력하다
  - C언어와 완벽한 연동
- 오픈소스
  - 무료사용, 자유배포
- 간결하다
  - 판독성이 높은 언어
- 빠른 개발 속도

Python의 활용범위

- GUI
  - Tkinter 사용 → 5 line으로 윈도 창을 띄울 수 있다?
  - wxPython, PyQT, PyGTK 등 -- Tkinter보다 빠른 속도와 더 좋은 인터페이스
- C/C++과 연계 프로그래밍
  - 다른 언어와 연계가 편리함
- CGI(Common Gateway Interface)
- 수치 연산, DB 프로그래밍
- 빅데이터 분석 프로그래밍

- Python으로 할 수 없는 일
  - 운영체제 개발 등 시스템 프로그래밍
  - 실행 효율이 매우 중요한 서버 프로그래밍
  - 모바일 프로그래밍

자료형(1/12) - 숫자형

항목	사용 예	비고
Integer	123, -456, 0	
Long Integer	123456789123L	Renamed to *int* in Python 3.x
Float	123.45, -1234.5, 3.4e10	지수는 e와 E 모두 허용
Complex	1+2j, -3j	a.real(), a.imag(), a.conjugate()
Octal	034, 025	0o34, 0o25 in Python 3.x
Hexadecimal	0x2A, 0xFF	

- 나눗셈 연산자 -- / vs. //
  - 1./2 vs. 1.//2
- ** : 지수 연산

자료형(2/12) - 문자열

- “abcd” vs. ‘abcd’
  - 예1) “I’m going”
  - 예2) ‘He says “Python is very easy.”’
  - \”, \’ 가능
- 여러 줄의 문자열 처리
  - 1) “안녕하세요\nPython 강의중입니다.”
    2) “””
    안녕하세요
    Python 강의중입니다.
    “””
- C와 동일한 escape code 지원

- a + b : string concatenation
- a * 2 : 문자열 2회 반복

자료형(3/12) – 문자열(계속)

- 문자열에 변수 대입
  - C언어와 동일 : %c, %s, %d, %f, %o, %x, …
- 문자열 관련 함수

함수	설명
a.upper() / a.lower()	문자열 전체를 대문자(소문자)로 변경
a.count(x)	문자열에서 x와 일치하는 것의 개수 리턴
a.find(x) / a.index(x)	문자열에서 x가 처음 출연하는 위치 리턴. 없으면 find는 -1 반환, index는 에러
a.join(s)	문자열 s의 각 문자 사이에 문자열 삽입
a.lstrip() / a.rstrip() / a.strip()	문자열의 왼쪽, 오른쪽, 양쪽 공백 삭제
a.replace(s,r)	문자열에서 s라는 부분 문자열을 r로 치환
a.split([s])	문자열을 공백 단위로 쪼개서 리스트로 리턴
a.swapcase()	문자열의 모든 단어의 대소문자를 각각 변환

자료형(4/12) – 리스트

- C언어의 array 개념
  - a = [ ]
  - b = [1, 2, 3]
  - c = [‘Life’, ‘is’, ‘too’, ‘short’]
  - d = [1, 2, ‘Life’, ‘is’]
  - e = [1, 2, [‘Life’, ‘is’]]

- 표현이 자유롭다.
  - 정적/동적 할당의 의미 없음 – 인터프리터가 제어
  - 특정 자료형에 국한되지 않는다.
    - 하나의 리스트에 다양한 형태의 자료형 존재 가능

- 문자열도 리스트로 인식됨

자료형(5/12) – 리스트(계속)

- 인덱싱
  - 리스트를 배열 형태로 사용
  - 0 ~ n : 앞에서부터 인덱싱
  - -1 ~ -n : 뒤에서부터 인덱싱
  - e = [1, 2, [‘Life’, ‘is’]]
    - e[0] -> 1, e[-1] -> [‘Life’, ‘is’], e[2][-1] -> ‘is’, e[2][1][0] -> ‘i’

- 슬라이싱
  - 리스트를 분할하여 부분 리스트 생성
  - 인덱싱을 사용한다
    - a[i:j] -> i번째부터 j번째까지
    - a[i:] -> i번째부터 리스트 끝까지
    - a[:j] -> 리스트 처음부터 j번째까지
    - a[:] -> 리스트 처음부터 끝까지
- ‘+’ : concatenation, * : repetition

자료형(6/12) – 리스트(계속)

- 리스트 관련 함수

함수	설명
a.append(x)	리스트 a의 마지막에 x 추가
a.sort()	리스트 a를 정렬
a.reverse()	리스트 a의 순서를 거꾸로 만든다.
a.index(x)	리스트 a에서 x를 찾아서 그 위치 리턴
a.insert(i,x)	리스트 a의 i번째 위치에 x 삽입
a.remove(x)	리스트 a에서 처음 나오는 x 삭제
a.pop()	리스트 a의 맨 마지막 요소를 반환하고 그 요소 삭제
a.count(x)	리스트 a 안에 있는 x의 개수를 리턴
a.extend(x)	리스트 a에 리스트 x를 더함(확장)

자료형(7/12) – 튜플

- Constant 리스트
  - 원소값에 대한 수정, 삭제가 불가능
- ()를 사용하여 생성
  - t1 = ()
  - t2 = (1,)
  - t3 = (1,2,3)
  - t4 = 1,2,3
  - t5 = (‘a’, ‘b’, (‘ab’, ‘cd’))
- 인덱싱, 슬라이싱, 더하기, 곱하기 가능
- 용도에 맞게 리스트와 혼용

자료형(8/12) - 딕셔너리

- Perl의 해쉬 변수
  - 대응관계(key-value)를 저장 하는 리스트
  - Associative array, hash 기능

- key, value의 쌍으로 저장
  - dic = {‘name’:‘kang’, ‘phone’:‘01012345678’, ‘major’:‘CS’}

key	value
name	kang
phone	01012345678
major	CS

- 리스트를 원소로 사용 가능
  - dic1 = {‘number’:[1,2,3,4]}

자료형(9/12) – 딕셔너리(계속)

- key값을 통해 value를 추출
  - dic[‘name’] -> ‘kang’
  - 인덱싱, 슬라이싱 방법은 사용할 수 없다.
  - [] 안의 값은 key값만 가능함
- 딕셔너리 쌍의 추가
  - dic[‘lab’] = ‘NLP’
    - dic -> {‘lab’:‘NLP’, ‘name’:‘kang’, ‘phone’:‘01012345678’, ‘major’:‘CS’}
  - 추가되는 요소에 대한 순서는 원칙이 없음
    - ‘무엇이 추가되었나’의 사실만 존재
- 딕셔너리 쌍의 삭제
  - del dic[‘phone’]
    - dic -> {‘lab’:‘NLP’, ‘name’:‘kang’, ‘major’:‘CS’}
- Key값 - unique
  - 중복되는 key값은 하나만 인식이 된다.
  - 추출되는 요소는 추측 불가

자료형(10/12) – 딕셔너리(계속)

- 딕셔너리 관련 함수

함수	설명
dic.keys(x)	딕셔너리의 key들을 모아놓은 리스트를 리턴
dic.values()	딕셔너리의 vlaue들을 모아놓은 리스트를 리턴
dic.items()	딕셔너리의 (key,value)쌍의 터플을 모아놓은 리스트를 리턴
dic.clear()	딕셔너리의 모든 key:value 쌍을 삭제
dic.get(x)	딕셔너리의 key x의 value를 리턴
dic.has_key(x)	딕셔너리에 key x가 있는지 확인하여 참, 거짓을 리턴

자료형(11/12) – 참/거짓

- 참/거짓의 종류

자료형	참/거짓
“”가 아닌 문자열(내용이 있는 문자열) []가 아닌 리스트(내용이 있는 리스트) ()가 아닌 튜플(내용이 있는 튜플) {}가 아닌 딕셔너리(내용이 있는 딕셔너리)	참
“” / [] / ()/ {}	거짓
0이 아닌 수	참
0	거짓
none	거짓

자료형(12/12) – 변수

- Python의 변수 = 객체
  - 주소 값을 저장(데이터를 수정하면 원본에도 영향)
- 변수 생성
  - a = 1 / a = b = 1
  - a, b = 1, 2 / (a, b) = (1, 2) / [a, b] = [1, 2]
- 변수 삭제
  - del(a)
  - Garbage collection
- swap
  - a, b = b, a
- 리스트 / 튜플 복사
  - 슬라이싱, copy(a)
- 변수 객체 비교
  - is 함수
  - a is b -> 참/거짓 리턴

제어문(1/3) - if

- syntax
  - : 사용, 들여쓰기에 유의

```
>>> if <조건문>:
...         <수행할 문장>
...   else:
...         <수행할 문장>
...
>>>
```

- in / not in
  - 문자열, 리스트, 튜플의 원소인지를 검사
- elif → Perl의 elsif
  - else if와 동일
- pass
  - if / elif / else 이후 수행할 문장이 없을 경우 사용

제어문(2/3) - while

- syntax

```
>>> while <조건문>:
…            <수행할 문장>
…
>>>
```

- 무한루프
  - while 1:

- break / continue 가능

제어문(3/3) – for

- syntax - Java의 for each와 비슷함

```
>>> for 변수 in 리스트(튜플, 문자열):
…          <수행할 문장>
…
>>>
```

- range() 함수
  - 주어진 범위의 숫자 리스트를 자동으로 작성
  - “시작점(default = 0)”부터 “끝번호 – 1”까지의 리스트를 리턴
  - range(10) → [0,1,2,3,4,5,6,7,8,9]
  - range(1,11) → [1,2,3,4,5,6,7,8,9,10]

입출력(1/3) - 입력

- input()
  - 기본 입력 -- 정수, 실수를 숫자로 변환
  - 문자열 입력 -- “abcd” 또는 ‘abcd’

- raw_input()
  - 모든 입력을 문자열로 인식
  - “” 또는 ‘’ 없이 사용

- input(질문), raw_input(질문)

```
>>> a = input(“정수 1개 : ”)
>>> b = raw_input(“한글 문장 : ”)
```

입출력(2/3) - 출력

- print
  - 기본 출력
- print 에서 “” 사용
  - + 연산자 (concatenation)
  - print “Life” “is” “too short”
  - print “Life” + “is” + “too short”
- print에서 , 사용
  - white space
  - print “Life” , “is” , “too short”

입출력(3/3) – 파일 입출력

- open
  - 파일객체 = open(파일이름, 파일모드)
  - 파일모드 : r(읽기) / w(쓰기) / a(추가)

- 파일 객체 함수

함수	설명
f.write(x)	파일에 변수 x를 기록
f.read()	파일의 전체 내용을 읽는다
f.readline()	파일의 내용을 한 줄씩 읽는다
f.readlines()	파일의 내용을 한 줄 단위로 모두 읽는다
f.tell()	현재 파일 포인터의 위치를 리턴
f.seek(x)	파일 포인터의 위치를 x로 이동
f.close()	파일을 닫는다

```
f = open("test.txt", 'r')
txt = f.readlines()
f.close()

f2 = open("output.txt", 'w')
f2.write("This is a test.")
f2.close()
```

함수

- syntax

```
>>> def 함수명(입력인수):
…          <수행할 문장>
…          return 리턴값
…
>>>
```

- 숫자가 불특정한 입력인자 허용
  - def function(arg1, *args)
  - args는 터플로 생성
- 함수호출, 지역변수의 범위
  - C언어와 동일
- 재귀 호출 가능

클래스(1/2) – 클래스 개요

- syntax

```
>>> class 클래스명([상속 클래스명]):
…          <멤버 변수>
…          def 멤버 함수(self[, 인수들])
…                    <함수 body>
…
>>>
```

- self
  - this와 비슷한 의미
    - this를 사용해야 할 부분에는 self를 입력
  - 객체의 확인 과정에 필요한 인수
    - 객체에 대한 확인 과정을 통해 자동으로 객체명으로 변경

- __init__
  - constructor

클래스(2/2) – 연산자 오버로딩

함수명	관련 연산자	호출시기
__init__	생성자(constructor)	객체 생성
__del__	소멸자(destructor)	객체 소멸
__add__	연산자 '+'	'+' 연산자를 사용
__sub__	연산자 '-'	'-' 연산자를 사용
__or__	연산자 '|'	'|' 연산자를 사용
__call__	함수 호출	함수 호출
__getattr__		객체의 자격부여 검사
__getitem__	x[i]	인덱싱, for loop, in 사용
__setitem__	x[i] = value	인덱스 치환
__getslice__	x[i:j]	슬라이싱
__len__	len()	len(x) 호출
__cmp__	비교 연산자	비교 연산자 사용

모듈

- C의 헤더 파일과 유사
  - 미리 생성한 변수, 함수, 클래스 등을 불러서 사용
- 사용 방법
  - import 모듈명
  - from 모듈명 import 모듈함수명
  - from 모듈명 import *
- if __name__ == “__main__”
  - 모듈 파일의 자동 실행을 방지
- reload(모듈명)
  - 인터프리터 실행 중, 모듈 내용이 변경 되었다면, 재실행 가능

라이브러리(1/2) – 내장 함수

- Python 자체 내장 함수 - import 없이 바로 실행 가능
  - abs(x), cmp(x,y), max(x), min(x), pow(x,y)
  - int(x), float(x), long(x), hex(x), oct(x), list(x), str(x), tuple(x)

함수	설명
apply(func, (args))	함수를 간접적으로 실행
chr(x)	정수를 입력 받아서 그에 해당하는 아스키 코드를 리턴
dir(x)	객체 x의 멤버 함수/멤버 변수들을 리스트 형태로 리턴
divmod(x,y)	x를 y로 나눈 몫과 나머지를 터플 형태로 리턴
eval(expression)	실행 가능한 형태의 문자열(ex-“1+1”)을 입력받아서 실행한 결과를 리턴
execfile(file)	*.py 파일을 실행
id(obj)	객체의 고유 번호(주소값)을 리턴
isinstance(obj,class)	주어진 객체가 주어진 클래스의 객체인지 확인 후 참/거짓 리턴
lambda	함수를 간단하게 생성하는 예약어. 리스트 등에도 사용 가능
map(func,list)	함수의 인수로 주어진 리스트의 요소들을 차례로 인풋, 모든 결과 값들을 리스트로 리턴
ord(x)	주어진 문자의 아스키 코드 값을 리턴(<->chr)
reduce(func, list)	입력 인수가 2개인 함수에 주어진 리스트를 순차적으로 인풋, 최종 결과를 리턴
type(obj)	주어진 객체의 종류를 리턴

라이브러리(2/2) – 외부 모듈

모듈	설명
sys	인터프리터 제어 관련(exit, args관리 등)
pickle	객체 <-> 파일
string	문자열
os	OS의 시스템 명령어 사용
StringIO	파일 객체 기능의 가상 객체 문자열
re	정규표현식 지원
shutil	파일 복사
glob	디렉토리 정보를 리스트로 변환
tempfile	임시 파일 생성
time	시간
calendar	달력
random / whrandom	난수 발생(wh: Wichmann-Hill)
thread	스레드 사용
socket / webbrowser / urllib / ftplib / smtplib / poplib	네트워크 관련

예제(1) – 해당 디렉토리의 파일명 얻기

```
# globtest.py
# process in current directory.. and files..
import glob

# Current directory
fileList = glob.glob("*")
print fileList

# Another directory..
fileList = glob.glob("c:\*")
print fileList

# only filename
import os
for fullFile in fileList:
    if os.path.isdir(fullFile): # is directory??
        print 'Directory\t:', os.path.basename(fullFile)
    else:
        print os.path.basename(fullFile)
```

예제(2) – 하위 디렉토리 검색(1)

```
import os

def search(dirname):
    flist = os.listdir(dirname)
    for f in flist:
        next = os.path.join(dirname, f)
        if os.path.isdir(next):
            search(next)
        else:
            doFileWork(next)
def doFileWork(filename):
    ext = os.path.splitext(filename)[-1]
    if ext == '.py': print filename

search("d:/")
```

예제(2) – 하위 디렉토리 검색(2)

```
def doFileWork(filename):
    ext = os.path.splitext(filename)[-1]
    if ext != ".py":return
    f = open(filename)
    before = f.read()
    f.close()
    after = before.replace("ABC", "DEF")
    f = open(filename, "w")
    f.write(after)
    f.close()
```

예제(3) – 파일을 읽어 단어별로 처리

```
# fileword.py
text = “””
You need python.
Why??
Answer is too simple.
Python is very easy and powerful.
First of all, It's very fun!!
“””

import cStringIO

likeFile = cStringIO.StringIO(text)
fileLines = likeFile.readlines()
word = []
for line in fileLines:
      word += line.split()

print word
```

What's New In Python 3.0

- Few changes compared to Python 2.5. --
- **Print Is A Function:** The print statement has been replaced with a function.
  - Old: print "The answer is", 2*2
  - New: print("The answer is", 2*2)
  - Old: print x, # Trailing comma suppresses newline
  - New: print(x, end=" ") # Appends a space instead of a newline
  - Old: print # Prints a newline
  - New: print() # You must call the function!
  - Old: print >>sys.stderr, "fatal error"
  - New: print("fatal error", file=sys.stderr)
  - Old: print (x, y) # prints repr((x, y))
  - New: print((x, y)) # Not the same as print(x, y)!
  - in Python 3.0, You can also customize the separator between items, e.g.:
  - print("There are <", 2**32, "> possibilities!", sep="") produces
    - There are <4294967296> possibilities!
  - in Python 2.x, print "A₩n", "B" would write "A₩nB₩n";
  - in Python 3.0, print("A₩n", "B") writes "A₩n B₩n "
- When using the 2to3 source-to-source conversion tool, all print statements are automatically converted to function calls, so this is mostly a non-issue for larger projects.

- **Views And Iterators Instead Of Lists**
  - dict methods dict.keys(), dict.items() and dict.values() return “views” instead of lists.
  - For example, this no longer works: k = d.keys(); k.sort(). Use k = sorted(d) instead.
  - dict.iterkeys(), dict.iteritems() and dict.itervalues() methods are no longer supported.
  - map() and filter() return iterators.

- **Ordering Comparisons**
  - The ordering comparison operators (<, <=, >=, >) raise a TypeError exception when the operands don't have a meaningful natural ordering.
  - Thus, expressions like 1 < '', 0 > None or len <= len are no longer valid, and e.g. None < None raises TypeError instead of returning False.
  - Sorting a heterogeneous list no longer makes sense.

- **Text Vs. Data Instead Of Unicode Vs. 8-bit**
  - Python 3.0 uses the concepts of *text* and (binary) *data* instead of Unicode strings and 8-bit strings.
  - Mixing text and data in Python 3.0 raises TypeError.
  - You can no longer use u"..." literals for Unicode text. However, you must use b"..." literals for binary data.
  - As the str and bytes types cannot be mixed, you must always explicitly convert between them. Use str.encode() to go from str to bytes, and bytes.decode() to go from bytes to str. You can also use bytes(s, encoding=...) and str(b, encoding=...), respectively.

- Filenames are passed to and returned from APIs as (Unicode) strings.

- Some system APIs like os.environ and sys.argv can also present problems when the bytes made available by the system is not interpretable using the default encoding. Setting the LANG variable and rerunning the program is probably the best approach.

- The default source encoding is now UTF-8.

- The StringIO and cStringIO modules are gone. Instead, import the io module and use io.StringIO or io.BytesIO for text and data respectively.

New Syntax in Python 3.0

- Function argument and return value annotations.
- Keyword-only arguments. Named parameters occurring after *args in the parameter list *must* be specified using keyword syntax in the call.
- Keyword arguments are allowed after the list of base classes in a class definition.
- **nonlocal** statement. Using *nonlocal x* you can now assign directly to a variable in an outer (but non-global) scope. nonlocal is a new reserved word.
- Extended Iterable Unpacking.
  - (a, *rest, b) = range(5)
  - This sets *a* to 0, *b* to 4, and *rest* to [1, 2, 3].
- Dictionary comprehensions: {k: v for k, v in stuff} means the same thing as dict(stuff) but is more flexible.

- Set literals, e.g. {1, 2}.
  - Note that {} is an empty dictionary; use set() for an empty set.
  - Set comprehensions are also supported;
  - E.g., {x for x in stuff} means the same thing as set(stuff) but is more flexible.
- New octal literals, e.g. 0o720 (already in 2.6).
  - The old octal literals (0720) are gone.
- New binary literals, e.g. 0b1010 (already in 2.6),
  - and there is a new corresponding builtin function, bin().
- Bytes literals are introduced with a leading b or B,
  - and there is a new corresponding builtin function, bytes().

Changed Syntax in Python 3.0

- New **raise** statement syntax: raise [*expr* [from *expr*]].
- **as** and **with** are now reserved words.
- True, False, and None are reserved words.
- Change from **except** *exc*, *var* to **except** *exc* as *var*.

- New Metaclass Syntax.
  - Instead of:
    - class C: __metaclass__ = M ...
  - you must now use:
    - class C(metaclass=M): ...
  - The module-global __metaclass__ variable is no longer supported.

- List comprehensions no longer support the syntactic form [... for *var* in *item1*, *item2*, ...].
  - Use [... for *var* in (*item1*, *item2*, ...)] instead.
- The *ellipsis* (...) can be used as an atomic expression anywhere.
  - Previously it was only allowed in slices.

Removed Syntax in Python 3.0

- Tuple parameter unpacking removed.
  - You can no longer write def foo(a, (b, c)): ....
  - Use def foo(a, b_c): b, c = b_c instead.
- Removed backticks (use **repr()** instead).
- Removed <> (use != instead).
- Removed keyword:
  - **exec()** is no longer a keyword; it remains as a function.
  - Also note that **exec()** no longer takes a stream argument;
  - instead of exec(f) you can use exec(f.read()).

- Integer literals no longer support a trailing l or L.
- String literals no longer support a leading u or U.
- The **from** *module* **import** * syntax is only allowed at the module level, no longer inside functions.
- The only acceptable syntax for relative imports is
  - from .[*module*] import *name*
  - All **import** forms not starting with . are interpreted as absolute imports.
- Classic classes are gone.

Changes Already Present In Python 2.6

- *PEP 343: The 'with' statement. The with statement is now a standard feature and no longer needs to be imported from the __future__. Also check out Writing Context Managers and The contextlib module.*
- *PEP 366: Explicit Relative Imports From a Main Module. This enhances the usefulness of the -m option when the referenced module lives in a package.*
- *PEP 370: Per-user site-packages Directory.*
- *PEP 371: The multiprocessing Package.*
- *PEP 3101: Advanced String Formatting. Note: the 2.6 description mentions the format() method for both 8-bit and Unicode strings. In 3.0, only the str type (text strings with Unicode support) supports this method; the bytes type does not. The plan is to eventually make this the only API for string formatting, and to start deprecating the % operator in Python 3.1.*
- *PEP 3105: print As a Function. This is now a standard feature and no longer needs to be imported from __future__. More details were given above.*
- *PEP 3110: Exception-Handling Changes. The except exc as var syntax is now standard and except exc, var is no longer supported. (Of course, the as var part is still optional.)*
- *PEP 3112: Byte Literals. The b"..." string literal notation (and its variants like b'...', b"""...""", and br"...") now produces a literal of type bytes.*
- *PEP 3116: New I/O Library. The io module is now the standard way of doing file I/O, and the initial values of sys.stdin, sys.stdout and sys.stderr are now instances of io.TextIOBase. The builtin open() function is now an alias for io.open() and has additional keyword arguments encoding, errors, newline and closefd. Also note that an invalid mode argument now raises ValueError, not IOError. The binary file object underlying a text file object can be accessed as f.buffer (but beware that the text object maintains a buffer of itself in order to speed up the encoding and decoding operations).*
- *PEP 3118: Revised Buffer Protocol. The old builtin buffer() is now really gone; the new builtin memoryview() provides (mostly) similar functionality.*
- *PEP 3119: Abstract Base Classes. The abc module and the ABCs defined in the collections module plays a somewhat more prominent role in the language now, and builtin collection types like dict and list conform to the collections.MutableMapping and collections.MutableSequence ABCs, respectively.*
- *PEP 3127: Integer Literal Support and Syntax. As mentioned above, the new octal literal notation is the only one supported, and binary literals have been added.*
- *PEP 3129: Class Decorators.*
- *PEP 3141: A Type Hierarchy for Numbers. The numbers module is another new use of ABCs, defining Python's "numeric tower". Also note the new fractions module which implements numbers.Rational.*

Library Changes

- Many old modules were removed. Some, like gopherlib (no longer used) and md5 (replaced by **hashlib**), were already deprecated by **PEP 0004**.
- The bsddb3 package was removed because its presence in the core standard library has proved over time to be a particular burden for the core developers due to testing instability and Berkeley DB's release schedule. However, the package is alive and well, externally maintained at http://www.jcea.es/programacion/pybsddb.htm.
- Some modules were renamed because their old name disobeyed **PEP 0008**, or for various other reasons. Here's the list:
  - Old Name New Name
  - _winreg winreg
  - ConfigParser configparser
  - copy_reg copyreg
  - Queue queue
  - SocketServer socketserver
  - markupbase _markupbase
  - repr reprlib
  - test.test_support test.support

- A common pattern in Python 2.x is to have one version of a module implemented in pure Python, with an optional accelerated version implemented as a C extension; for example, pickle and cPickle.
  - This places the burden of importing the accelerated version and falling back on the pure Python version on each user of these modules.
  - In Python 3.0, the accelerated versions are considered implementation details of the pure Python versions. Users should always import the standard version, which attempts to import the accelerated version and falls back to the pure Python version.
  - The pickle / cPickle pair received this treatment. The profile module is on the list for 3.1. The StringIO module has been turned into a class in the io module.

- Some related modules have been grouped into packages, and usually the submodule names have been simplified. The resulting new packages are:
  - dbm (anydbm, dbhash, dbm, dumbdbm, gdbm, whichdb).
  - html (HTMLParser, htmlentitydefs).
  - http (httplib, BaseHTTPServer, CGIHTTPServer, SimpleHTTPServer, Cookie, cookielib).
  - tkinter (all Tkinter-related modules except turtle). The target audience of turtle doesn't really care about tkinter. Also note that as of Python 2.6, the functionality of turtle has been greatly enhanced.
  - urllib (urllib, urllib2, urlparse, robotparse).
  - xmlrpc (xmlrpclib, DocXMLRPCServer, SimpleXMLRPCServer).

- Some other changes to standard library modules, not covered by :
  - Killed sets. Use the builtin set() function.
  - Cleanup of the sys module: removed sys.exitfunc(), sys.exc_clear(), sys.exc_type, sys.exc_value, sys.exc_traceback. (Note that sys.last_type etc. remain.)
  - Cleanup of the array.array type: the read() and write() methods are gone; use fromfile() and tofile() instead. Also, the 'c' typecode for array is gone – use either 'b' for bytes or 'u' for Unicode characters.
  - Cleanup of the operator module: removed sequenceIncludes() and isCallable().
  - Cleanup of the thread module: acquire_lock() and release_lock() are gone; use acquire() and release() instead.
  - Cleanup of the random module: removed the jumpahead() API.
  - The new module is gone.
  - The functions os.tmpnam(), os.tempnam() and os.tmpfile() have been removed in favor of the tempfile module.
  - The tokenize module has been changed to work with bytes. The main entry point is now tokenize.tokenize(), instead of generate_tokens.
  - string.letters and its friends (string.lowercase and string.uppercase) are gone. Use string.ascii_letters etc. instead. (The reason for the removal is that string.letters and friends had locale-specific behavior, which is a bad idea for such attractively-named global “constants”.)
  - Renamed module __builtin__ to builtins (removing the underscores, adding an ‘s’). The __builtins__ variable found in most global namespaces is unchanged. To modify a builtin, you should use builtins, not __builtins__!

- **A New Approach To String Formatting**
  - A new system for built-in string formatting operations replaces the % string formatting operator.
  - (However, the % operator is still supported; it will be deprecated in Python 3.1 and removed from the language at some later time.)
  - Read PEP 3101 for the full scoop.

Changes To Exceptions

- The APIs for raising and catching exception have been cleaned up and new powerful features added:
- PEP 0352: All exceptions must be derived (directly or indirectly) from BaseException. This is the root of the exception hierarchy. This is not new as a recommendation, but the requirement to inherit from BaseException is new. (Python 2.6 still allowed classic classes to be raised, and placed no restriction on what you can catch.) As a consequence, string exceptions are finally truly and utterly dead.
- Almost all exceptions should actually derive from Exception; BaseException should only be used as a base class for exceptions that should only be handled at the top level, such as SystemExit or KeyboardInterrupt. The recommended idiom for handling all exceptions except for this latter category is to use except Exception.
- StandardError was removed (in 2.6 already).
- Exceptions no longer behave as sequences. Use the args attribute instead.
- PEP 3109: Raising exceptions. You must now use raise Exception(args) instead of raise Exception, args. Additionally, you can no longer explicitly specify a traceback; instead, if you have to do this, you can assign directly to the __traceback__ attribute (see below).
- PEP 3110: Catching exceptions. You must now use except SomeException as variable instead of except SomeException, variable. Moreover, the variable is explicitly deleted when the except block is left.
- PEP 3134: Exception chaining. There are two cases: implicit chaining and explicit chaining. Implicit chaining happens when an exception is raised in an except or finally handler block. This usually happens due to a bug in the handler block; we call this a secondary exception. In this case, the original exception (that was being handled) is saved as the __context__ attribute of the secondary exception. Explicit chaining is invoked with this syntax:
- raise SecondaryException() from primary_exception
- (where primary_exception is any expression that produces an exception object, probably an exception that was previously caught). In this case, the primary exception is stored on the __cause__ attribute of the secondary exception. The traceback printed when an unhandled exception occurs walks the chain of __cause__ and __context__ attributes and prints a separate traceback for each component of the chain, with the primary exception at the top. (Java users may recognize this behavior.)
- PEP 3134: Exception objects now store their traceback as the __traceback__ attribute. This means that an exception object now contains all the information pertaining to an exception, and there are fewer reasons to use sys.exc_info() (though the latter is not removed).
- A few exception messages are improved when Windows fails to load an extension module. For example, error code 193 is now %1 is not a valid Win32 application. Strings now deal with non-English locales.

Miscellaneous Other Changes

- **Operators And Special Methods**
  - != now returns the opposite of ==, unless == returns NotImplemented.
  - The concept of “unbound methods” has been removed from the language. When referencing a method as a class attribute, you now get a plain function object.
  - __getslice__(), __setslice__() and __delslice__() were killed. The syntax a[i:j] now translates to a.__getitem__(slice(i, j)) (or __setitem__() or __delitem__(), when used as an assignment or deletion target, respectively).
  - PEP 3114: the standard next() method has been renamed to __next__().
  - The __oct__() and __hex__() special methods are removed – oct() and hex() use __index__() now to convert the argument to an integer.
  - Removed support for __members__ and __methods__.
  - The function attributes named func_X have been renamed to use the __X__ form, freeing up these names in the function attribute namespace for user-defined attributes. To wit, func_closure, func_code, func_defaults, func_dict, func_doc, func_globals, func_name were renamed to __closure__, __code__, __defaults__, __dict__, __doc__, __globals__, __name__, respectively.
  - __nonzero__() is now __bool__().

- **Builtins**
  - PEP 3135 : New super(). You can now invoke super() without arguments and (assuming this is in a regular instance method defined inside a class statement) the right class and instance will automatically be chosen. With arguments, the behavior of super() is unchanged.
  - PEP 3111: raw_input() was renamed to input(). That is, the new input() function reads a line from sys.stdin and returns it with the trailing newline stripped. It raises EOFError if the input is terminated prematurely. To get the old behavior of input(), use eval(input()).
  - A new builtin next() was added to call the __next__() method on an object.
  - Moved intern() to sys.intern().
  - Removed: apply(). Instead of apply(f, args) use f(*args).
  - Removed callable(). Instead of callable(f) you can use hasattr(f, '__call__'). The operator.isCallable() function is also gone.
  - Removed coerce(). This function no longer serves a purpose now that classic classes are gone.
  - Removed execfile(). Instead of execfile(fn) use exec(open(fn).read()).
  - Removed the file type. Use open(). There are now several different kinds of streams that open can return in the io module.
  - Removed reduce(). Use functools.reduce() if you really need it; however, 99 percent of the time an explicit for loop is more readable.
  - Removed reload(). Use imp.reload().
  - Removed. dict.has_key() – use the in operator instead.

Build and C API Changes

- Due to time constraints, here is a *very* incomplete list of changes to the C API.
- Support for several platforms was dropped, including but not limited to Mac OS 9, BeOS, RISCOS, Irix, and Tru64.
- **PEP 3118** : New Buffer API.
- **PEP 3121** : Extension Module Initialization & Finalization.
- **PEP 3123** : Making PyObject_HEAD conform to standard C.
- No more C API support for restricted execution.
- PyNumber_Coerce, PyNumber_CoerceEx, PyMember_Get, and PyMember_Set C APIs are removed.
- New C API PyImport_ImportModuleNoBlock, works like PyImport_ImportModule but won't block on the import lock (returning an error instead).
- Renamed the boolean conversion C-level slot and method: nb_nonzero is now nb_bool.
- Removed METH_OLDARGS and WITH_CYCLE_GC from the C API.

Porting To Python 3.0

- For porting existing Python 2.5 or 2.6 source code to Python 3.0, the best strategy is the following:
  - (Prerequisite:) Start with excellent test coverage.
  - Port to Python 2.6. This should be no more work than the average port from Python 2.x to Python 2.(x+1). Make sure all your tests pass.
  - (Still using 2.6:) Turn on the *-3* command line switch. This enables warnings about features that will be removed (or change) in 3.0. Run your test suite again, and fix code that you get warnings about until there are no warnings left, and all your tests still pass.
  - Run the 2to3 source-to-source translator over your source code tree. (See *2to3 - Automated Python 2 to 3 code translation* for more on this tool.) Run the result of the translation under Python 3.0. Manually fix up any remaining issues, fixing problems until all tests pass again.
- It is not recommended to try to write source code that runs unchanged under both Python 2.6 and 3.0; you'd have to use a very contorted coding style, e.g. avoiding print statements, metaclasses, and much more. If you are maintaining a library that needs to support both Python 2.6 and Python 3.0, the best approach is to modify step 3 above by editing the 2.6 version of the source code and running the 2to3 translator again, rather than editing the 3.0 version of the source code.
- For porting C extensions to Python 3.0, please see *Porting Extension Modules to 3.0*.