

# 점프 투 파이썬

박응용

2015년 9월 21일

# 차례

※ 문서정보 . . . . .	2
1. Endless-Edition을 계획하며 . . . . .	3
1) 주요변경이력 . . . . .	5
2. 들어가기 전에 . . . . .	6
책의 구성 . . . . .	6
1) 머리말 . . . . .	8
2) 추천사 . . . . .	9
3. 파이썬이란 무엇인가? . . . . .	13
1) 파이썬의 특징 . . . . .	14
2) 무엇을 할 수 있나? . . . . .	18
파이썬으로 할 수 있는 일 . . . . .	18
파이썬으로 할 수 없는 일 . . . . .	20
3) 파이썬 설치하기 . . . . .	21
원도우에서 파이썬 설치 . . . . .	21
리눅스에서 파이썬 설치 . . . . .	22
4) 파이썬 둘러보기 . . . . .	24
따라해 보기 . . . . .	24
에디터로 파이썬 프로그램 작성하기 . . . . .	28
4. 자료형 . . . . .	31
1) 숫자형 (Number) . . . . .	32
숫자 연산 . . . . .	35

2) 문자열 (String) . . . . .	38
문자열 연산 . . . . .	42
인덱싱과 슬라이싱 . . . . .	44
문자열 포매팅(Formatting) . . . . .	52
문자열 함수 . . . . .	57
3) 리스트 (List) . . . . .	66
리스트의 인덱싱과 슬라이싱 . . . . .	67
리스트의 인덱싱 . . . . .	67
리스트의 슬라이싱 알아보기 . . . . .	70
리스트 관련 함수들 . . . . .	75
4) 튜플 (tuple) . . . . .	81
튜플의 인덱싱, 슬라이싱, 더하기와 반복 . . . . .	82
5) 딕셔너리 (Dictionary) . . . . .	85
딕셔너리는 어떻게 생겼을까? . . . . .	85
딕셔너리 사용하기 . . . . .	86
딕셔너리 주의사항 . . . . .	90
딕셔너리 관련함수 . . . . .	91
6) 집합 (Sets) . . . . .	96
set의 특징 . . . . .	96
교집합, 합집합, 차집합 . . . . .	97
추가와 삭제 . . . . .	99
7) 참과 거짓 . . . . .	101
8) 변수 . . . . .	104
변수란 . . . . .	104
리스트 복사 . . . . .	107
5. 제어문 . . . . .	110
1) if문 . . . . .	111
if문의 기본 구조 . . . . .	111
조건문이란 무엇인가? . . . . .	114

2) while문 . . . . .	124
무한루프(Loop) . . . . .	125
while문 빠져 나가기(break) . . . . .	127
조건문으로 돌아가기(continue) . . . . .	130
3) for문 . . . . .	132
for문의 기본구조 . . . . .	132
예제를 통해 for 알아보기 . . . . .	132
for와 continue . . . . .	134
for와 range함수 . . . . .	135
리스트 내포(List comprehension) . . . . .	138
6. 입출력 . . . . .	141
1) 함수 . . . . .	142
함수를 사용하는 이유? . . . . .	143
파이썬 함수의 구조 . . . . .	143
함수의 입력값과 리턴값 . . . . .	145
입력값에 초기치 설정하기 . . . . .	154
함수 내에서 선언된 변수의 효력 범위 . . . . .	157
2) 입력과 출력 . . . . .	160
사용자 입력 . . . . .	160
print 자세히 알기 . . . . .	161
3) 파일 읽고 쓰기 . . . . .	164
파일 생성하기 . . . . .	164
파일을 쓰기 모드로 열어서 출력값 적기 . . . . .	165
파일을 읽는 여러가지 방법 . . . . .	167
파일에 새로운 내용 추가하기 . . . . .	170
with문과 함께 사용하기 . . . . .	170
sys모듈 입력 . . . . .	171
7. 파이썬 날개달기 . . . . .	174
1) 클래스 . . . . .	175

클래스는 도대체 왜 필요하게 되었을까? . . . . .	175
클래스(class) 개념잡기 . . . . .	179
조금 쉽게 이해해 보기 . . . . .	180
이야기 형식으로 클래스 기초쌓기 . . . . .	181
클래스 변수 . . . . .	181
클래스 함수 . . . . .	182
self 제대로 알기 . . . . .	186
__init__ 이란 무엇인가? . . . . .	190
클래스 자세히 알기 . . . . .	192
클래스의 구조 . . . . .	192
사칙연산 하는 클래스 만들기 . . . . .	193
“박씨네 집” 클래스 . . . . .	204
초기값 설정하기 . . . . .	209
__init__ 메쏘드, 초기치를 설정한다. . . . .	209
상속 (Inheritance) . . . . .	211
연산자 오버로딩(Overloading) . . . . .	214
2) 모듈 . . . . .	220
모듈 만들고 불러보기 . . . . .	220
if __name__ == "__main__": 의 의미 . . . . .	224
클래스나 변수등을 포함한 모듈 . . . . .	226
다른 프로그램 파일에서 만든 모듈 불러오기 . . . . .	228
모듈 불러오는 또 다른 방법 . . . . .	229
3) 패키지 . . . . .	232
패키지 만들기 . . . . .	233
__init__.py 의 용도 . . . . .	236
__all__ 의 용도 . . . . .	237
relative 패키지 . . . . .	238
4) 예외처리 . . . . .	240
에러는 어떤 때 일어나는가? . . . . .	240

에러 처리하기 . . . . .	241
try .. else . . . . .	244
try .. finally . . . . .	244
에러 회피하기 . . . . .	245
에러 발생시키기](raise) . . . . .	245
5) 내장함수 . . . . .	248
abs . . . . .	248
all . . . . .	249
any . . . . .	250
chr . . . . .	250
dir . . . . .	251
divmod . . . . .	251
enumerate . . . . .	252
eval . . . . .	252
filter . . . . .	253
hex . . . . .	254
id . . . . .	255
input . . . . .	255
int . . . . .	256
isinstance . . . . .	257
lambda . . . . .	258
len . . . . .	259
list . . . . .	260
map . . . . .	261
max . . . . .	263
min . . . . .	263
oct . . . . .	263
open . . . . .	264
ord . . . . .	265

pow . . . . .	265
range . . . . .	266
repr . . . . .	267
sorted . . . . .	267
str . . . . .	268
tuple . . . . .	269
type . . . . .	269
zip . . . . .	270
6) 외장함수 . . . . .	271
명령행에서 인수를 전달(sys.argv) . . . . .	271
객체를 그 상태 그대로 파일에 저장하고 싶을 때(pickle) . . . . .	273
현재 내 시스템 환경변수값을 알고싶을 때는? (os.environ) . . . . .	274
디렉토리에 대한 것들(os.chdir, os.getcwd) . . . . .	275
시스템 명령(os.system, os.popen) . . . . .	275
파일 복사(shutil) . . . . .	277
디렉토리에 있는 파일들을 리스트로 만들려면 (glob) . . . . .	277
임시파일 (tempfile) . . . . .	278
시간에 관한 것들(time) . . . . .	278
파이썬에서 달력쓰기(calendar) . . . . .	282
난수 발생시키기 (random) . . . . .	284
파이썬에서의 쓰레드 (threading) . . . . .	286
웹브라우저 실행시키기 (webbrowser) . . . . .	289
8. 어디서부터 시작할 것인가? . . . . .	291
1) 내가 프로그램을 만들 수 있을까? . . . . .	292
2) 간단한 메모장 . . . . .	297
3) tab을 4개의 space로 바꾸기 . . . . .	299
4) 게시판 페이징 . . . . .	301
5) 하위디렉토리 검색 . . . . .	303
6) 3과 5의 배수 합하기 . . . . .	306

7) 코딩도장 . . . . .	308
9. 파이썬 정규표현식과 XML . . . . .	309
1) 정규표현식 . . . . .	310
정규표현식은 왜 필요한가? . . . . .	311
정규표현식의 기초 . . . . .	313
정규표현식 시작하기 . . . . .	320
강력한 정규표현식의 세계로 . . . . .	333
2) XML 처리 . . . . .	349
ElementTree . . . . .	349
XML문서 생성하기 . . . . .	349
XML문서 파싱하기 . . . . .	354
A. 부록 . . . . .	358
1) 파이썬 2.7 vs 파이썬 3 . . . . .	359
print . . . . .	359
자동 형 변환 . . . . .	360
input . . . . .	361
소스코드 인코딩 . . . . .	361
에러처리 . . . . .	362
2) 파이썬과 에디터 . . . . .	363
에디트 플러스(Edit Plus) . . . . .	363
3) 파이썬과 다른 언어들과의 비교 . . . . .	367
4) 유용한 파이썬 문서들 . . . . .	371
유용한 온라인 문서들 . . . . .	373

## ※ 문서정보

Copyright © 2013 박응용. All rights reserved.

이 책의 무단전재와 복제를 금합니다

이 책은 석재욱(ice273k@naver.com) 님이 구매하신 문서입니다.

(구매 : <http://wikidocs.net/book/1>)

## 1. Endless-Edition을 계획하며

“점프 투 파이썬”은 원래 2001년에 정보게이트란 출판사를 통해서 출간된 책입니다. 이미 절판되었기 때문에 서점에서 구할 수는 없는 상태입니다.

대학 4학년 시절에 집필한 책이어서 지금 다시 읽어보면 부끄러운 마음을 금 할 수가 없습니다. 하지만 지금 다시 작성한다 하더라도 더 좋은 책을 만들 수 있다고는 장담할 수 없을 것 같습니다. 왜냐하면 이 책에는 초보 프로그래머가 프로그래밍을 공부하면서 겪게되는 어려움에 대한 얘기들이 고스란히 들어있기 때문입니다. 초보 프로그래머의 경험을 반영한다고 해야 할까요? 아무튼 이제 막 프로그래밍을 시작하려는 분들에게는 분명 도움이 될 수 있을 거라고 생각 합니다.

이제 이 책을 집필한지 10년이 지났고 저에게 프로그래머로서의 10년 경험이 생겼습니다. 초보시절의 마음과 10년의 프로그래밍 경험을 잘 섞으면 더 좋은 책이 되리라 기대합니다. 시간이 날 때마다 틈틈히 삭제할 부분은 삭제하고 보강해야 할 부분은 보강하며 “점프 투 파이썬”을 다듬어 나가고 있는 중입니다.

Endless Edition이란 끊임없이 변화하는 책을 의미합니다.

Endless Edition은 다음과 같은 모토로 온라인 상에서 계속 진화해 나갈 예정입니다:

어제의 “점프 투 파이썬”과 오늘의 “점프 투 파이썬”은 다르다!

여러분도 저와 함께 좀 더 나은 “점프 투 파이썬”을 만들어가고 싶지 않으신가요?

여러분은 아래와 같은 방법으로 “점프 투 파이썬”을 변화시킬 수 있습니다.

- 잘못된 예나 더욱 좋은 예를 알려주기
- 오타 잡아주기
- 파이썬 관련한 문서 작성하여 “점프 투 파이썬”에 기고하기

감사합니다.

어제와 다른 어느날

박용용

## 1) 주요변경이력

점프 투 파이썬은 오프라인 서적에서 시작하여 무려 10년 이상을 거듭 발전해오고 있는 온라인 서적입니다.

다음은 점프 투 파이썬의 주요변경이력입니다.

순서	주요 내용	날짜
1	점프 투 파이썬 오프라인 책 출간 (정보제이트)	2001.09
2	개인 위키에 “점프 투 파이썬” 공개	2006.03
3	위키독스에 “점프 투 파이썬” 무료 온라인 책 출간	2008.03
4	파이썬 버전 3으로 개정	2013.05
5	전자책(E-book) 판매 시작	2013.10
6	파이썬 버전 통합 (Python 2.7 + Python 3)	2014.02
7	Sets 챕터 추가	2014.06
8	패키지(Packages) 챕터 추가	2014.09
9	정규표현식과 XML 챕터 추가	2014.09

## 2. 들어가기 전에

이 책은 파이썬이란 언어를 처음 접해보는 독자들과 프로그래밍을 한 번도 해 본적이 없는 사람들을 대상으로 한다. 프로그래밍을 할 때 사용되는 전문적인 용어들을 알기 쉽게 풀어서 쓰려고 노력하였으며, 파이썬이란 언어의 개별적인 특성만을 강조하지 않고 프로그래밍 전반에 관한 사항을 파이썬이란 언어를 통해 알 수 있도록 알기 쉽게 설명하였다. 파이썬에 대한 기본적인 지식을 알고 있는 사람이라도 이 책은 파이썬 프로그래밍에 대한 흥미를 가질 수 있는 좋은 안내서가 될 것이다.

이 책의 목표는 독자가 파이썬을 통해 프로그래밍에 대한 전반적인 이해를 갖게하는 것이며, 또 파이썬이라는 도구를 이용하여 원하는 프로그램을 쉽고 재미 있게 만들 수 있게 하는 것이다.

### [파이썬 2.7 or 파이썬 3]

첨프 투 파이썬은 파이썬 3 버전을 기준으로 설명하고 있지만 파이썬 2.7을 사용하는 경우에도 무리없이 책을 읽을 수 있도록 서로 다른부분에 대한 설명도 포함하고 있다. (since: 2014.02) (참고: [파이썬 2.7 vs 파이썬 3](#))

## 책의 구성

[들어가기전에] 에서는 이 책이 어떻게 시작해서 여기까지 진행되어 왔는지, 누가 이 책을 읽어야 하는지에 대해서 설명한다. 그리고 국내 파이썬 역사에 큰 기여를 한 이만용님의 추천사를 소개한다.

[파이썬이란 무엇인가?] 에서는 파이썬으로 할 수 있는 일과 할 수 없는 일과 파이썬만의 독특한 특징들에 대해서 설명한다. 그리고 파이썬 문법을 처음부터 끝까지 아주 간단하게 둘러봄으로서 앞으로 공부해야 할 문법들에 대해서 개략적으로 알아본다.

[자료형] 과 [제어문] 에서는 문자열과 리스트와 같은 파이썬 자료형과 if, for, while등의 파이썬 제어문에 대해서 설명한다.

[입출력] 에서는 파이썬의 함수, 입력과 출력, 파일을 읽고 쓰는 방법에 대해서 설명한다.

[파이썬 날개달기] 에서는 지금까지 배워온 아주 간단한 파이썬 문법과 자료형들을 기반으로 이제 조금 어려운 것들에 대해서 배워본다. 객체지향 프로그래밍의 핵심인 클래스와 재사용을 위한 모듈 그리고 유연한 프로그래밍을 할 수 있게 해 주는 예외처리에 대해서 설명한다.

[어디서부터 시작할 것인가?] 에서는 도대체 어디서부터 프로그래밍을 시작할 것인지 망설이는 독자들을 위해서 프로그래밍에 대한 감을 잡을 수 있는 간단한 예제들을 소개한다.

[파이썬 정규표현식과 XML] 에서는 초보 프로그래머에게 어울리지 않는 고급 주제이지만 한번 도전해 볼만하고 매우 가치가 있는 정규표현식과 XML에 대해서 설명한다.

[부록] 에서는 파이썬2와 파이썬3의 차이점에 대해서 간단하게 설명한다. 파이썬과 궁합이 잘 맞는 에디터 사용법에 대해서 설명한다.

## 1) 머리말

필자는 파이썬의 >>> 프롬프트를 처음 보았던 순간부터 지금까지 파이썬과 함께 지내온 듯 하다. 항상 “프로그래밍은 어렵고 지루한 것이다.”라는 고정관념을 가지고 있었던 필자에게 파이썬은 커다란 충격으로 다가왔다. 파이썬의 깔끔한 문장구조와 프로그래밍이 정말로 즐겁게 느껴지게 하는 야릇한 매력은 지금껏 경험해 보지 못한 것이었다. 아직 파이썬의 향기를 맡지 못한 독자라면 꼭 파이썬이란 언어를 느껴볼 것을 당부한다. 분명 파이썬의 매력에 흠뻑 젖게 될 것이다.

이 책을 쓸 수 있도록 도와주신 많은 분들께 감사의 말을 전하고 싶다.

이 책이 나올 수 있도록 여러모로 힘써 주신 정보게이트 관계자 여러분, 특히 무엇보다도 처음으로 책을 써 보는 필자에게 큰 힘이 되어주신 유혜규 기획실장님께 감사를 드린다.

그리고 한번도 프로그래밍을 해 본적이 없는 작은형이 지적해준 사항은 이 책을 초보자가 이해하기 쉽게 쓰도록 해 주었는데, 꼼꼼하게 원고를 살펴보아 주었던 석이 형에게 이 책을 통해서 감사의 마음을 표하고 싶다. 또한 내가 편안한 마음으로 책을 쓸 수 있게 많은 배려를 해 주었던 가족들에게 고마운 마음을 전하고 싶다.

항상 격려와 도움을 주었던 학교 친구들에게도 고마운 마음을 전하고 싶다. 특히 내가 힘들어 할 때 나의 넋두리를 받아주고 많은 충고를 해 주었던 형석, 승용에게 이 책을 빌어 고마운 마음을 전하고 싶다.

마지막으로 바쁜 와중에도 책의 내용을 검토해 주시고 서문을 써 주신 리눅스 코리아의 이만용 이사님께 진심으로 감사의 뜻을 전하고 싶다. 또한 국내 파이썬 보급과 발전에 많은 힘을 기울이는 파이썬 사용자모임의 여러분들 모두에게 감사의 마음을 전하고 싶다.

## 2) 추천사

리눅스코리아 CTO 이만용

yong@linuxkorea.co.kr

프로그래머에게 있어 언어의 선택은 중요하다. 특히, 프로그래머가 되고자 마음 먹은 사람에게 있어 ‘첫 번째’ 언어의 선택은 더더욱 중요하다. 화가에게 양질의 붓과 캔버스가 필요하고 음악가에게 훌륭한 악기가 필요하듯, 프로그래머에게는 프로그래밍이라는 창조적 두뇌 작업을 도와 줄 훌륭한 프로그래밍 언어가 필요하다. 인간은 도구의 동물이라 하지 않았던가? 인간이 도구를 만들 듯, 도구는 또 다시 인간을 만들어준다는 의미에서 도구의 선택 행위는 인간의 가장 중요한 선택 행위 중 하나이다.

수많은 프로그래머 지망생들은 프로그래밍 언어를 선택함에 있어 처음부터 장벽에 부딪히기 시작한다. 내게 어떤 프로그래밍 언어를 해야 할지 물어오는 많은 고등학생, 대학생들은 마이크로소프트사나 썬 마이크로시스템즈사가 프로그래밍 잡지를 통해 열심히 홍보하는 주류 언어만을 나열하면서 ‘낙점’을 해 달라고 요청한다. 역시 비주얼 C++을 해야겠죠? 자바(Java)가 대세이니 자바를 먼저 해야겠죠? 아니면 주위 얘기를 들어보니 프로그래머의 근본을 갖추기 위해서는 C 언어부터 다시 해야겠죠? (비주얼 C++은 제품 이름일 뿐인데 C++ 언어와 착각하는 사람도 많다. 현재 많은 사람들이 제품과 언어를 착각하곤 한다.) 그들의 질문은 크게 다르지 않다. 그들은 빨리 프로그래밍을 하고 싶어하면서도 동시에 누구나 다 사용하는 현실적인 언어를 선택하고 싶어한다. 누군들 그리고 싶지 않은 사람이 있을까?

지난 1999년 파이썬(Python)이라는 언어를 알기 전까지 이 모든 대답에 통쾌한 답변을 주지 못했다. 언어란 결국 도구이므로 자기에게 가장 알맞은 것을 선택하라는 답변 뿐 현실적인 대안을 제시하기 힘들었다. 그러나 이제는 자신감을 가지고 고민하는 모든 프로그래머와 프로그래머가 되고자 마음먹은 초보 프로그래머들에게 파이썬이라는 언어를 추천할 수 있다.

파이썬은 그 자체로 매우 훌륭한 설계를 가지고 있으며(수학적 정확성 속에서 탄생하였다) 배우기 쉽고 현대적인 설계를 가지고 있으며 또한 '현실적'이고 자하는 많은 사람들을 위해 현실적인 강력함도 고루 갖춘 언어이다. 다른 언어와 비교하지 않아도 그 자체로 가치있는 프로그래밍 언어이다(본인과 본인의 회사 개발팀은 글자 그대로 100% 모든 프로젝트를 파이썬으로 진행하고 있다.)

그러나 이 글의 필자와 본인은 그냥 또 다른 언어 중 하나를 여러분에게 추천하는 것이 아니다. 이 세상에는 이미 프로그래밍 언어가 넘쳐난다. 이 글과 책은 여러분이 파이썬을 통해 2000년대의 프로그래밍 언어가 어떤 것이어야 하며, 프로그래머에게 있어 언어 선택이란 어떤 것인지, 그리고 결국에는 파이썬이든 그 무엇이든 상관없이 자기의 필요에 따라 어떻게 언어를 선택하고 언어를 창조해 나갈 수 있는지를 보여주고자 할 것이다.

프로그래밍에 있어 중요한 것 중 하나가 바로 스타일(Style)이다. 여러분이 생각하고 코딩하는 스타일을 제대로 배우지 못하면 즐길 줄 아는 프로그래머가 되기 어렵다. 훌륭한 스타일은 좀 더 높은 상상력을 낳고 프로그래밍을 재미(fun)로서 향유할 수 있는 여유를 가져다 준다. 직업 프로그래머가 되고자 하는 사람이라면 파이썬이든 자바든 하나의 언어만 가지고 살아갈 수는 없다. 여러분의 의지와 상관없이 독점적 지위를 가진 업체에서 언어를 하나 새롭게 만들어서 추진하면 배울 수밖에 없는 것이 현실이다(예를 들어 마이크로소프트사의 C#이 그러하다). 따라서 더더욱 중요한 것은 그 어떤 언어에도 쉽게 적응할 수 있도록 해 주는 기본기와 스타일이다. 바로 그것을 파이썬 언어에서 배울 수 있기를 기대한다. 무작정 C++이나 자바를 먼저 배우는 것보다 여유를 가지고 파이썬을 배우고 점차로 C++이나 자바를 배워간다면 오히려 더 빨리 더 많은 것을 성취할 수 있다. 그것이 바로 스타일의 힘이다. 사실 이 책을 들고 있는 독자의 대부분은 어디에선가 파이썬의 '명성'을 들었을 것이다. 1~2년전에는 꿈도 꾸지 못할 일이다. 리눅스처럼 강력한 오픈 소스 프로젝트 중 하나가 되어버린 파이썬은, 언어의 창조자 Guido van Rossum의 노력은 물론이고 파이썬을 현실적으로도 강력한 언어가 되도록 모듈을 만들어내고 있는 전세계 자발적인 프로그래머, 그리고 한국 파이썬 사용자모임에서 각종 세미나와 홍보 활동을 적

극 별이고 있는 회원들의 노력(이미 한국은 전세계적으로도 파이썬 열기가 가장 높은 나라 중 하나이다)을 통해 많은 사람들에게 알려져가고 있기 때문이다. 여러분도 파이썬을 자기 것으로 만든 후, 이 대열에 동참할 수 있기를 바란다. 파이썬을 배우면서 파이썬의 창조자 대열에 설 수 있다면 여러분은 이미 최고의 프로그래머가 되었다고 말할 수 있다.

이 글을 쓴 저자와 본인은 아직 한 번도 만나지 못했다. 그러나 같이 파이썬에 미쳐서 프로그래밍의 참 재미(fun)를 만끽할 수 있는 사람이라는 동지애를 느끼지 않을 수 없다. 그 동안 훌륭한(또는 대충 훌륭한) 외국 파이썬 서적이 몇 권 나왔다. 그러나 언어적 장벽 때문인지 초보 파이썬 입문자에게는 어려운 점이 없지 않았다. 추천사를 쓰기 위해 원고를 모두 읽으면서 몇 가지 미흡한 점을 발견하지 않은 것은 아니나, 적어도 이 책의 내용은 저자가 내면화한 뒤 다시 써내려간 내용이므로 의미를 갖는다. 안타까운 국내 출판 실정 때문에 나올 수밖에 없는 조잡한 번역서류에서 얻을 수 없는 것은 분명히 얻을 수 있다. 한국인 저자가 한국인을 위해 쓴 첫 번째 책 중 하나라는 사실을 높이 평가하고 싶다.

파이썬은 여러분에게 그 자체로 흥미로운 언어라고 확신한다. 그러나 무엇보다도 프로그래머는 주체적으로 언어를 선택하고 새로운 언어를 계속 습득해야 한다는 깨달음을 얻기 바란다. 그것이 저자와 본인의 똑 같은 소망일 것이다.

마지막으로 파이썬을 통해 프로그래밍이란 원래 지적인 재미(fun)에 미치는 일이라는 사실을 느낄 수 있기 바란다. 본인은 사람들에게 매번 이 말을 강조한다. “똑똑한 사람은 결국 성실한 사람을 이길 수 없다. 그러나 성실한 사람이라도 이길 수 없는 사람이 있으니, 바로 그 일에 미친 사람이다.” 여러분이 개인적으로나 사회적으로나 성공하는 전문 프로그래머가 되기 위해서는 미치지 않으면 안된다. 그 일을 파이썬이 도와 줄 것이다.

이만 본인은 파이썬 프로젝트 Na에 대한 구상을 시작하고자 한다.  
한국 파이썬 사용자모임에서 만날 수 있기를 바란다.

2001년 7월

Happy Python!

### 3. 파이썬이란 무엇인가?



파이썬이란 1990년 암스테르담의 귀도 반 로섬(Guido Van Rossum)에 의해 만들어진 인터프리터 언어이다. 귀도는 이 파이썬이라는 이름을 본인이 좋아하는 “Monty Python’s Flying Circus”라는 코미디 쇼에서 따왔다고 한다. 파이썬(Python)의 사전적인 뜻은 고대 신화 속의 파르나수스(Parnassus) 산의 동굴에 살던 큰 뱀으로서, 아폴로가 델파이에서 파이썬을 퇴치했다는 이야기가 전해지고 있다. 대부분의 파이썬 책 표지와 아이콘이 뱀 모양으로 그려져 있는 이유가 여기에 있다.

현재 파이썬은 국내에서는 많이 알려져 있지 않지만 외국에서는 학습의 목적뿐만 아니라 실용적인 부분에서도 많이 사용되고 있는데 그 대표적인 예는 바로 구글(Google)이다. 구글에서 만들어진 소프트웨어의 50%이상이 파이썬으로 만들어졌다고 한다. 이 외에도 유명한 것을 몇가지 들어보면 Dropbox(파일 동기화 서비스), Django(파이썬 웹 프레임워크) 등을 들 수 있다.

또한 파이썬 프로그램은 공동작업과 유지보수가 매우 쉽고 편하기 때문에 이미 다른 언어로 작성된 많은 프로그램과 모듈들이 파이썬으로 다시 재구성되고 있는 상황이다. 국내에서도 그 가치를 인정받아 사용자층이 더욱 넓어져 가고 있고, 파이썬을 이용한 프로그램을 개발하는 기업체들이 늘어가고 있는 추세이다.

이제 파이썬의 특징과 장단점, 실제로 파이썬 프로그래밍을 하기 위한 환경을 구축하는 법에 대해서 알아보고 실제로 간단한 파이썬 프로그램을 작성해볼 것이다.

## 1) 파이썬의 특징

필자는 파이썬을 무척이나 좋아한다. 모든 프로그래밍 언어에는 각기 나름대로의 장점이 있지만 파이썬에는 다른 언어들에서 쉽게 찾아볼 수 없는 파이썬만의 독특한 특징들이 있기 때문이다. 이 특징들을 살펴보면 파이썬의 매력을 흠뻑 느낄 수 있을 것이다. 파이썬의 특징들을 알아보는 과정을 통해서 왜 파이썬을 공부해야 하는지, 과연 이것에 시간을 투자할 만한 가치가 있는 것인지에 대한 독자의 판단은 분명해질 것이다.

### 인간다운 언어이다

프로그래밍이란 컴퓨터에 인간이 생각하는 것을 입력시키는 행위라고 할 수 있다. 앞으로 살펴볼 파이썬 문법들에서도 볼 수 있겠지만 파이썬은 사람이 생각하는 방식을 그대로 표현할 수 있도록 해 주는 언어이다. 따라서 프로그래머는 굳이 컴퓨터식 사고 방식으로 프로그래밍을 하려고 애쓸 필요가 없다. 이제 곧 어떤 프로그램을 구상하자마자 생각한 대로 쉽게 술술 써내려가는 여러분의 모습에 놀라게 될 것이다. 아래 예문을 보면 이 말이 쉽게 이해될 것이다.

```
if 4 in [1,2,3,4]: print ("4가 있습니다")
```

위의 예제는 다음처럼 읽을 수 있다:

“만약 4가 1,2,3,4중에 있으면” 4가 있습니다“를 출력한다.”

프로그램을 모르더라도 직관적으로 무엇을 뜻하는지 알 수 있지 않겠는가? 마치 영어문장을 읽는 듯한 착각에 빠져든다.

## 문법이 쉬워 빠르게 학습할 수 있다

어려운 문법과 수많은 규칙들에 둘러싸인 언어에서 탈피하고 싶지 않은가? 파이썬은 문법 자체가 아주 쉽고 간결하며, 사람의 사고 방식과 매우 닮아있다. 배우기 쉬운 언어, 활용하기 쉬운 언어가 가장 좋은 언어가 아닐까? 참고로 프로그래밍 경험이 있는 어떤 사람(Eric Raymond)은 파이썬을 공부한지 단 하루만에 자신이 원하는 프로그램을 작성할 수 있었다고 한다.<sup>1</sup>

## 강력하다

파이썬으로 프로그래머는 대부분의 모든 일들을 해낼 수가 있다. 물론 시스템 프로그래밍, 하드웨어 제어, 매우 복잡하고 많은 반복연산 등은 파이썬과는 어울리지 않는다. 하지만 이러한 몇 가지를 제외하면 파이썬으로 할 수 없는 것은 거의 없다고 해도 과언이 아니다.

또한 파이썬은 위의 약점을 극복할 수 있게끔 다른 언어로 만든 모듈을 파이썬 프로그램에 포함할 수가 있다. 파이썬과 C는 찰떡궁합이란 말이 있다. 즉, 프로그램의 전반적인 뼈대는 파이썬으로 만들고 빠른 속도를 필요로 하는 부분은 C로 만들어서 파이썬 프로그램 안에 포함시키자는 것이다. (정말 놀라울 정도로 영악한 언어가 아닌가?) 사실 파이썬 라이브러리 중에는 순수 파이썬만으로 제작된 것도 많지만 C로 만들어진 것도 많다. C로 만들어진 것들은 대부분 속도가 빠르다.

## 무료이다

파이썬은 오픈소스이므로 당연히 무료이다. 언제 어디서든 파이썬을 다운로드하여 사용할 수 있고, 사용료를 지불해야 할 필요가 없다.

---

<sup>1</sup>[저자주] 보통 프로그래밍 경험이 있는 평범한 사람이라면 기본적인 파이썬 프로그래밍을 익히는 데에는 1주일이면 충분하리라 생각된다.

## 간결하다

파이썬은 간결하다. 이 간결함은 파이썬을 만든 귀도(Guido)의 의도적인 산물이다. 만약 어떤 언어(Perl?)가 하나의 일을 하기 위한 방법이 100가지라면 파이썬은 가장 좋은 방법 1가지를 선호한다. 이 파이썬의 간결함이란 철학은 소스코드에도 그대로 적용되어 파이썬 프로그래밍을 하는 사람들은 잘 정리되어 있는 소스코드를 볼 수 있게 되었다. 다른 사람들의 소스 코드가 한눈에 들어오기 때문에 이 간결함은 공동 작업에 매우 큰 역할을 하게 되었다. 다음은 파이썬 프로그램의 예제이다:

```
# simple.py
languages = ['python', 'perl', 'c', 'java']

for lang in languages:
    if lang in ['python', 'perl']:
        print("%6s need interpreter" % lang)
    elif lang in ['c', 'java']:
        print("%6s need compiler" % lang)
    else:
        print("should not reach here")
```

위의 프로그램 소스 코드를 이해하려 하기는 말자. 이것을 이해할 수 있다면 당신은 이미 파이썬에 중독된 사람일 것이다. 그냥 한번 구경해 보도록 하자. 다른 언어들에서 늘 보이는 단락을 구분하는 괄호({, })문자들이 보이지 않는 것을 확인 할 수 있다. 또한, 줄을 참 잘 맞춘 코드라는 것도 확인 할 수 있다. 줄을 맞추지 않으면 실행이 되지 않는다. 파이썬 프로그래머는 코드를 이쁘게 작성하려고 저렇게 줄맞추어 코딩을 하는것이 아니다. 다만 실행이 되게 하기 위해서 줄을 맞추어야 하는 것이다. 이렇듯 줄을 맞추어 코드를 작성하는 행위는 가독성에 큰 도움을 준다.

### **프로그래밍이 재밌다**

이 부분이 가장 강조하고 싶은 부분이다. 필자에게 파이썬만큼 프로그래밍을 즐기게 해준 언어는 없었던 것 같다. 파이썬은 다른 것에 신경 쓸 필요 없이 내가 하고자 하는 부분에만 집중할 수 있게 해주었기 때문이다. 리눅스 텔넷은 재미로 리눅스를 만들었다고 하지 않는가? 파이썬을 배우고 나면 다른 언어로 프로그래밍을 하는 것에 지루함을 느끼게 될지도 모른다. 조심하자!!

### **개발속도가 빠르다**

마지막으로 다음의 재미있는 말로 파이썬의 특징을 마무리하려 한다.

*Life is too short, You need python.*

파이썬의 엄청나게 빠른 개발 속도를 두고 유행처럼 퍼진 말이다. 이 유머러스한 문장은 이 책에서 계속 예제로 사용할 것이다.

## 2) 무엇을 할 수 있나?

좋은 프로그래밍 언어와 나쁜 프로그래밍 언어는 이미 정해진 걸까? 그렇다면 어떤것이 최고의 언어일까? 가만히 살펴보면 어떤 언어든지 강한 부분과 약한 부분이 존재한다. 어떤 프로그램을 만들 것인지에 따라 선택해야 할 언어도 달라진다. 한 언어만을 고집하고 그 언어로만 모든 것을 하겠다는 생각은 현실과는 맞지 않는다. 따라서 자신이 만들고자 하는 프로그램을 가장 잘 만들 수 있게 도와주는 언어가 어떤 것인지 알아내고 선택하는 것은 중요한 일이다. 하지만 할 수 있는 일과 할 수 없는 일을 가리기는 쉽지 않다. 왜냐하면 어떤 언어든지 할 수 없는 일은 별로 없기 때문이다. 하지만 한 프로그래밍 언어가 어떤 일에 적합한지에 대해서 아는 것은 매우 중요하다. 따라서 파이썬으로 하기에 적당한 일과 적당하지 않은 일에 대해서 알아보는 것은 매우 가치있는 일이 될 것이다.

### 파이썬으로 할 수 있는 일

파이썬으로 할 수 있는 일은 너무나 많다. 대부분의 컴퓨터 언어가 하는 일을 파이썬은 쉽고 깔끔하게 처리한다. 이것들에 대해서 나열하자면 끝도 없겠지만 대표적인 몇 가지의 예를 들어보도록 하자.

#### 1. 시스템 유ти리티

파이썬은 운영체제(윈도우즈, 리눅스등)의 시스템 명령어들을 이용할 수 있는 도구들을 갖추고 있기 때문에 이러한 것들을 바탕으로 갖가지 시스템 관련한 유ти리티를 만드는 데 유리하다. 여러분은 시스템에서 사용중인 다른 유ти리티성 프로그램들을 하나로 뭉쳐서 큰 힘을 발휘하게 하는 프로그램들을 무수히 만들어 낼 수 있다.

## 2. GUI(Graphic User Interface) 프로그램

GUI 프로그래밍이라는 것은 쉽게 말해서 윈도우즈 창같은 프로그램을 만드는 것이다. 파이썬으로 GUI프로그램을 작성하는 것은 다른 언어로 하는 것보다 훨씬 쉽다. 대표적인 것으로 파이썬 프로그램을 설치할 때 함께 설치되는 Tkinter 를 들 수 있다. 실제로 Tkinter를 이용한 파이썬 GUI프로그램의 프로그램 소스는 매우 간단하다. 놀라운 사실은 Tkinter를 이용하면 소스코드 단 5줄 만으로도 윈도우즈 창을 띄울 수 있다는 것이다. 이 외에도 wxPython, PyQt, PyGTK등의 Tkinter보다 빠른 속도와 미려한 윈도우 화면을 자랑하는 것들도 있다.

## 3. C/C++과의 결합

파이썬은 접착(glue)언어라고도 불리운다. 그 이유는 다른 언어와 함께 잘 어울릴 수 있기 때문이다. C로 만든 프로그램을 파이썬에서 쓸 수 있으며, 파이썬으로 만든 프로그램을 C에서 역시 사용할 수 있다.

## 4. 웹 프로그래밍

우리는 익스플로러나 크롬, 파이어폭스와 같은 브라우저를 이용하여 인터넷을 사용한다. 누구나 한번쯤 웹 서핑을 하면서 게시판이나 방명록에 글을 남겨 본 적이 있을 것이다. 그러한 게시판이나 방명록을 바로 웹 프로그램이라고 한다. 파이썬은 웹 프로그램을 작성하기에 매우 적합한 도구이며 실제로 파이썬으로 제작된 웹 사이트는 셀 수 없을 정도로 많다.

## 5. 수치연산 프로그래밍

사실 파이썬은 수치연산 프로그래밍에 적합한 언어는 아니다. 왜냐하면 복잡하고 연산이 많다면 C와 같은 언어로 하는 것이 더 빠르기 때문이다. 하지만 파이썬에서는 Numeric Python이라는 수치 연산 모듈을 제공한다. 이 Numeric Python은 C로 작성되었기 때문에 매우 빠르게 수학연산을 수행한다. 이 모듈을 이용하면 파이썬에서 수치연산을 빠르게 할 수 있다.

## 6. 데이터베이스 프로그래밍

파이썬은 Sybase, Infomix, Oracle, MySQL, Postgresql등의 데이터 베이스에 접근할 수 있게 해주는 도구들을 제공한다. 또한 이런 굽직한 데이터베이스를 직접 이용하는 것 외에도 파이썬에는 재미있는 함수가 하나 있다. 바로 pickle이라는 모듈이다. 이 모듈은 파이썬에서 쓰이는 자료들을 변형없이 그대로 파일에 저장하고 불러오는 일들을 해 준다. 이 곳에서는 pickle을 어떻게 사용하고 활용하는지에 대해서도 알아본다.

## 파이썬으로 할 수 없는 일

파이썬으로 도스나 리눅스 같은 운영체제, 엄청난 횟수의 반복과 연산을 필요로 하는 프로그램 또는 데이터 압축 알고리즘 개발 프로그램등을 만들기는 어렵다. 즉, 대단히 빠른 속도를 요구하거나 하드웨어를 직접 건드려야 하는 프로그램에는 어울리지 않는다.

### 3) 파이썬 설치하기

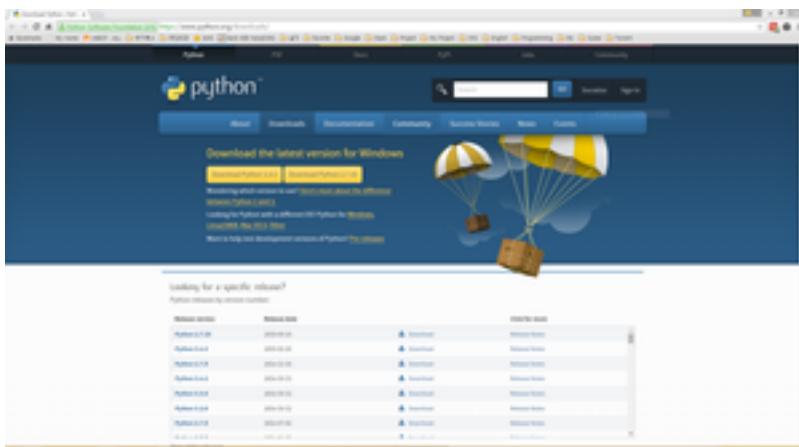
자신의 컴퓨터에 파이썬을 설치해보자. 여기서는 윈도우즈와 리눅스의 경우만을 다루도록 한다. 다른 시스템에서는 파이썬 홈페이지의 설명을 참고하도록 하자.

#### 윈도우에서 파이썬 설치

윈도우즈의 경우에는 설치가 정말 쉽다. 우선 <http://www.python.org/download> (파이썬 공식 홈페이지)에서 윈도우용 파이썬 언어 패키지를 다운로드한다.

다음 그림과 같이 파이썬 버전별로 다운로드 받을 수 있는 목록을 볼 수 있을 것이다. 이 중에서 Python 3.x로 시작하는 버전 중 가장 최근의 윈도우 인스톨러를 다운로드 받도록 하자. (이 글을 작성하는 시점의 최신버전은 3.4.3이다.)

- 64비트 윈도우즈용 : Windows x86-64 MSI installer (python-3.4.3.amd64.msi)
- 32비트 윈도우즈용 : Windows x86 MSI installer (python-3.4.3.msi)

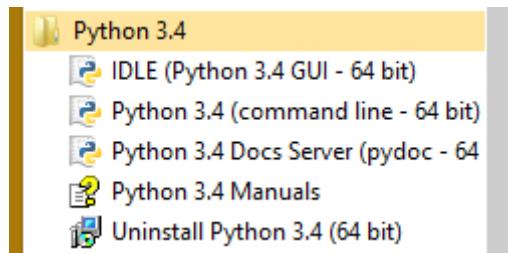


인스톨러를 다운로드 받은 후에 더블클릭하면 바로 설치가 시작된다.

1. Install for All Users 를 선택하고 “Next”를 클릭한다.
2. 다음은 파이썬의 설치디렉토리를 묻는 화면이다. 디폴트 값(아마도 C:\Python34\ )을 그대로 두고 “Next”를 클릭한다.
3. Customize Python 부분에서는 “Next”를 클릭한다.
4. 파이썬 설치가 진행된다. 설치가 완료되면 “Finish”를 클릭하여 종료하도록 하자.

파이썬이 정상적으로 설치되었다면 다음과 같이 프로그램 메뉴에서 확인 할 수 있을 것이다.

[시작 -> 모든프로그램 -> Python 3.4]



※ 만약 파이썬 2.7 버전을 설치할 경우에는 Python 2.7용 인스톨러 파일을 받아서 설치하면 된다.

## 리눅스에서 파이썬 설치

리눅스 사용자라면 이미 디폴트로 파이썬이 설치되어 있을 것이다.

```
$ python -V
```

위의 명령어를 치면 파이썬 버전을 확인할 수 있다.

<http://www.python.org/download> 에 접속해 Python-3.4.3.tgz를 다운로드한다.

여기서는 소스를 컴파일하여 설치하는 방법에 대해서 알아보도록 한다.

우선 다음처럼 압축을 푼다:

```
$ tar xvzf Python-3.4.3.tgz
```

다음에 해당 디렉토리로 이동한다:

```
$ cd Python-3.4.3
```

Makefile 파일을 만들기 위해서 configure를 실행한다:

```
$ ./configure
```

파이썬 소스를 컴파일 한다:

```
$ make
```

루트 계정으로 인스톨한다:

```
$ su -
$ make install
```

※ 파이썬 2.7 버전을 설치할 경우 Python-3.4.3.tgz 가 아닌 Python-2.7.tgz 파일을 다운로드 받아서 동일한 방법으로 설치하면 된다.

## 4) 파이썬 둘러보기

도대체 파이썬이란 언어는 어떻게 생겼는지, 어떤식으로 코드를 작성하는지 간단히 알아보자. 파이썬이란 언어를 자세히 탐구하기 전에 전체적인 모습을 한번 훑어보는 것은 매우 유익한 일이 될 것이다.

“백문이 불여 일견, 백견이 불여 일타”라 했다. 따라해 보자.

### 따라해 보기

먼저 파이썬이라는 것이 도대체 어떻게 생긴 것인지 보도록 하자. [시작 -> 모든프로그램 -> Python 3.4.3 -> Python (Command Line)]을 선택하자.

다음과 같은 화면을 볼 수 있을 것이다.

```
Python 3.4.3 (v3.4.3:9b73f1c3e601, Feb 24 2015, 22:44:40) [MSC v.1600 64 b...
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

위와 같은 것을 대화형 인터프리터라고 하는데 앞으로 이 책에서는 이 인터프리터로 파이썬 프로그래밍의 기초적인 사항들에 대해서 설명할 것이다. (※ 대화형 인터프리터는 파이썬 쉘(Python shell)이라고도 말한다.) 세 개의 꺾은 괄호(>>>)는 프롬프트(prompt)라고 부른다.

대화형 인터프리터를 종료하는 방법은 Ctrl-Z 키를 입력하는 것이다(유닉스 계열에서는 Ctrl + D).

또는 다음의 예와 같이 sys 모듈을 사용하여 종료하는 방법이 있다.

```
>>> import sys
>>> sys.exit()
```

앞으로 보게 될 내용들은 나중에 다시 자세하게 다를 것이니 이해가 되지 않는다고 절망하거나 너무 오래동안 고심하지 말도록 하자.

다음과 같이 순서대로 따라 해 보자.

1 더하기 2는 3이라는 값을 출력한다:

```
>>> 1 + 2  
3
```

나눗셈과 곱셈 역시 예상한대로의 결과값을 보여준다:

```
>>> 3 / 2.4  
1.25  
>>> 3 * 9  
27  
>>>
```

a에 1을 b에 2를 대입한 다음 a와 b를 더하면 3이란 결과값을 보여준다:

```
>>> a = 1  
>>> b = 2  
>>> a + b  
3
```

a라는 변수에 “Python”이라는 값을 대입한 다음 print(a)를 해 주면 a의 값을 출력해 준다:

```
>>> a = "Python"
```

---

```
>>> print(a)
Python
```

파이썬은 복소수도 지원한다:

```
>>> a = 2 + 3j
>>> b = 3
>>> a * b
(6+9j)
```

a에  $2+3j$ 라는 값을 설정하였다. 여기서  $2+3j$ 란 복소수를 의미하는 것이다. 보통 우리는 고등학교때 복소수를 표시할 때 알파벳 i를 이용해서  $2 + 3i$ 처럼 사용했지만 파이썬에서는 j를 사용한다. 위의 예는  $2 + 3j$  와 3을 곱하는 방법이다. 당연히 결과 값으로  $6+9j$ 가 될 것이다.

다음은 간단한 조건문 if를 이용한 예제이다:

```
>>> a = 3
>>> if a > 1:
...     print("a is greater than 1")
...
a is greater than 1
```

print문 앞의 ...은 아직 문장이 끝나지 않았음을 알려주는 프롬프트이다. 위 예제에서 두번째 ... 이후에 “엔터”키를 입력하면 if 문이 종료되고 “a is greater than 1”이라는 문장이 출력된다.

if a > 1: 다음 문장은 탭키 또는 스페이스키 4개를 이용하여 들여쓰기를 한 후에 print("a is grater than 1")이라고 입력해야 한다. 들여쓰기 규칙에

대해서는 “제어문”에서 자세하게 알아볼 것이다. 뒤의 “for”, “while” 예제도 마찬가지로 들여쓰기가 필요하다.

다음은 **for**를 이용해서 [1, 2, 3]안의 값들을 하나씩 출력해 주는 것을 보여준다:

```
>>> for a in [1, 2, 3]:  
...     print(a)  
...  
1  
2  
3
```

위의 예는 대괄호([]) 사이에 있는 값들을 하나씩 출력해 준다. 위 코드의 의미는 “[1, 2, 3]이라는 리스트에서 앞에서부터 하나씩 꺼내어 a라는 변수에 대입한 후 print(a)를 수행하라”이다. 당연히 a에 차례로 1, 2, 3이란 값이 대입될 것이고 print(a)에 의해서 그 값이 차례대로 출력 될 것이다.

다음은 **while**을 이용하는 모습이다:

```
>>> i = 0  
>>> while i < 3:  
...     i=i+1  
...     print(i)  
...  
1  
2  
3
```

while 이란 영어단어는 “~인 동안”이란 뜻이다. 위의 예제는 “i값이 3보다 작다”인 동안 “i = i + 1”과 “print(i)”를 수행하라는 말이다. i = i + 1이란

---

문장은 i 의 값을 1씩 더하게 한다. i값이 3보다 커지게 되면 while문을 빠져 나가게 된다.

파이썬의 **함수**는 다음처럼 생겼다:

```
>>> def sum(a, b):
...     return a+b
...
>>> print(sum(3,4))
7
```

파이썬에서 **def**는 함수를 만들 때 사용되는 예약어이다. 위의 예는 sum이란 함수를 만들고 그 함수를 어떻게 사용하는지에 대한 예를 보여준다. sum(a, b)에서 a, b는 입력값이고 a+b 는 리턴값이다. 즉 3, 4가 입력으로 들어오면 3+4를 수행하고 그 결과값인 7을 돌려준다.

이상과 같이 기초적인 파이썬 문법들에 대해서 간략하게 알아보았다.

## 에디터로 파이썬 프로그램 작성하기

이미 눈치를 챘을지도 모르지만 독자들이 유용한 파이썬 프로그램을 작성하기 위해서는 인터프리터보다는 에디터를 이용하여 파이썬 프로그램을 작성하는 것이 좋다. 에디터라는 것은 문서를 편집할 수 있는 프로그래밍 툴을 말한다. 독자가 즐겨 사용하는 에디터가 없다면 필자는 “에디트 플러스”라는 프로그램을 적극 추천한다. 에디트 플러스 프로그램의 설치법과 사용법에 대한 사항이 부록에 자세하게 나와 있으니 참고하기 바란다.

다음과 같은 프로그램을 에디터로 직접 작성해 보자:

```
# hello.py
```

```
print("Hello world")
```

이러한 내용을 담은 파일을 “hello.py”라는 이름으로 저장하도록 하자. 위의 파일에서 “# hello.py”라는 문장은 주석이다. ‘#’로 시작하는 문장은 '#'부터 시작해서 그 줄 끝까지 프로그램 실행에 전혀 영향을 끼치지 않는다. 주석은 프로그래머를 위한 것으로 프로그램 소스에 설명문을 달 때 사용하게 된다.

에디터로 파이썬 프로그램을 작성한 후 저장할 때 파일 이름의 확장자명을 항상 py로 해주도록 하자. py는 파이썬 파일임을 알려주는 관례적인 확장자명이다. 이제 이 hello.py라는 프로그램을 실행시키기 위해서 도스창을 열고([윈도 우키+R -> cmd 입력 후 엔터]) hello.py라는 파일이 저장된 곳으로 이동한 후 다음과 같이 입력하자. (여기서는 hello.py라는 파일이 C:\Python이라는 디렉토리에 있다고 가정을 한다.)

```
C:\Users\home>cd c:/Python
```

```
c:\Python>c:\python34\python hello.py  
Hello world
```

```
c:\Python>
```

위와 같은 결과값을 볼 수 있을 것이다. 위와 같은 결과 값을 볼 수 없는 독자라면 파이썬이 c:\python34라는 디렉토리에 설치되어 있는지, hello.py라는 파일이 이동한 디렉토리에 존재하는지에 대해서 다시 한번 살펴 보도록 하자. (참고 - 만약 에디트 플러스라는 에디터를 사용한다면 보다 쉽게 에디터로 작성한 파이썬 프로그램을 실행할 수가 있다.)

위에서는 “Hello world”라는 문장을 출력하는 단순한 프로그램을 에디터로 작성했지만 보통 에디터로 작성하는 프로그램은 꽤 여러 줄로 이루어 질 것이다.

에디터로 만든 프로그램 파일은 언제나 다시 사용할 수 있다. 대화형 인터프리터에서 만든 프로그램은 인터프리터를 종료함과 동시에 사라지게 되지만 에디터로 만든 프로그램은 파일로 존재하기 때문에 언제고 다시 사용할 수 있다.

왜 대부분 에디터를 이용해서 파이썬 프로그램을 작성해야하는지 이제 이해가 될 것이다.

## 4. 자료형

자료형이란 프로그래밍을 할 때 쓰이는 숫자, 문자열등의 자료 형태로 사용하는 그 모든 것을 뜻한다. 프로그램의 가장 기본이 되고 핵심적인 단위가 되는 것이 바로 자료형이다. 따라서 자료형을 충분히 이해하지 않고 프로그래밍을 시작 하려는 것은 기초 공사가 마무리되지 않은 상태에서 빌딩을 세우는 것과 같다. 프로그래밍 언어를 배울 때 “그 언어의 자료형을 알게 된다면 이미 그 언어의 절반을 터득한 것이나 다름없다”라는 말이 있다. 자료형은 가장 기초가 되는 중요한 부분이니 주의를 기울여 자세히 살펴보도록 하자.

우리가 이곳에서 기본적으로 알아야 할 자료형에는 숫자, 문자열, 리스트, 튜플, 딕셔너리 등이 있다. 이것들에 대해서 하나씩 자세하게 알아 보도록 하자.

## 1) 숫자형 (Number)

숫자형이란 숫자 형태로 이루어진 자료형으로 우리가 이미 알고 있는 것들이다. 우리가 흔히 사용해 왔던 것들을 생각해 보자. 123과 같은 정수, 12.34 같은 실수, 공대생이라면 필수적으로 알아야 할 복소수( $1 + 2j$ ) 같은 것들도 있고, 드물게 쓰이긴 하지만 8진수나 16진수 같은 것들도 있다.

이런 숫자들을 파이썬에서는 어떻게 사용하는지 알아보자.

항목	사용 예
정수	123, -345, 0
실수	123.45, -1234.5, 3.4e10
복소수	$1 + 2j$ , $-3j$
8진수	0o34, 0o25
16진수	0x2A, 0xFF

위의 표는 숫자들이 파이썬에서 어떻게 사용되는지를 간략하게 보여준다.

### 정수형(Integer)

정수형이란 말 그대로 정수를 뜻하는 숫자를 말한다. 아래의 a는 각각의 숫자를 대입한 변수이다.

다음 예는 양의 정수와 음의 정수, 숫자 0을 변수 a에 대입하는 예제이다.

```
>>> a = 123
>>> a = -178
>>> a = 0
```

## 소수점 포함된 것(Floating-point)

다음의 예제를 보자.

```
>>> a = 1.2  
>>> a = -3.45
```

위의 방식은 우리가 늘 사용하던 소수점 표현방식이다.

```
>>> a = 4.24E10  
>>> a = 4.24e-10
```

위의 방식은 컴퓨터식 지수 표현방법으로 파이썬에서는 4.24e10 또는 4.24E10처럼 표현한다. (e, E둘중 어느 것을 사용해도 무방하다.) 여기서 4.24E10은  $4.24 * 10^{10}$ , 4.24e-10은  $4.24 * 10^{-10}$  을 의미한다.

## 8진수(Octal)

8진수를 만들기 위해서는 숫자가 0o 또는 0O(숫자 0 + 알파벳 o 또는 대문자 O)으로 시작하면 된다.

```
>>> a = 0o177
```

## 16진수(Hexadecimal)

16진수를 만들기 위해서는 숫자가 0x로 시작하면 된다.

```
>>> a = 0x8ff
>>> b = 0xABCD
```

8진수나 16진수는 잘 사용하지 않는 형태의 숫자 자료형이다.

## 복소수 (Complex number)

보통 우리는 중고등학교 시절에 'j' 대신 'i'를 사용했을 것이다. 파이썬은 'i' 대신 'j'를 사용한다. 'j'를 써도 되고 'J'를 써도 된다.

```
>>> a = 1+2j
>>> b = 3-4J
```

복소수를 활용하는 예들을 몇가지 보도록 하자. 복소수에는 복소수가 자체적으로 가지고 있는 내장함수가 있다. 그것들을 이용하면 좀더 다양한 방법으로 복소수를 사용할 수 있게 된다.

다음의 복소수 예들을 보자.

복소수.real은 복소수의 실수 부분을 돌려준다.

```
>>> a = 1+2j
>>> a.real
1.0
```

복소수.imag는 복소수의 허수 부분을 돌려준다.

```
>>> a = 1+2j  
>>> a.imag  
2.0
```

복소수.conjugate()는 복소수의 켤레 복소수를 돌려준다.

```
>>> a = 1+2j  
>>> a.conjugate()  
(1-2j)
```

abs(복소수)는 복소수의 절대값을 돌려준다. ( $1+2j$ 의 절대값은  $\sqrt{1^2 + 2^2}$  이다.)

```
>>> a = 1+2j  
>>> abs(a)  
2.2360679774997898
```

## 숫자 연산

프로그래밍을 한번도 해 본적이 없는 독자라도 사칙연산(+, -, \*, /)은 알고 있을 것이다. 파이썬 역시 계산기와 마찬가지로 아래의 연산자를 이용하여 사칙연산을 수행한다.

```
>>> a = 3  
>>> b = 4  
>>> a + b  
7  
>>> a * b
```

---

12

>>> a / b

0.75

※ 파이썬 2.7의 경우 가장 마지막 예제인 a / b를 실행하면 0이 리턴된다. 동일한 결과를 얻고 싶다면 a / (b\*1.0)으로 b를 강제로 실수형으로 변환한 후 수행해야만 한다. 파이썬 2.7의 경우 정수형 간 나눗셈의 경우 정수로만 그 결과를 리턴하기 때문이다. 파이썬 3는 위의 예제에서 보듯이 따로 형 변환이 필요없다. (참고: [파이썬 2.7 vs 파이썬 3](#))

다음에 알아야 될 연산자로 \*\* 라는 연산자가 있다. 이것은 x \*\* y처럼 사용되었을 때 x의 y승 값을 돌려 준다. 다음의 예를 통해 알아보자.

>>> a = 3

>>> b = 4

>>> a \*\* b

81

프로그래밍을 접해 본 적이 없는 독자라면 %연산자는 본 적이 없을 것이다. %는 나머지 값을 반환하는 연산자이다. 7을 3으로 나누면 나머지는 1이 될 것이고 3을 7로 나누면 나머지는 3이 될 것이다. 다음의 예로 확인해 보자.

>>> 7 % 3

1

>>> 3 % 7

3

이번에는 소수점 자리를 버리는 // 연산자에 대해서 알아보자. 다음의 예를 보며 확인 해 보자.

```
>>> 7 / 4  
1.75
```

7 나누기 4의 값은 예상대로 1.75가 된다.

이번에는 소수점 자리를 버리는 // 연산자를 사용한 경우를 보자.

```
>>> 7 // 4  
1
```

1.75의 소수점 부분인 0.75가 제거되어 1이 나오는 것을 확인할 수 있다.

## 2) 문자열 (String)

문자열이란 문장을 뜻한다. 예를 들어 다음과 같은 것들이 문자열이다.

```
"Life is too short, You need Python"  
"a"  
"123"
```

위의 예를 보면 이중 인용부호("")로 둘러싸인 것은 모두 문자열이 되는 것을 알 수 있다. “123”은 숫자인데 왜 문자열인가?라는 의문이 드는 독자는 인용부호("")로 둘러싸인 것은 모두 문자열이라고 생각하면 될 것이다.

[저자주] 만약 PDF문서로 이 문서를 보고 계신다면 단일인용부호, 이중인용부호 기호가 조금 이상하게 표시되어 있을 수 있습니다. PDF는 여는 따옴표와 닫는 따옴표를 구분하기 때문에 변환과정이 매끄럽지 못한 점 양해 부탁드립니다. 여는 따옴표, 닫는 따옴표 구분 없이 모두 같은 따옴표라고 생각하시고 읽어 주시면 감사하겠습니다.

### 문자열 만드는 방법 4가지

위의 예에서는 문자열을 만들 때 이중인용부호("")만을 사용했지만 이 외에도 문자열을 만드는 방법은 세 가지가 더 있다. 파이썬에서 문자열을 만드는 방법은 다음과 같이 네 가지로 구분된다.

```
"Hello World"  
'Python is fun'  
"""Life is too short, You need python"""  
'''Life is too shor, You need python'''
```

문자열을 만들기 위해서는 위의 예에서 보듯이 이중 인용부호("")로 문자열의 양 쪽을 둘러싸거나 단일 인용부호(') 또는 이중 인용부호나 단일 인용부호 세 개를 연속으로 쓰는(""", ''')를 양쪽으로 둘러싸면 된다. 그렇다면 왜 단순함을 좋아하는 파이썬에서 이렇듯 다양한 문자열 만드는 방법을 가지게 되었을까? 그 이유에 대해서 알아보도록 하자.

### 문자열 내에 ('') 또는 (") 을 포함시키고 싶을 때

문자열을 만들어주는 주인공은 ('') 와 (")이다. 하지만 문자열 내에 ('') 와 (") 를 포함시켜야 할 경우가 있다. 이 때는 좀 더 특별한 기술이 필요하다. 예제를 하나씩 살펴보면서 원리를 익혀보도록 하자.

#### 예 1) 단일 인용부호(')를 포함시키고 싶을 때

```
Python's favorite food is perl
```

예 1과 같은 문자열을 변수에 저장하고 싶다고 가정해보자. Python's에서 보듯이 단일인용부호(')가 포함되어 있다. 이럴 때는 다음과 같이 문자열을 이중 인용부호("")로 둘러싸야 한다. 이중 인용부호("")안에 들어 있는 단일 인용부호(')는 문자열을 나타내기 위한 기호로 인식되지 않는다.

```
>>> food = "Python's favorite food is perl"
```

시험삼아 다음과 같이 (")이 아닌 ('')로 문자열을 둘러싼 뒤 실행시켜 보자. 'Python' 이 문자열로 인식되어 에러(Syntax Error)가 날 것이다.

```
>>> food = 'Python's favorite food is perl'
```

#### 예 2) 이중 인용부호("")를 포함시키고 싶을 때

---

```
"Python is very easy." he says.
```

예 2와 같이 이중 인용부호(")가 포함된 문자열이라면 어떻게 해야 (" ) 이 제대로 표현될까?

다음과 같이 그 문자열을 단일인용부호(')로 둘러싸면 된다.

```
>>> say = '"Python is very easy." he says.'
```

이렇게 단일인용 부호(')안에 사용된 이중인용부호("")는 문자열을 나타내는 기호로 인식되지 않는다.

예 3) \ (역슬래시)로 (' )과 (" )를 문자열에 포함시키기

```
>>> food = 'Python\'s favorite food is perl'
>>> say = "\"Python is very easy.\\" he says."
```

(' )나 (" ")를 문자열에 포함시킬 수 있는 또 다른 방법은 '\' (역슬래시)를 이용하는 것이다. 즉 (\') 가 문자열 내에 삽입되면 그것은 문자열을 둘러싸는 기호의 의미가 아니라 문자 (') 그 자체를 뜻하게 된다. (\") 역시 마찬가지이다. 어떤 것을 사용할 것인지는 각자의 선택이다. 대화형 인터프리터를 실행시킨 뒤 위의 예를 꼭 직접 실행해 보도록 하자.

### 여러 줄 짜리 문자열 처리

문자열이 항상 한 줄 짜리만 있는 것은 아니다. 다음과 같이 여러 줄 짜리 문자열이 있을 때는 어떻게 처리해야 할까?

```
Life is too short
You need python
```

이러한 문자열을 변수에 대입하려면 어떻게 하겠는가?

예1) 줄바꿈 문자인 '\n' 삽입

```
>>> multiline = "Life is too short\nYou need python"
```

위의 예처럼 줄바꿈 문자인 '\n'을 삽입하는 방법이 있지만 읽기에 너무 불편하고 너무 줄이 길어지는 단점이 있다. 이것을 극복하기 위해 파이썬에서는 다음과 같이 (""""")를 이용한다.

예 2) 연속된 이중인용부호 세 개(""""") 이용

```
multiline="""  
Life is too short  
You need python  
"""
```

위 예에서도 확인할 수 있듯이 문자열이 여러줄일 경우 위와같은 방식이 상당히 유리하고 깔끔하다는 것을 알 수 있을 것이다.

#### /참고/ 이스케이프 코드

여러 줄 짜리 문장을 처리할 때 '\n'과 같은 역슬래시 문자를 이용한 이스케이프 코드를 사용했다. 이와 같은 문자를 이스케이프 코드라고 부르는데, 출력물을 보기 좋게 정렬하거나 그 외 특별한 용도로 자주이용된다. 몇 가지 이스케이프 코드를 정리하면 다음과 같다.

---

코드	설명
\n	개행 (줄바꿈)

---

코드	설명
\v	수직 텁
\t	수평 텁
\r	캐리지 리턴
\f	폼 피드
\a	벨 소리
\b	백 스페이스
\000	널문자
\\"	문자 “\”
\'	단일 인용부호(‘)
\"	이중 인용부호(“”)

---

이중에서 활용빈도가 높은 것은 \n, \t, \\, \', \"이다. 나머지는 대부분의 프로그램에서 잘 쓰이지 않는다.

## 문자열 연산

파이썬에서는 문자열을 더하고 곱할 수 있다. 이것은 다른 언어에서는 쉽게 찾아 볼 수 없는 재미있는 기능이다. 우리의 생각을 그대로 반영해주는 파이썬만의 장점이라고 할 수 있다.

문자열을 더하거나 곱하는 방법에 대해 알아보기로 하자.

예 1) 문자열 합치기(Concatenation)

```
>>> head = "Python"  
>>> tail = " is fun!"  
>>> head + tail  
'Python is fun!'
```

예 1의 세번 째 라인을 보자. 복잡하게 생각하지 말고 눈에 보이는 대로 생각해보자. “Python”이라는 head변수와 ” is fun”이라는 tail변수를 더한 것이다. 결과는 ‘Python is fun!’이다. 즉, head와 tail변수가”+“에 의해 합쳐진 것이다.

직접 실행해보고 결과값이 제시한 것과 똑같이 나오는지 확인해보자.

#### 예 2) 문자열 곱하기

```
>>> a = "python"  
>>> a * 2  
'pythonpython'
```

여기도 마찬가지로 \*의 의미는 숫자 곱하기의 의미와는 다르게 사용되었다. 여기서 사용된 \*는 문자열의 반복을 뜻하는 의미로 사용되었다. 굳이 예를 설명할 필요가 없을 정도로 직관적이다. a \* 2라는 문장은 a를 두번 반복하라는 뜻이다.

문자열 곱하기를 좀 더 응용해보자. 다음과 같은 소스를 에디터로 작성해 실행시켜보자.

```
# multistring.py  
  
print("=" * 50)  
print("My Program")  
print("=" * 50)
```

---

결과값은 다음과 같이 나타날 것이다.

```
=====
My Program
=====
```

위와 같은 표현은 자주 사용하게 된다. 프로그램을 만들고 실행시켰을 때 출력되는 화면 제일 상단에 프로그램 제목으로 위처럼 만들면 보기 좋지 않겠는가?

## 인덱싱과 슬라이싱

인덱싱(indexing)이란 무엇인가를 ‘가리킨다’는 의미이고, 슬라이싱(slicing)은 무엇인가를 ‘잘라낸다’라는 의미이다. 이것들을 생각하면서 다음의 예를 따라해 보도록 하자.

```
>>> a = "Life is too short, You need Python"
```

각 문자열의 문자마다 번호를 매겨 보았다.

```
Life is too short, You need Python
0         1         2         3
0123456789012345678901234567890123
```

즉 “Life is too short, You need Python”이라는 문자열에서 ‘L’은 첫 번째 자리를 뜻하는 숫자인 0을 바로 다음인 ‘i’는 1을 이런식으로 계속 번호를 붙인 것이다. 중간쯤에 있는 “short”의 s는 12라는 번호가 된다.

그리고 다음 예를 실행해 보자.

```
>>> a = "Life is too short, You need Python"  
>>> a[3]  
'e'
```

a[3] 이 뜻하는 것은 a라는 문자열의 네 번째 문자인 'e'를 말한다. 프로그래밍을 처음 접하는 독자라면 a[3]에서 3이란 숫자는 세 번째인데 왜 네 번째 문자를 말한다는 것인지 의아할 것이다.

이 부분이 사실 필자도 가끔 많이 헷갈리는 부분인데, 이렇게 생각하면 쉽게 알 수 있을 것이다.

“파이썬은 0부터 숫자를 센다”

위의 문자열을 파이썬은 이렇게 바라보고 있는 것이다.

```
a[0]: 'L', a[1]: 'i', a[2]: 'f', a[3]: 'e', a[4]: ' ', ...,
```

0부터 숫자를 센다는 것이 처음에는 익숙하지 않겠지만 이것도 하다 보면 자연스럽게 될 것이다. 위의 예에서 보듯이 a[3]이란 것은 문자열 내 특정한 값을 뽑아내는 역할을 해준다. 이러한 것을 인덱싱(Indexing)이라고 부른다.

몇가지를 인덱싱을 더 해 보도록 하자.

```
>>> a[0]  
'L'  
>>> a[12]  
's'  
>>> a[-1]  
'n'
```

마지막의 `a[-1]`이 뜻하는 것은 뭘까? 눈치 빠른 독자는 이미 알겠지만 바로 문자열을 뒤에서부터 읽기 위해서 마이너스(-) 기호를 붙이는 것이다. 즉 `a[-1]`은 뒤에서부터 세어서 첫 번째가 되는 문자를 말한다. `a`는 “Life is too short, You need Python”이라는 문장이므로 뒤에서부터 첫 번째 문자는 가장 마지막 문자인 ’n’이 될 것이다.

뒤에서부터 첫 번째 문자를 표시 할 때 `a[-0]`이라고 해야 하지 않겠는가?라는 의문이 들수도 있겠지만 잘 생각해 보자. 0과 -0은 똑같은 것이기 때문에 `a[-0]`이라는 것은 `a[0]`과 똑같은 값을 보여준다.

```
>>> a[-0]
'L'
```

계속해서 몇가지 예를 더 보자.

```
>>> a[-2]
'o'
>>> a[-5]
'y'
```

위의 첫 번째 예는 뒤에서부터 두 번째 문자를 가리키는 것이고 두 번째 예는 뒤에서부터 다섯 번째 문자를 가리키는 것이다. 그렇다면 “Life is too short, You need Python”이라는 문자열에서 단순히 한 문자만을 뽑아내는 것이 아니라 ’Life’ 또는 ’You’ 같은 단어들을 뽑아낼 수 있는 방법은 없을까?

다음과 같이 하면 될 것이다.

```
>>> b = a[0] + a[1] + a[2] + a[3]
>>> b
'Life'
```

위의 방법처럼 할 수도 있겠지만 파이썬에서는 보다 더 좋은 방법을 제공한다. 바로 슬라이싱(Slicing)이라는 기법이다.

위의 예는 슬라이싱 기법으로 다음과 같이 간단하게 처리할 수 있다. (주의 - 지금까지의 예제는 계속 이어지는 것이기 때문에 이미 인터프리터를 닫고 다시 시작하는 독자라면 >>> a = "Life is too short, You need Python"이라는 것을 먼저 수행한 뒤 다음의 예제들을 따라하도록 하자)

```
>>> a[0:4]  
'Life'
```

a[0:4]가 뜻하는 것은 a라는 문자열 즉, "Life is too short, You need Python"이라는 문장에서 0부터 4까지의 문자를 뽑아낸다는 뜻이다. 하지만 다음과 같은 의문이 들 것이다.

a[0]은 'L', a[1]은 'i', a[2]은 'f', a[3]은 'e'이니까 a[0:3]만으로도 'Life'라는 단어를 뽑아낼 수 있지 않을까?

다음의 예를 보도록 하자.

```
>>> a[0:3]  
'Lif'
```

이렇게 되는 이유는 간단하다. a[시작번호: 끝번호] 처럼 쓰면 끝번호에 해당하는 것은 포함이 되지 않는다.

a[0:3]을 수식으로 나타내면 다음과 같다.

```
0 <= a < 3
```

---

즉 위의 수식을 만족하는 a는 a[0], a[1], a[2] 일 것이다. 따라서 a[0:3]은 'Lif'이고 a[0:4]는 'Life'가 되는 것이다. 이 부분이 문자열 연산에서 가장 혼동하기 쉬운 부분이니 스스로 많이 연습해 보기를 바란다.

슬라이싱의 예를 조금 더 보도록 하자.

```
>>> a[0:5]  
'Life '
```

위의 예는 a[0] + a[1] + a[2] + a[3] + a[4]와 동일하다. a[4]라는 것은 공백문자 ‘이기 때문에 ’Life’가 아닌 ’Life ’가 되는 것이다. 공백문자 역시 ’L’, ’i’ , ’f’, ’e’와 동일하게 취급되는 것을 잊지 말도록 하자. ’Life’와 ’Life ’는 완전히 다른 문자열이다.

항상 시작번호가 ’0’일 필요는 없다.

```
>>> a[0:2]  
'Li'  
>>> a[5:7]  
'is'  
>>> a[12:17]  
'short'
```

a[시작번호:끝번호]에서 끝번호 부분을 생략하면 시작번호부터 그 문자열의 끝까지를 뽑아내게 된다.

```
>>> a[19:]  
'You need Python'
```

a[시작번호:끝번호]에서 시작번호를 생략하면 그 문자열의 처음부터 끝번호까지 뽑아내게 된다.

```
>>> a[:17]
'Life is too short'
```

a[시작번호:끝번호]에서 시작번호와 끝번호를 생략하면 처음부터 끝까지 뽑아낸다.

```
>>> a[:]
'Life is too short, You need Python'
```

a[시작번호:끝번호]에서 시작번호와 끝번호를 모두 생략했기 때문에 이것은 처음부터 끝까지를 말하게 되므로 위와 같은 결과를 보여주는 것이다.

슬라이싱에서 역시 인덱싱과 마찬가지로 '-'기호를 사용할 수가 있다.

```
>>> a[19:-7]
'You need'
```

a[19:-7]이 뜻하는 것은 a[19]에서부터 a[-7]까지를 말한다. 이것 역시 a[-7]은 포함하지 않는다.

### 자주 사용되는 슬라이싱 예

다음은 자주 사용하게 되는 슬라이싱 기법 중의 하나이다.

```
>>> a = "20010331Rainy"
>>> date = a[:8]
>>> weather = a[8:]
```

```
>>> date  
'20010331'  
>>> weather  
'Rainy'
```

a라는 문자열을 두 부분으로 나누는 기법이다. 동일한 숫자 '8'을 기준으로 a[:8], a[8:]처럼 사용을 하였다. a[:8]은 a[8]이 포함이 안되고 a[8:]은 a[8:]을 포함하기 때문에 8을 기준으로 해서 두 부분으로 나눌 수 있는 것이다. 위의 예에서는 “20010331Rainy”라는 문자열을 날짜를 나타내는 부분인 ‘20010331’과 날씨를 나타내는 부분인 ‘Rainy’로 나누는 방법을 보여준다.

위의 문자열 “20010331Rainy”라는 것을 연도인 2001과 월과 일을 나타내는 0331 그리고 날씨를 나타내는 Rainy를 세 부분으로 나누려면 다음과 같이 할 수 있다.

```
>>> a = "20010331Rainy"  
>>> year = a[:4]  
>>> day = a[4:8]  
>>> weather = a[8:]  
>>> year  
'2001'  
>>> day  
'0331'  
>>> weather  
'Rainy'
```

위의 예는 4와 8이란 숫자로 “20010331Rainy”라는 문자열을 세 부분으로 나누는 방법을 보여준다.

이상과 같이 인덱싱과 슬라이싱에 대해서 살펴 보았다. 인덱싱과 슬라이싱은 프로그래밍을 할 때 매우 자주 사용되는 기법이니 꼭 반복해서 연습을 해 두도록 하자.

### “Pithon”이란 문자열을 “Python”으로 바꾸려면?

위의 제목과 같이 “Pithon”이란 문자열을 “Python”으로 바꾸려면 어떻게 해야 할까? 제일 먼저 떠오르는 생각은 다음과 같을 것이다.

```
>>> a = "Pithon"  
>>> a[1]  
'i'  
>>> a[1] = 'y'
```

위의 예에서 보듯이 우선 a라는 변수에 “Pithon”이란 문자열을 대입하고 a[1]이란 값이 ‘i’니까 a[1]을 위의 예처럼 ‘y’로 바꾸어 준다는 생각이다. 하지만 결과는 어떻게 나올까?

당연히 에러가 나고 실패하게 될 것이다. 에러가 나는 이유는 문자열의 요소 값은 바꿀 수 있는 값이 아니기 때문이다. (문자열, 튜플등의 immutable한 자료형은 그 요소값을 변경할 수 없다.) 하지만 앞서 살펴보았던 슬라이싱 기법을 이용해서 “Pithon”이란 문자열을 “Python”으로 바꿀 수 있는 방법이 있다.

다음의 예를 보자.

```
>>> a = "Pithon"  
>>> a[:1]  
'P'  
>>> a[2:]  
'thon'
```

---

```
>>> a[:1] + 'y' + a[2:]
'Python'
```

위의 예에서 보듯이 슬라이싱을 이용해서 먼저 'Pithon'이라는 문자열을 'P'부분과 'thon'부분으로 나눌 수 있기 때문에 그 사이에 'y'라는 문자를 추가하여 'Python'이라는 새로운 문자열을 만들면 된다.

## 문자열 포매팅(Formatting)

문자열에서 알아야 할것으로는 또하나, 문자열 포매팅이라는 것이 있다. 이것을 공부하기에 앞서 다음과 같은 문자열을 출력하는 프로그램을 작성했다고 가정해보자.

“현재 온도는 18도입니다.”

하지만 시간이 지나서 20도가 된다면 아래와 같은 문장을 출력하는 프로그램.

“현재 온도는 20도입니다”

이러한 프로그램을 어떻게 만들 수 있는지에 대해서는 생각하지 말고 출력해주는 문자열에만 주목해 보자. 출력된 문자열은 모두 같은데 20이라는 숫자와 18이라는 숫자만이 다르다. 이렇게 문자열 내의 어떤 특정한 값을 변화시키는 것이 필요한 경우가 생기는데 이것을 가능하게 해 주는 것이 바로 문자열 포매팅 기법이다.

문자열 포매팅이란 문자열 내에 어떤 값을 삽입하는 방법이다. 다음의 예들을 따라해 보면서 그 사용법을 알아보자.

예 1) 숫자 바로 대입

```
>>> "I eat %d apples." % 3
```

```
'I eat 3 apples.'
```

위의 예제 결과 값을 보면 알겠지만 위의 예는 문자열 내에 3이라는 정수값을 삽입하는 방법을 보여준다. 삽입할 3이라는 숫자는 가장 뒤에 “%” 문자 다음에 쓰였다. 그리고 문자열 내에 3이라는 값을 넣고 싶은 자리에 “%d”라는 문자를 넣어주었다. 하지만 꼭 문자열 내에 숫자만 넣으라는 법은 없다. 이번에는 숫자 대신 문자열을 넣어 보자.

예 2) 문자열 바로 대입

```
>>> "I eat %s apples." % "five"  
'I eat five apples.'
```

위와 예에서 보는 것과 같이 문자열 내에 또 다른 문자열을 삽입하기 위해서는 앞서 사용했던 %d 가 아닌 %s를 썼음을 알 수 있다. 그렇다면 위와 같은 사실로 독자는 유추할 수 있을 것이다. 숫자를 삽입하게 위해서는 %d를 써야 하고 문자열을 삽입하기 위해서는 %s를 써야 한다는 사실을.

예 3) 숫자 변수로 대입

```
>>> number = 3  
>>> "I eat %d apples." % number  
'I eat 3 apples.'
```

예 1처럼 숫자를 바로 대입하나 예 3처럼 숫자값을 나타내는 변수를 대입하나 같은 결과가 나온다.

그렇다면 문자열 안에 한 개가 아닌 여러개의 값을 삽입하고 싶다면 어떻게 해야 할까?

---

예 4) 두 개 이상의 값을 치환

```
>>> number = 10
>>> day = "three"
>>> "I ate %d apples. so I was sick for %s days." % (number, day)
'I ate 10 apples. so I was sick for three days.'
```

예 4처럼 두 개 이상의 값을 치환하려면 위에서 보듯이 마지막 % 다음에 ( ) 사이에 콤마로 구분하여 변수를 넣어 주어야만 한다.

### 문자열 포맷 코드

예 4는 대입시키는 자료형으로 정수와 문자열을 사용했지만 이 이외에도 다양한 것들을 대입시킬 수 있다. 문자열 포맷 코드로는 다음과 같은 것들이 있다.

---

코드	설명
%s	문자열 (String)
%c	문자 한개(character)
%d	정수 (Integer)
%f	부동소수 (floating-point)
%o	8진수
%x	16진수
%%	Literal % (문자 % 자체)

---

여기서 재미있는 것은 %s 포맷 코드로 이것은 어떤 형태로든 변환이 가능하다.

무슨 말인지 예를 통해 확인해 보자.

```
>>> "I have %s apples" % 3  
'I have 3 apples'  
>>> "rate is %s" % 3.234  
'rate is 3.234'
```

3을 문자열 내에 삽입하려면 문자열 내에 %d가 있어야 하고 3.234를 삽입하려면 문자열 내에 %f가 있어야 하지만 %s를 사용하면 이런 것을 생각하지 않아도 된다. 왜냐하면 %s는 자동으로 % 뒤의 파라미터 값을 문자열로 치환하기 때문이다.

/참고] 포맷팅 연산자 %d와 %를 같이 쓸 때는 %%를

```
>>> "Error is %d%." % 98
```

당연히 결과값으로 “Error is 98%.”가 출력될 것이라고 예상하겠지만 파이썬은 에러 메시지를 보여준다. 이유는 문자열 포맷코드인 %d 와 % 가 같은 문자열 내에 존재하면 %를 나타내기 위해서는 반드시 %%로 해야 %로 되는 법칙이 있다. 이건 꼭 기억해두어야 한다. 하지만 문자열 내에 %d 같은 포맷팅 연산자가 없으면 %는 홀로 쓰여도 상관이 없다. 따라서 위 예를 제대로 실행될 수 있게 하려면 다음과 같이 해야 한다.

```
>>> "Error is %d%%." % 98  
'Error is 98%.'
```

### 포맷 코드의 또 다른 가능

위에서 보았듯이 %d, %s등의 포맷코드는 문자열 내에 어떤 값을 삽입하기 위해서 사용됨을 알 수 있었다. 하지만 포맷코드를 숫자와 함께 사용하면 더 유용하게 사용할 수 있다. 다음의 예를 보고 따라해 보도록 하자.

---

예 1) 정렬과 공백

```
>>> "%10s" % "hi"
'          hi'
~~~~~
```

위의 “%10s”의 의미는 길이가 10개인 문자열 공간에서 오른쪽으로 정렬하고 그 앞의 나머지는 공백으로 남겨 두라는 의미이다.

그렇다면 반대쪽인 왼쪽 정렬은 “%-10s”가 될 것이다. 확인해 보자.

```
>>> "%-10s{jane." % 'hi'
'hi           jane.'
~~~~~
```

왼쪽으로 정렬하고 나머지는 공백으로 채웠음을 볼 수 있다.

예 2) 소수점 표현

```
>>> "%0.4f" % 3.42134234
'3.4213'
```

3.42134234를 소수점 4번째까지만 나타내고 싶을 경우에 위와 같이 할 수 있다. 즉 여기서 ‘.’의 의미는 소수점 포인트를 말하고 그 뒤의 숫자 4는 뒤에 나올 숫자의 개수를 말한다. ∴ 앞의 숫자는 이전의 예에서와 같이 오른쪽 또는 왼쪽 정렬을 말하는 숫자이다.

```
>>> "%10.4f" % 3.42134234
'      3.4213'
~~~~~
```

위의 예는 3.42134234라는 숫자를 10개의 문자열 공간에 오른쪽으로 정렬하고 소수점은 4번째 자리까지만 표시하게 하는 예를 보여준다. 위의 예와의 차이점은 숫자를 오른쪽으로 정렬했다는 점이다.

지금까지 문자열을 가지고 할 수 있는 일들에 대해서 대부분 알아보았지만 문자열을 가지고 할 수 있는 일들에 대해서 공부해야 할 사항들이 아직 많이 남아 있다. 지겹다면 잠시 책을 접고 휴식을 취하도록 하자.

## 문자열 함수

앞서 복소수에서 복소수가 자체적으로 가지고 있는 복소수 관련함수가 있었던 것처럼 문자열 역시 자체적으로 가지고 있는 관련함수들이 몇 개 있다. 그들을 사용하기 위해서는 문자열 변수이름 뒤에 '.'을 붙인 다음에 관련함수 이름을 써 주면 된다. 이제 문자열이 자체적으로 가지고 있는 유용한 함수들에 대해서 알아보자.

### 고급 문자열 포매팅(format)

문자열의 format 함수를 이용하면 좀 더 발전된 스타일로 문자열 포맷을 지정할 수 있다. 위에서 알아본 문자열 포매팅 예제들은 format을 이용하면 다음과 같이 바꿀 수 있다.

\*\* 숫자 바로 대입 \*\*

```
>>> "I eat {0} apples".format(3)
'I eat 3 apples'
```

“I eat {0} apples” 문자열 중 {0} 부분이 3이라는 숫자로 바뀌었다.

\*\* 문자열 바로 대입 \*\*

```
>>> "I eat {0} apples".format("five")
'I eat five apples'
```

문자열의 {0} 항목이 five라는 문자열로 바뀌었다.

\*\* 숫자 변수로 대입 \*\*

```
>>> number = 3
>>> "I eat {0} apples".format(number)
'I eat 3 apples'
```

문자열의 {0} 항목이 number값으로 바뀌었다.

\*\* 두 개 이상의 값을 치환 \*\*

```
>>> number = 10
>>> day = "three"
>>> "I ate {0} apples. so I was sick for {1} days.".format(number, day)
'I ate 10 apples. so I was sick for three days.'
```

두 개 이상의 값을 치환할 경우 문자열의 {0}, {1}과 같은 인덱스 항목들이 format 함수의 입력값들로 순서에 맞게 치환된다. 즉 위 예에서 {0}은 format 함수의 첫번째 입력값인 number로 바뀌었고 {1}은 format 함수의 두번째 입력값인 day로 바뀌었다.

\*\* 이름으로 치환하기 \*\*

```
>>> "I ate {number} apples. so I was sick for {day} days.".format(number=10,  
'I ate 10 apples. so I was sick for 3 days.'
```

위 예에서 보듯이 {0}, {1} 과 같은 인덱스 항목 대신 {name} 형태를 이용하는 보다 편리한 방법도 있다. {name} 형태를 사용할 경우에는 format 함수의 입력값에는 반드시 name=value와 같은 형태의 입력값이 있어야만 한다.

위 예는 문자열의 {number}, {day}가 format 함수의 입력값인 number=10, day=3 값으로 각각 치환되는 것을 보여주고 있다.

\*\* 인덱스와 이름을 혼용해서 치환하기 \*\*

```
>>> "I ate {0} apples. so I was sick for {day} days.".format(10, day=3)  
'I ate 10 apples. so I was sick for 3 days.'
```

위와 같이 인덱스와 name=value 두개를 혼용해서 사용하는 것도 가능하다.

\*\* 좌측 정렬 \*\*

```
>>> "{0:<10}".format("hi")  
'hi' ,
```

:<10 표현식을 이용하면 치환되는 문자열을 좌측으로 정렬하고 문자열의 총 자리수를 10으로 맞출 수 있다.

\*\* 우측 정렬 \*\*

```
>>> "{0:>10}".format("hi")  
'          hi'
```

---

우측 정렬은 < 대신 >을 이용하면 된다. 화살표 방향을 생각하면 어느쪽으로 정렬이 되는지 금방 알 수 있을 것이다.

\*\* 가운데 정렬 \*\*

```
>>> "{0:^10}".format("hi")
      hi      ,
```

^기호를 이용하면 가운데 정렬도 가능하다.

\*\* 공백 채우기 \*\*

```
>>> "{0:=-10)".format("hi")
'=====hi====='
>>> "{0:!=<10)".format("hi")
'hi!!!!!!'
```

정렬 시 공백문자 대신에 지정한 문자값으로 채워넣는 것도 가능하다. 채워넣을 문자값은 정렬문자인 <, >, ^ 바로 앞에 삽입해야 한다. 위 예에서 첫번째 예 제는 가운데(^)로 정렬하고 빈공간을 = 문자로 채웠고 두번째 예제는 좌측(<)으로 정렬하고 빈공간을 !문자로 채웠다.

\*\* 소수점 표현 \*\*

```
>>> y = 3.42134234
>>> "{0:0.4f)".format(y)
'3.4213'
```

위 예는 format 함수를 이용하여 소수점을 4자리까지만 표현하는 방법을 보여 준다. 이전에 살펴보았던 0.4f의 표현식이 그대로 사용됨을 알 수 있다.

```
>>> "{0:10.4f}".format(y)
      3.4213'
```

위와같이 자리수를 10으로 맞출 수도 있다. 역시 이전에 살펴보았던 10.4f의 표현식이 그대로 사용됨을 알 수 있다.

\*\* { 또는 }문자 표현하기 \*\*

```
>>> "{{ and }}".format()
'{ and }'
```

format 함수를 이용하여 문자열 포매팅을 할 경우에 { 나 } 문자를 포매팅 문자가 아닌 문자 그대로 사용하고 싶은 경우에는 위 예처럼 {{와 }}처럼 중괄호(brace) 두개를 연속해서 사용해야 한다.

### 소문자를 대문자로 바꾸기(upper)

```
>>> a = "hi"
>>> a.upper()
'HI'
```

upper()함수는 소문자를 대문자로 바꾸어준다. 만약 문자열이 이미 대문자라면 아무런 변화도 일어나지 않을 것이다.

### 문자 갯수 세기(count)

```
>>> a = "hobby"
>>> a.count('b')
```

---

2

문자열 중 문자 'b'의 개수를 반환한다.

### 위치 알려주기1(find)

```
>>> a = "Python is best choice"  
>>> a.find('b')  
10
```

문자열 중 문자 'b'가 처음으로 나온 위치를 반환한다. 만약 찾는 문자나 문자열이 존재하지 않는다면 -1을 반환한다.

### 위치 알려주기2(index)

```
>>> a = "Life is too short"  
>>> a.index('t')  
8
```

문자열 중 문자 't'가 처음으로 나온 위치를 반환한다. 만약 찾는 문자나 문자열이 존재하지 않는다면 에러를 발생시킨다. 위의 find함수와 다른 점은 없는 문자를 찾으려고 하면 에러가 발생한다는 점이다.

### 문자열 삽입 (join)

```
>>> a= " , "
```

```
>>> a.join('abcd')
'a,b,c,d'
```

“abcd”라는 문자열의 각각의 문자사이에 변수 a의 값인 ','을 삽입한다.

### 대문자를 소문자로 바꾸기(lower)

```
>>> a = "HI"
>>> a.lower()
'hi'
```

대문자를 소문자로 바꾸어준다.

### 왼쪽 공백 지우기 (lstrip)

```
>>> a = " hi"
>>> a.lstrip()
'hi'
```

문자열중 가장 왼쪽의 연속된 공백들을 모두 지운다. 여기서 lstrip에서 'l'이 의미하는 것은 left이다.

### 오른쪽 공백 지우기 (rstrip)

```
>>> a= "hi "
>>> a.rstrip()
```

```
'hi'
```

문자열 중 가장 오른쪽의 연속된 공백들을 모두 지운다. 여기서 `rstrip`에서 'r'이 의미하는 것은 right이다.

### 양쪽 공백 지우기 (`strip`)

```
>>> a = " hi "
>>> a.strip()
'hi'
```

양쪽의 연속된 공백을 모두 지운다.

### 문자열 바꾸기 (`replace`)

```
>>> a = "Life is too short"
>>> a.replace("Life", "Your leg")
'Your leg is too short'
```

`replace`(바꿔야 될 문자열, 바꿀 문자열)처럼 사용해서 문자열 내의 특정한 값을 다른 값으로 치환해 준다.

### 문자열 나누기 (`split`)

```
>>> a = "Life is too short"
>>> a.split()
```

```
[‘Life’, ‘is’, ‘too’, ‘short’]  
>>> a = "a:b:c:d"  
>>> a.split(’:’)  
[‘a’, ‘b’, ‘c’, ‘d’]
```

a.split()처럼 팔호안에 아무런 값도 넣어주지 않으면 공백을 기준으로 문자열을 나누어 준다.

만약 a.split(’:’)처럼 팔호안에 특정한 값이 있을 경우에는 팔호안의 값을 구분자로 해서 문자열을 나누어 준다. 이 나눈 값은 리스트에 하나씩 들어가게 된다. ['Life', 'is', 'too', 'short']나 ['a', 'b', 'c', 'd']는 리스트라는 것인데 조금 후에 자세히 알아볼 것이니 너무 신경쓰지 말도록 하자.

위에서 소개한 문자열 관련 함수들은 문자열 처리에서 매우 사용 빈도가 높은 것들이고 유용한 것들이다. 이 외에도 몇 가지가 더 있지만 자주 사용되는 것들은 아니다.

### 3) 리스트 (List)

지금까지 우리는 숫자와 문자열에 대해서 알아보았다. 하지만 이러한 것들로 프로그래밍을 하기엔 부족한 점이 많다. 예를 들어 1부터 10까지의 숫자들 중 홀수들의 모임인 1, 3, 5, 7, 9라는 집합을 생각해 보자. 이것들을 숫자나 문자열로 표현하기는 쉽지 않다. 파이썬에는 이러한 불편함을 해소 할 수 있는 자료형이 존재한다. 그것이 바로 이곳에서 공부하게 될 리스트이다.

리스트를 이용하면 1, 3, 5, 7, 9라는 숫자의 모임을 다음과 같이 간단하게 표현할 수 있다.

```
>>> odd = [1,3,5,7,9]
```

리스트를 만들 때는 위에서 보는 것과 같이 대괄호([ ])로 감싸주고 안에 들어갈 값들은 쉼표로 구분해준다.

여러가지 리스트의 생김새를 살펴보면 다음과 같다.

```
>>> a = []
>>> b = [1, 2, 3]
>>> c = ['Life', 'is', 'too', 'short']
>>> d = [1, 2, 'Life', 'is']
>>> e = [1, 2, ['Life', 'is']]
```

a처럼 리스트는 아무 것도 포함하지 않는 빈 리스트([])일 수도 있고 b처럼 숫자를 그 요소 값으로 가질 수도 있고, c처럼 문자열을 요소값으로 가질 수 있고 d처럼 숫자와 문자열을 함께 요소값으로 가질 수 있으며 또한 e처럼 리스트 자체를 그 요소 값으로 가질 수도 있다. 즉, 리스트 내에는 어떠한 자료형도 포함시킬 수 있다.

※ 참고. 빈리스트는 다음과 같이 생성할 수도 있다.

```
>>> a = list() # a = [] 과 동일
```

## 리스트의 인덱싱과 슬라이싱

리스트의 경우에도 문자열처럼 인덱싱과 슬라이싱이 가능하다. 백문이 불여일견. 말로 설명하는 것보다 직접 예를 따라해보면서 리스트의 기본 구조를 이해하는 것이 쉽다. 대화형 인터프리터로 예를 따라하며 알아보자.

### 리스트의 인덱싱

a 변수에 [1, 2, 3]이라는 값을 세팅한다.

```
>>> a = [1, 2, 3]
>>> a
[1, 2, 3]
```

리스트 역시 문자열에서처럼 인덱싱이 적용된다.

```
>>> a[0]
1
```

a[0]는 리스트 'a'의 첫 번째 요소값을 말한다.

아래의 예는 리스트의 첫 번째 요소인 a[0]와 세 번째 요소인 a[2]의 값을 더해주었다.

---

```
>>> a[0] + a[2]
```

4

이것은  $1 + 3$  으로 해석되어서 4라는 값을 출력해 준다.

문자열을 공부할 때 이미 알아보았지만 컴퓨터는 숫자를 0부터 세기 때문에  $a[1]$  이 리스트  $a$ 의 첫 번째 요소가 아니라  $a[0]$ 가 리스트  $a$ 의 첫 번째 요소임을 명심하도록 하자.  $a[-1]$ 은 문자열에서와 마찬가지로 리스트  $a$ 의 마지막 요소를 말한다.

```
>>> a[-1]
```

3

이번에는 아래의 예처럼 리스트  $a$ 를 숫자 1, 2, 3과 또 다른 리스트인  $['a', 'b', 'c']$ 를 포함하도록 만들어 보자.

```
>>> a = [1, 2, 3, ['a', 'b', 'c']]
```

다음의 예를 따라해 보자.

```
>>> a[0]
```

1

```
>>> a[-1]
```

$['a', 'b', 'c']$

```
>>> a[3]
```

$['a', 'b', 'c']$

예상한 대로  $a[-1]$ 은 마지막 요소값인  $['a', 'b', 'c']$ 를 나타낸다.  $a[3]$ 은 리스트  $a$ 의 네 번째 요소를 나타내기 때문에 마지막 요소를 나타내는  $a[-1]$ 과 동일한

---

결과값을 보여준다.

그렇다면 여기서 리스트 a에 포함된 ['a', 'b', 'c']라는 리스트의 'a'라는 값을 인덱싱을 이용하여 꺼집어 낼 수 있는 방법은 없을까?

다음의 예를 보도록 하자.

```
>>> a[-1] [0]
```

'a'

위처럼 하면 'a'를 꺼집어 낼 수가 있다. a[-1]은 ['a', 'b', 'c']이고 다시 이것의 첫 번째 요소를 불러오기 위해서 [0]을 붙여준 것이다.

아래의 예도 역시 마찬가지 경우이므로 이해가 될 것이다.

```
>>> a[-1] [1]
```

'b'

```
>>> a[-1] [2]
```

'c'

조금은 복잡하지만 다음의 예를 따라해 보자

```
>>> a = [1, 2, ['a', 'b', ['Life', 'is']]]
```

리스트 a안에 리스트 ['a', 'b', ['Life', 'is']]라는 리스트가 포함되어 있고 그 리스트 안에 역시 리스트 ['Life', 'is']라는 리스트가 포함되어 있다. 삼중 리스트 구조이다.

'Life'라는 문자열만을 꺼집어 내기 위해서는 다음과 같이 해야 한다.

```
>>> a[2] [2] [0]
```

### 'Life'

즉 위의 예는 리스트 a의 세 번째 요소인 리스트[‘a’, ‘b’, [‘Life’, ‘is’]]의 세 번째 요소인 리스트 [‘Life’, ‘is’]의 첫 번째 요소를 말하는 것이다. 이렇듯 리스트를 중첩해서 쓰면 혼란스럽기 때문에 자주 사용되지는 않지만 알아두는 것이 좋을 것이다.

## 리스트의 슬라이싱 알아보기

문자열에서와 마찬가지로 리스트에서도 슬라이싱 기법이 적용된다. 슬라이싱이라는 것은 “나눈다”라는 뜻이라고 했었다. 자, 그럼 리스트의 슬라이싱에 대해서 살펴보도록 하자.

```
>>> a = [1, 2, 3, 4, 5]
>>> a[0:2]
[1, 2]
```

이것을 문자열에서 했던 방법과 비교해서 생각 해보자.

```
>>> a = "12345"
>>> a[0:2]
'12'
```

두 가지가 완전히 동일하게 사용됨을 독자는 이미 눈치 챘을 것이다. 문자열에서 했던 것과 사용법이 완전히 동일하다.

몇가지 예를 더 들어 보도록 하자.

```
>>> a = [1, 2, 3, 4, 5]
```

```
>>> b = a[:2]
>>> c = a[2:]
>>> b
[1, 2]
>>> c
[3, 4, 5]
```

b라는 변수는 리스트 a의 처음 요소부터 2번째 요소까지를 나타내는 리스트이다. 여기서는 a[2] 값인 '3'이 포함되지 않는다. c라는 변수는 리스트 3번째 요소부터 끝까지를 나타내는 리스트이다.

리스트가 포함된 리스트역시 똑같이 적용된다.

```
>>> a = [1, 2, 3, ['a', 'b', 'c'], 4, 5]
>>> a[2:5]
[3, ['a', 'b', 'c'], 4]
>>> a[3][:2]
['a', 'b']
```

위의 예에서 a[3]은 ['a', 'b', 'c']를 나타내기 때문에 a[3][:2]는 ['a', 'b', 'c']의 a[0]에서 a[2]직전까지의 값 즉, ['a', 'b']를 나타내는 리스트가 된다.

### 리스트를 더하고(+) 반복하기(\*)

리스트 역시 + 기호를 이용해서 더할 수가 있고 \* 기호를 이용해서 반복을 할 수가 있다. 문자열과 마찬가지로 리스트에서도 되는지 한번 직접 확인해 보도록 하자.

예 1) 리스트 합치기

---

```
>>> a = [1, 2, 3]
>>> b = [4, 5, 6]
>>> a + b
[1, 2, 3, 4, 5, 6]
```

두 개의 리스트를 ‘+’ 기호를 이용해 합치는 방법이다. 리스트 사이에서 ‘+’ 기호는 두 개의 리스트를 합치는 기능을 한다. 문자열에서 “abc” + “def” = “abcdef”가 되는 것과 같은 이치이다.

#### 예 2) 리스트 반복

```
>>> a = [1, 2, 3]
>>> a * 3
[1, 2, 3, 1, 2, 3, 1, 2, 3]
```

위에서 보듯이 [1, 2, 3]이란 리스트가 세 번 반복되어 새로운 리스트를 만들어내는 것을 볼 수 있다. 문자열에서 “abc” \* 3 = “abcabcabc” 가 되는 것과 같은 이치이다.

#### 리스트의 수정, 변경과 삭제

참고 - 다음의 예들은 서로 연관되어 있으므로 따로따로 실행해 보지 말고 예 1부터 예 4까지 차례대로 진행해 나가야 한다.

#### 예 1) 리스트 수정1

```
>>> a = [1, 2, 3]
>>> a[2] = 4
>>> a
```

---

[1, 2, 4]

a[2]의 요소값 '3'을 '4'로 바꾸었다.

예 2) 리스트 수정2

```
>>> a[1:2]
[2]
>>> a[1:2] = ['a', 'b', 'c']
>>> a
[1, 'a', 'b', 'c', 4]
```

a[1:2] 는 a[1]부터 a[2]까지를 말하는데 a[2]를 포함하지 않는다고 했으므로 a = [1, 2, 4]에서 2값만을 말한다. 즉, a[1:2]를 ['a', 'b', 'c']로 바꾸었으므로 a 리스트에서 2라는 값대신에 ['a', 'b', 'c']라는 값을 대입하게 되는 것이다.

참고 - 여기서 a[1] = ['a', 'b', 'c']라고 하는 것과는 전혀 다른 결과값을 갖게 되므로 주의 하도록 하자. a[1] = ['a', 'b', 'c']는 리스트 a의 두 번째 요소를 ['a', 'b', 'c']로 바꾼다는 말이고 a[1:2]는 a[1]에서 a[2]사이의 리스트를 ['a', 'b', 'c']로 바꾼다는 말이다. 따라서 a[1] = ['a', 'b', 'c']처럼 하면 위와는 달리 리스트 a가 [1, ['a', 'b', 'c'], 4]라는 값으로 변하게 된다.

예 3) 리스트 요소 삭제1

```
>>> a[1:3] = []
>>> a
[1, 'c', 4]
```

지금까지의 리스트 a의 값은 [1, 'a', 'b', 'c', 4]였다. 여기서 a[1:3]은 a의 인덱스 1부터 3까지( $1 \leq a < 3$ ) 즉, a[1], a[2]를 의미하므로 a[1:3] 은 ['a', 'b']이다.

그런데 위의 예에서 보듯이 `a[1:3]`을 `[]`으로 바꾸어 주었기 때문에 `a`는 `a`에서 `['a', 'b']`가 사라진 `[1, 'c', 4]`가 되는 것이다.

예 4) 리스트 요소 삭제 2

```
>>> a
[1, 'c', 4]
>>> del a[1]
>>> a
[1, 4]
```

`del a[x]`는 `x`번째 요소값을 삭제한다. `del` 함수는 파이썬이 자체적으로 가지고 있는 내장 함수로 다음과 같이 사용되어 진다.

`del` 객체

(참고 - 객체란 파이썬에서 사용되는 모든 자료형을 말한다.)

`del a[x:y]`는 `x`번째부터 `y`번째 요소 사이의 값을 삭제한다. 예 4에서는 `a[1]`을 삭제하는 방법을 보여준다.

[참고] 초보가 범하기 쉬운 리스트 연산 오류  
우선 다음과 같은 예를 먼저 만들어보자.

```
>>>a = [1, 2, 3]
>>>a[2] + "hi"
Traceback (innermost last):
File "", line 1, in ?
a[2] + "hi"
TypeError: number coercion failed
TypeError 에러가 발생했다. 에러의 원인은 무엇일까?
```

a[2]는 3이라는 정수인데 “hi”는 문자열이다.  
두 값(정수와 문자열)을 더한다는 것은 상식적으로 맞지 않는 방법  
이다.  
그래서 Type 에러가 발생하는 것이다. 숫자와 문자열을 더할 수는  
없다.

만약 숫자와 문자열을 더해서 '3hi'처럼 만들고 싶다면 숫자 3을 문  
자 '3'으로 바꾸어 주어야 한다.

다음과 같이 할 수 있다.

```
>>> str(a[2]) + "hi"
```

str 함수로 정수를 문자열로 바꾸어 주는 방법이다.

## 리스트 관련 함수들

문자열과 마찬가지로 리스트 변수명 뒤에 '.'을 붙여서 여러 가지 리스트의 관련  
함수들을 이용할 수가 있다. 유용하게 쓰이는 리스트 관련 함수들 몇 가지에 대  
해서만 알아보기로 하자.

### 리스트에 요소 추가 (append)

append의 뜻이 무엇인지 안다면 아래의 예가 금방 이해가 될 것이다. append(x)  
는 리스트의 맨 마지막에 x를 추가시키는 함수이다.

```
>>> a = [1, 2, 3]
>>> a.append(4)
>>> a
[1, 2, 3, 4]
```

---

리스트내에는 어떤 자료형도 추가시킬 수가 있다. 아래의 예는 리스트에 다시 리스트를 추가시킨 결과를 보여준다.

```
>>> a.append([5,6])
>>> a
[1, 2, 3, 4, [5, 6]]
```

### 리스트 정렬(sort)

sort 함수는 리스트의 요소를 순서대로 정렬한다.

```
>>> a = [1, 4, 3, 2]
>>> a.sort()
>>> a
[1, 2, 3, 4]
```

문자 역시 마찬가지로 알파벳 순서로 정렬이 가능하다.

```
>>> a = ['a', 'c', 'b']
>>> a.sort()
>>> a
['a', 'b', 'c']
```

### 리스트 뒤집기(reverse)

reverse 함수는 리스트를 역순으로 뒤집어준다. 하지만 이것이 의미하는 것이 먼저 순서대로 정렬한 다음에 다시 역순으로 정렬하는 것은 아니다. 그저 리스트

그대로를 거꾸로 뒤집는 일을 할 뿐이다.

```
>>> a = ['a', 'c', 'b']
>>> a.reverse()
>>> a
['b', 'c', 'a']
```

### 위치 반환 (index)

index(x) 함수는 리스트에 x라는 값이 있으면 그 위치를 돌려준다.

```
>>> a = [1,2,3]
>>> a.index(3)
2
>>> a.index(1)
0
```

위의 예에서 리스트 a에 있는 3이라는 숫자는 a[2]이므로 2를 돌려주고, 1이라는 숫자는 a[0]이므로 0을 돌려준다.

아래의 예에서 0이라는 값은 a 리스트에 존재하지 않기 때문에 에러가 난다.

```
>>> a.index(0)
Traceback (innermost last):
File "", line 1, in ?
a.index(0)
ValueError: list.index(x): x not in list
```

---

Traceback]란 문장부터 ValueError라는 문장까지가 에러메시지이다.

### 리스트에 요소 삽입 (insert)

insert(a, b)는 리스트의 a번째 위치에 b를 삽입하는 함수이다.

```
>>> a = [1,2,3]
>>> a.insert(0, 4)
[4, 1, 2, 3]
```

위의 예에서는 0번째 자리 즉 첫 번째 자리에 4라는 값을 삽입하라는 뜻이 된다.

```
>>> a.insert(3, 5)
[4, 1, 2, 5, 3]
```

위의 예는 리스트 a의 a[3], 즉 네 번째 자리에 5라는 값을 삽입하라는 뜻이다.

### 리스트 요소 제거 (remove)

remove(x)는 첫번째 나오는 x를 삭제하는 함수이다.

```
>>> a = [1,2,3,1,2,3]
>>> a.remove(3)
[1, 2, 1, 2, 3]
```

위의 예는 a가 3이라는 값을 두개 가지고 있을경우 첫번째 3만을 제거하는 것을 보여준다.

```
>>> a.remove(3)
[1, 2, 1, 2]
```

다시 또 3을 삭제한다.

### 리스트 요소 끄집어내기(pop)

pop() 함수는 리스트의 맨 마지막 요소를 돌려주고 그 요소는 삭제한다.

```
>>> a = [1,2,3]
>>> a.pop()
3
>>> a
[1, 2]
```

위의 예에서 보듯이 a 리스트 [1,2,3]에서 3을 끄집어내어서 최종적으로 [1, 2]만 남는 것을 볼 수 있다.

pop(x)는 리스트의 x번째 요소를 돌려주고 그 요소는 삭제한다.

```
>>> a = [1,2,3]
>>> a.pop(1)
2
>>> a
[1, 3]
```

위의 예에서 보듯이 a.pop(1)는 a[1]의 값을 끄집어낸다.

## 갯수세기 (count)

count(x)는 리스트 중에서 x가 몇 개 있는지를 조사하여 그 갯수를 돌려주는 함수이다.

```
>>> a = [1,2,3,1]
>>> a.count(1)
2
```

위의 예에서는 1이라는 값이 리스트 a에 두 개가 들어 있으므로 2를 돌려준다.

## 리스트 확장(extend)

extend(x)에서 x에는 리스트만 올 수 있다. 원래의 a 리스트에 x 리스트를 더하게 된다.

```
>>> a = [1,2,3]
>>> a.extend([4,5])
>>> a
[1, 2, 3, 4, 5]
```

a.extend([4,5])는 a += [4,5]와 동일하다.

## 4) 튜플 (tuple)

튜플 또는 터플이라고 부른다.

튜플이란 리스트와 몇 가지 점을 제외하곤 모든 것이 동일하다. 그 다른 점은 다음과 같다.

- 리스트는 [ 과 ] 으로 둘러싸지만 튜플은 (과 )으로 둘러싼다.
- 리스트는 그 값을 생성, 삭제, 수정이 가능하지만 튜플은 그 값을 변화시킬 수 없다.

튜플은 다음과 같은 모습이다.

```
>>> t1 = ()
>>> t2 = (1,)
>>> t3 = (1,2,3)
>>> t4 = 1,2,3
>>> t5 = ('a', 'b', ('ab', 'cd'))
```

리스트와 생김새가 거의 비슷하지만, 특이할 만한 점이라면 단지 한 개의 요소만을 갖는 튜플은  $t2 = (1,)$ 처럼 한 개의 요소와 그 뒤에 콤마(',')를 넣어야 한다는 점과 네 번째 보기  $t4 = 1, 2, 3$ 처럼 괄호()를 생략해도 무방하다는 점이다.

얼핏 보면 튜플과 리스트는 비슷한 역할을 한다. 하지만 프로그래밍을 할 때 튜플과 리스트는 구분해서 사용하는것이 유리하다. 튜플과 리스트의 가장 큰 차이는 값을 변화시킬 수 있는지 없는지의 차이라고 했다. 리스트의 항목 값은 변화가 가능하고 튜플의 항목 값은 변화가 불가능하다. 따라서 프로그램이 진행되는 동안 그 값이 항상 변하지 않기를 바란다면 또는 바뀔까 걱정하고 싶지 않다면 주저하지 말고 튜플을 사용해야 할 것이다. 이와는 반대로 수시로 그 값을

---

변화시켜야 할 경우라면 리스트를 사용해야 한다. 실제 프로그램에서는 평균적으로 튜플보다는 리스트를 훨씬 많이 쓰게 된다.

## 튜플의 인덱싱, 슬라이싱, 더하기와 반복

튜플은 값을 변화시킬 수 없다는 점만 제외하면 리스트와 완전히 동일하므로 간단하게만 살펴 보기로 하자. 아래의 예제는 서로 연관 되어 있으므로 예1부터 차례대로 수행해 보기를 바란다.

예 1) 인덱싱

```
>>> t1 = (1, 2, 'a', 'b')
>>> t1[0]
1
>>> t1[3]
'b'
```

문자열, 리스트와 마찬가지로 t1[0], t1[3]처럼 인덱싱이 가능하다.

예 2) 슬라이싱

```
>>> t1 = (1, 2, 'a', 'b')
>>> t1[1:]
(2, 'a', 'b')
```

슬라이싱의 예이다.

예 3) 튜플 더하기(합)

```
>>> t2 = (3, 4)
```

```
>>> t1 + t2  
(1, 2, 'a', 'b', 3, 4)
```

튜플을 합하는 방법을 보여준다.

예 4) 튜플 곱(반복)

```
>>> t2 * 3  
(3, 4, 3, 4, 3, 4)
```

튜플의 곱(반복)을 보여준다.

튜플의 요소 값은 변경시킬 수 없다 튜플의 요소값은 한 번 정하면 지우거나 변경할 수 없다. 다음에 소개하는 두 개의 예를 살펴보면 무슨말인지 알 수 있을 것이다.

예 1) 튜플의 요소를 지우려고 할 때의 에러

```
>>> del t1[0]  
Traceback (innermost last):  
File "", line 1, in ?del t1[0]  
TypeError: object doesn't support item deletion
```

튜플의 요소를 리스트처럼 del 함수로 지우려고 한 경우이다. 튜플은 요소를 지우는 행위가 지원되지 않는다는 메시지를 확인 할 수 있다.

예 2) 튜플 요소값 변경시 에러

```
>>> t1[0] = 'c'  
Traceback (innermost last):  
File "", line 1, in ?t1[0] = 'c'
```

```
TypeError: object doesn't support item assignment
```

튜플의 요소 값을 변경하려고 해도 마찬가지로 에러가 발생하는 것을 확인할 수 있다.

## 5) 딕셔너리 (Dictionary)

‘사람’을 예로 들면 누구든지 “이름” = “홍길동”, “생일” = “몇 월 며칠” 등으로 구분할 수 있다. 파이썬은 영리하게도 이러한 대응관계를 자료형으로 만들었다. 이것은 요즘 나오는 대부분의 언어들도 갖고 있는 자료형으로 Associative array, Hash라고도 불린다.

딕셔너리란 단어 그대로 해석하면 사전이란 뜻이다. 즉, people 이란 단어에 ‘사람’, baseball 이라는 단어에 ‘야구’라는 뜻이 부합되듯이 딕셔너리는 Key와 Value라는 것을 한 쌍으로 갖는 자료형이다. 위의 예에서 보면 Key가 ‘baseball’이라면 Value는 ’야구’가 될 것이다.

딕셔너리는 리스트나 터플처럼 순차적으로(sequential) 해당 요소 값을 구하지 않고 key를 통해 value를 얻는다. 딕셔너리의 가장 큰 특징이라면 key로 value를 얻어낸다는 점이다. baseball이란 단어의 뜻을 찾기 위해서 사전의 내용을 순차적으로 모두 검색하는 것이 아니라 baseball이라는 단어가 있는 곳만을 펼쳐보는 것이다.

### 딕셔너리는 어떻게 생겼을까?

다음은 기본적인 딕셔너리의 모습이다.

```
{Key1:Value1, Key2:Value2, Key3:Value3 ...}
```

즉, Key와 Value쌍들이 여러개가 ‘{과}’으로 둘러싸이고 각각의 요소는 Key : Value 형태로 이루어져 있고 쉼표(,)로 구분되어져 있음을 볼 수 있다.

다음의 딕셔너리 예를 보도록 하자.

```
>>> dic = {'name': 'pey', 'phone': '0119993323', 'birth': '1118'}
```

위에서 key는 각각 ‘name’, ‘phone’, ‘birth’이고 그에 해당하는 value는 ‘pey’, ‘0119993323’, ‘1118’이 된다.

### 딕셔너리 dic의 정보

---

key	value
name	pey
phone	0119993323
birth	1118

---

다음의 예는 Key로 정수값 1을 Value로 ’hi’란 문자열을 사용한 예이다.

```
>>> a = {1: 'hi'}
```

또한 다음의 예처럼 Value에 리스트도 넣을 수 있다.

```
>>> a = { 'a': [1,2,3]}
```

## 딕셔너리 사용하기

딕셔너리는 주로 어떤 것을 표현하는 데 쓸 수 있을까?라는 의문이 들 것이다. 4명의 사람이 있는데 각각의 사람의 특기를 표현할 수 있는 좋은 방법에 대해서 생각해 보자. 리스트나 문자열로는 표현하기가 상당히 까다로울 것이다. 하지만 파이썬의 딕셔너리를 사용한다면 위의 상황을 표현하는 것은 정말 쉽다.

다음의 예를 보자.

```
{ "김연아": "피겨스케이팅", "류현진": "야구", "박지성": "축구", "귀도": "파이썬"}
```

사람이름과 특기를 한쌍으로 하는 딕셔너리이다. 정말 간편하지 않은가? 지금껏 우리는 딕셔너리를 만드는 방법에 대해서만 살펴 보았는데 이것들을 제대로 활용하기 위해서 알아야 할 것들이 있다. 그것들에 대해서 알아보도록 하자.

### Key 이용하여 Value 얻기

다음의 예를 따라해 보도록 하자.

```
>>> grade = {'pey': 10, 'julliet': 99}  
>>> grade['pey']  
10  
>>> grade['julliet']  
99
```

리스트나 터플이나 문자열은 요소값을 얻어내기 위해서 인덱싱이나 슬라이싱이라는 기법을 사용했지만 딕셔너리는 단 한가지의 방법만이 있을 뿐이다. 바로 Key를 이용해서 Value를 얻어내는 방법이다. 위의 예에서처럼 Value를 얻기 위해서는 “딕셔너리변수[Key]” 와 같이 하여 Value를 얻을 수 있다.

몇가지 예를 더 보도록 하자.

```
>>> a = {1:'a', 2:'b'}  
>>> a[1]  
'a'  
>>> a[2]  
'b'
```

먼저 a라는 변수에 {1:'a', 2:'b'}라는 덕셔너리를 대입하였다. 위의 예에서 보듯이 a[1]은 'a'라는 값을 돌려준다. 여기서 a[1]이 의미하는 것은 리스트나 터플의 a[1]과는 아주 다른 것이다. 여기서 [ ] 안의 숫자 1은 몇번째 요소를 뜻하는 것 이 아니라 Key에 해당하는 1을 나타낸다. 덕셔너리는 리스트나 터플의 인덱싱 방법이란 것이 존재하지 않는다. 따라서 a[1]은 덕셔너리 {1:'a', 2:'b'}에서 Key 가 1인것의 Value인 'a'를 돌려주게 된다. a[2] 역시 마찬가지이다.

```
>>> a = {'a':1, 'b':2}
>>> a['a']
1
>>> a['b']
2
```

이번에는 a라는 변수에 위에서 사용했던 덕셔너리의 Key와 Value를 뒤집어 놓은 덕셔너리를 대입해 보았다. 역시 a['a'], a['b']처럼 Key를 이용해서 Value를 얻을 수 있다. 이상 정리해 보면 덕셔너리 a 는 a[Key] 처럼 해서 Key에 해당 하는 Value를 얻을 수 있다.

다음은 위에서 한번 언급했던 덕셔너리인데 Key를 이용하여 Value를 얻는 방법을 잘 보여주고 있다.

```
>>> dic = {'name':'pey', 'phone':'0119993323', 'birth': '1118'}
>>> dic['name']
'pey'
>>> dic['phone']
'0119993323'
>>> dic['birth']
'1118'
```

## 딕셔너리 쌍 추가, 삭제하기

딕셔너리 쌍을 추가하는 방법은 Key를 이용해 Value를 호출했던 것처럼 새로운 Key에 Value를 설정하면 바로 딕셔너리에 추가된다. 예 1부터 예 3까지는 딕셔너리에 쌍을 추가하는 예를 보여준다. 딕셔너리는 순서를 따지지 않는다. 예에서 알 수 있듯이 추가되는 순서는 원칙이 없다. 중요한 것은 “무엇이 추가되었는가”이다.

다음의 예를 함께 따라해 보자.

### 예 1) 딕셔너리 쌍 추가1

```
>>> a = {1: 'a'}
>>> a[2] = 'b'
>>> a
{2: 'b', 1: 'a'}
```

{1: 'a'}라는 딕셔너리에 a[2] = 'b'와 같이 사용해서 2 : 'b'라는 딕셔너리 쌍을 추가하였다.

### 예 2) 딕셔너리 쌍 추가2

```
>>> a['name'] = 'pey'
{'name': 'pey', 2: 'b', 1: 'a'}
```

딕셔너리 a에 'name': 'pey'라는 쌍을 추가한 모습이다.

### 예 3) 딕셔너리 쌍 추가3

---

```
>>> a[3] = [1,2,3]
{'name': 'pey', 3: [1, 2, 3], 2: 'b', 1: 'a'}
```

Key는 3 Value는 [1, 2, 3]을 가지는 한 쌍을 또 추가하였다.

예 4) 딕셔너리 요소 삭제 1

```
>>> del a[1]
>>> a
{'name': 'pey', 3: [1, 2, 3], 2: 'b'}
```

예 4는 딕셔너리의 요소를 지우는 방법을 보여준다. `del a[key]`하면 그에 해당하는 key:value 쌍이 삭제된다.

## 딕셔너리 주의사항

딕셔너리를 만들때 주의해야 할 사항은 Key는 고유한 값이므로 중복되는 값을 설정해 놓으면 하나를 제외한 나머지의 것들은 무시된다는 점이다. 다음 예에서 보듯이 Key가 동일한 것이 존재할 경우 1:'a'라는 쌍이 무시된다. 이때 꼭 딕셔너리를 만들 때 앞에 썼던 것이 무시되는 것은 아니고 어떤 것이 무시될지는 예측이 불가능하다. 결론은 중복되는 Key를 사용하지 말라는 것이다.

```
>>> a = {1:'a', 1:'b'}
>>> a
{1: 'b'}
```

이렇게 중복되었을 때 한 개를 제외한 나머지의 Key:Value값이 무시되는 이유는 딕셔너리는 Key를 통해서 Value를 얻게 되는데 만약 동일한 Key가 존재

한다면 어떤 Key에 해당하는 Value를 불러야 할지 알 수가 없기 때문이다.

또 한 가지 주의해야 할 사항으로는 Key에 리스트는 쓸 수가 없다는 것이다. 하지만 터플은 Key로 쓸 수가 있다. 딕셔너리의 Key로 쓸 수 있고 없고의 구별은 Key가 변하는 값인지 변하지 않는 값인지에 달려 있다. 리스트를 Key로 사용한다면 그 값이 변할 수 있기 때문에 리스트를 Key로 쓸 수 없는 것이다. 아래 예처럼 리스트를 Key로 설정하면 TypeError가 난다.

```
>>> a = {[1,2] : 'hi'}
```

Traceback (most recent call last):  
File "", line 1, in ?  
TypeError: unhashable type

따라서 딕셔너리의 Key를 딕셔너리로 할 수 없음은 당연한 얘기가 될 것이다. Value에는 변하는 값이든 변하지 않는 값이든 상관없이 아무 값이나 넣을 수 있다.

## 딕셔너리 관련함수

딕셔너리를 자유자재로 사용하기 위해 딕셔너리가 자체적으로 가지고 있는 관련 함수들을 사용해 보도록 하자.

### Key리스트 만들기(keys)

```
>>> a = {'name': 'pey', 'phone': '0119993323', 'birth': '1118'}  
>>> a.keys()  
dict_keys(['name', 'phone', 'birth'])
```

---

a.keys()는 딕셔너리 a의 Key만을 모아서 dict\_keys 객체를 리턴한다.

※ 파이썬 2.7 버전까지는 a.keys() 호출 시 리턴값으로 dict\_keys 가 아닌 list 를 리턴한다. list 를 리턴하기 위해서는 메모리의 낭비가 발생하는데 파이썬 3.0 이후버전에서는 이러한 메모리 낭비를 줄이기 위해서 dict\_keys라는 객체를 리턴하게 되었다. 아래 소개될 dict\_values, dict\_items 역시 마찬가지로 파이썬 3.0 이후에 추가된 것들이다. 만약 파이썬 2.7에서와 같이 list 가 필요한 경우에는 list(a.keys()) 로 리스트 형태로 변환하여 사용하면 된다. dict\_keys, dict\_values, dict\_items 등은 리스트로 변환하지 않더라도 기본적인 iterate성 구문(예: for문)들을 실행 할 수 있다.

dict\_keys 객체는 다음과 같이 사용할 수 있다. 리스트를 사용하는 것과 차이가 없다. (※ 리스트 고유의 메소드인 append, insert, pop, remove, sort등의 메소드를 수행 할 수는 없다.)

```
>>> for k in a.keys():
...     print(k)
...
phone
birth
name
```

dict\_keys 객체를 리스트로 변환하려면 다음과 같이 하면 된다.

```
>>> list(a.keys())
['phone', 'birth', 'name']
```

### Value리스트 만들기 (values)

```
>>> a.values()
dict_values(['pey', '0119993323', '1118'])
```

마찬가지 방법으로 value만을 얻고 싶다면 a.values()처럼 values 함수를 사용하면 된다. values 함수 호출 시 dict\_values 객체가 리턴된다. dict\_values 객체 역시 dict.keys 객체와 마찬가지로 리스트를 사용하는 것과 동일하게 사용하면 된다.

### Key, Value 쌍 얻기(items)

```
>>> a.items()
dict_items([('name', 'pey'), ('phone', '0119993323'), ('birth', '1118')])
```

items 함수는 key와 value의 쌍을 터플로 묶은 값을 dict.items 객체로 돌려준다.

### Key: Value 쌍 모두 지우기(clear)

```
>>> a.clear()
>>> a
{}
```

clear() 함수는 딕셔너리 안의 모든 요소를 삭제한다. 위에서 보듯이 빈 리스트가 [], 빈 터플이 ()인 것과 마찬가지로 빈 딕셔너리도 {}과 같이 표현된다.

## Key로 Value얻기 (get)

```
>>> a = {'name': 'pey', 'phone': '0119993323', 'birth': '1118'}
>>> a.get('name')
'pey'
>>> a.get('phone')
'0119993323'
```

`get(x)` 함수는 `x`라는 key에 대응되는 value를 돌려준다. 앞서 살펴 보았듯이 `a.get('name')`은 `a['name']`처럼 사용하는 것과 동일한 결과값을 돌려 받는다. 여기서 주의해야 할 점은 아래 예제에서 보듯이 `a['nokey']`처럼 존재하지 않는 키('nokey')로 값을 가져오려고 할 경우에 `a['nokey']`는 `KeyError`를 발생시키고 `a.get('nokey')`는 `None`을 리턴하는 차이가 있다. 어떤 것을 사용하는가는 독자의 선택이다.

```
>>> a.get('nokey') # None을 리턴
>>> a['nokey']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'nokey'
```

딕셔너리 내에 찾으려는 키값이 없을 경우 디폴트 값으로 가져오도록 하고 싶을 때에는 `get(x, '디폴트값')`을 사용하면 편리하다.

```
>>> a.get('foo', 'bar')
'bar'
```

a 딕셔너리에는 'foo'에 해당하는 값이 없다. 따라서 디폴트 값인 'bar'를 리턴한다.

### 해당 Key가 있는지 조사 (in)

```
>>> a = {'name': 'pey', 'phone': '0119993323', 'birth': '1118'}  
>>> 'name' in a  
True  
>>> 'email' in a  
False
```

'name'이라는 문자열은 a 딕셔너리의 key이다. 따라서 'name' in a 호출시 True를 리턴한다. 반대로 'email'은 a 딕셔너리에 존재하지 않는 키이므로 False를 리턴하게 된다.

## 6) 집합 (Sets)

set은 파이썬 2.3 부터 지원되기 시작한 자료형으로 보통 집합에 관련된 것들을 쉽게 처리하기 위해 만들어진 자료형이다.

set 자료형은 다음과 같이 set키워드를 이용하여 만들 수 있다.

```
>>> s1 = set([1,2,3])
>>> s1
{1, 2, 3}
```

위와 같이 리스트를 입력으로 만들 수 있다.

```
>>> s2 = set("Hello")
>>> s2
{'e', 'l', 'o', 'H'}
```

또는 위와 같이 문자열을 입력으로 만들 수도 있다.

### set의 특징

자, 그런데 위에서 살펴 본 “Hello” set의 결과가 좀 이상하지 않은가? 분명 “Hello”라는 문자열로 set 자료형을 만들었는데 생성된 자료형에는 1 문자가 하나 빠져 있고 순서도 뒤죽박죽이다.

그 이유는 set에는 다음과 같은 두가지 큰 특징이 있기 때문이다.

1. 중복을 허용하지 않는다.
2. 순서가 없다. (unordered)

※ set의 이러한 중복을 허용하지 않는 특징은 자료형의 중복을 제거하기 위한 필터 역할로 종종 사용되기도 한다.

리스트나 튜플은 순서가 있기(ordered) 때문에 인덱싱을 통해서 자료형의 값을 얻을 수 있지만 set 자료형은 순서가 없기(unordered) 때문에 인덱싱으로 값을 얻을 수 없다. 이것은 우리가 이전에 살펴 본 딕셔너리와 비슷하다. 딕셔너리 역시 순서가 없는 자료형이라 인덱싱을 지원하지 않는다.

set 자료형에 저장된 값을 인덱싱으로 접근하려면 다음과 같이 리스트나 터플로 변환한 후 가능하다.

```
>>> s1 = set([1,2,3])
>>> l1 = list(s1)
[1, 2, 3]
>>> l1[0]
1
```

## 교집합, 합집합, 차집합

set 자료형이 정말 유용하게 사용되는 경우는 다음과 같이 교집합, 합집합, 차집합을 구하려 할 때이다.

우선 다음과 같이 두개의 set 자료형을 만들어보자.

```
>>> s1 = set([1,2,3,4,5,6])
>>> s2 = set([4,5,6,7,8,9])
```

s1은 1부터 6까지의 값을 가지게 되었고 s2는 4부터 9까지의 값을 가지게 되었다. 자 이제 이 두개의 자료형의 교집합을 구해 보도록 하자.

```
>>> s1 & s2  
{4, 5, 6}  
>>>
```

& 기호를 이용하면 교집합을 간단히 구할 수 있다.

또는 다음과 같이 intersection 함수를 사용해도 동일한 결과를 리턴한다.

```
>>> s1.intersection(s2)  
{4, 5, 6}
```

합집합은 다음과 같이 만들 수 있다.

```
>>> s1 | s2  
{1, 2, 3, 4, 5, 6, 7, 8, 9}
```

| 기호를 이용한 방법이다.

```
>>> s1.union(s2)  
{1, 2, 3, 4, 5, 6, 7, 8, 9}
```

또는 union 함수를 이용하면 된다.

차집합은 다음과 같이 만들 수 있다.

```
>>> s1 - s2  
{1, 2, 3}
```

- 기호를 이용하는 방법이다.

```
>>> s1.difference(s2)
{1, 2, 3}
```

또는 `difference` 함수를 이용하면 된다.

## 추가와 삭제

한번 만들어진 set 자료형에 값을 추가할 수 있다.

```
>>> s1 = set([1,2,3])
>>> s1.add(4)
>>> s1
{1, 2, 3, 4}
```

한개의 값만 추가(`add`)할 경우에는 위와 같이 한다.

```
>>> s1 = set([1,2,3])
>>> s1.update([4,5,6])
>>> s1
{1, 2, 3, 4, 5, 6}
```

여러개의 값을 한꺼번에 추가(`update`)할 경우에는 위와 같이 하면 된다.

```
>>> s1 = set([1,2,3])
>>> s1.remove(2)
>>> s1
{1, 3}
```

특정 값을 제거(remove)하고 싶을 때에는 위와 같이 하면 된다.

이상과 같이 파이썬에서 사용되는 가장 기본이 되는 자료형인 숫자, 문자열, 리스트, 튜플, 딕셔너리, 집합에 대해서 알아 보았다. 여기까지 잘 따라온 독자라면 파이썬에 대해서 대략 50% 정도 습득했다고 보아도 된다. 그만큼 위에서 언급한 자료형들은 중요하기 때문이다. 책에 있는 예제들만 따라하지 말고 직접 여러 가지 예들을 만들어 보고 테스트해 가며 반복해서 위의 자료형들에 익숙해지기를 당부한다. 왜냐하면 자료형은 프로그램의 근간이 되기 때문에 확실하게 해 놓지 않으면 좋은 프로그램을 만들 수 없기 때문이다.

## 7) 참과 거짓

자료형에 참과 거짓이 있다?

조금 이상한 말처럼 들리지만 참과 거짓은 분명히 있다. 이것은 매우 중요하며 자주 쓰이게 된다. 자료형의 참과 거짓을 구분짓는 기준을 다음과 같이 정리했다.

값	참 or 거짓
“python”	참
“”	거짓
[1, 2, 3]	참
[]	거짓
()	거짓
{}	거짓
1	참
0	거짓
None	거짓

문자열, 리스트, 터플, 덱셔너리 등의 값이 비어 있으면(“”, [], (), {}) 거짓이 된다. 당연히 비어 있지 않으면 참이 된다. 숫자에서는 그 값이 0일 때 거짓이 된다. 위의 표를 보면 None이란 것이 있는데 이것에 대해서는 아직은 신경쓰지 말도록 하자. 그저 None이란 것은 거짓을 뜻한다고만 알아두자.

다음의 예를 보고 참과 거짓이 프로그램에서 어떻게 쓰이는지에 대해서 간단히

---

알아보자.

```
>>> a = [1,2,3,4]
>>> while a:
...     a.pop()
...
4
3
2
1
```

먼저 `a = [1,2,3,4]`라는 리스트를 하나 만들었다.

`while`문은 뒤에서 자세히 다루겠지만 간단히 알아보면 다음과 같다.

```
while <조건문>:
    <수행할 문장>
```

<조건문>이 참인 동안에 <수행할 문장>을 계속 수행한다. 즉, 위의 예에서 보면 `a`가 참인 경우에 `a.pop()`을 계속 실행하라는 의미이다. `a.pop()`이란 함수는 리스트 `a`의 마지막 요소를 끄집어내는 함수이므로 `a`가 참인 동안(리스트 내에 요소가 존재하는 한)에 마지막 요소를 계속해서 끄집어 낼 것이다. 결국 더 이상 끄집어 낼 것이 없으면 `a`가 빈 리스트(`[]`)가 되어 거짓이 될 것이다. 따라서 `while`문은 거짓에 의해서 중지된다. 위에서 본 예는 파이썬 프로그래밍에서 매우 자주 쓰이는 기법중 하나이다.

위의 예가 너무 복잡하다고 생각하는 독자는 다음의 예를 보면 쉽게 이해가 될 것이다.

```
>>> if []:
:
```

```
...     print("True")
... else:
...     print("False")
...
False
```

if문에 대해서 잘 모르는 독자라도 위의 문장을 다음과 같이 해석하는 데 무리가 없을 것이다. (if문에 대해서는 바로 조금후에 자세히 다루게 된다.)

만약 []가 참이면 “True”라는 문자열을 출력하고 그렇지 않으면 “False”라는 문자열을 출력해라.

[]는 위의 테이블에서 보듯이 거짓이기 때문에 “False”란 문자열이 출력된다.

```
>>> if [1,2,3]:
...     print("True")
... else:
...     print("False")
...
True
```

위 코드를 해석해 보면 다음과 같다.

만약 [1,2,3]이 참이면 “True”라는 문자열을 출력하고 그렇지 않으면 “False”라는 문자열을 출력해라.

[1,2,3]은 요소 값이 있는 리스트이기 때문에 참이다. 따라서 “True”를 출력한다.

## 8) 변수

우리는 이미 변수들을 사용해 왔다. 다음 예와 같은 a, b, c를 파이썬 변수라고 한다.

```
>>> a = 1
>>> b = "python"
>>> c = [1,2,3]
```

변수를 만들 때는 위의 예에서와 같이 '='(assignment) 기호를 사용한다.

앞으로 설명할 부분은 프로그래밍 초보자가 얼른 이해하기 어려운 부분이므로 당장 이해가 되지 않는다면 그냥 건너뛰어도 무방하다. 파이썬에 대해서 공부하다 보면 자연스럽게 알 게 될 것이다.

### 변수란

변수는 객체를 가리키는 것이다. 객체란 우리가 지금껏 보아왔던 자료형을 포함한 파이썬에서 사용되는 그 모든 것을 뜻하는 말이다.

```
>>> a = 3
```

만약 위의 코드처럼 a = 3이라고 했다면 3이라는 값을 가지는 정수 자료형(객체)이 자동으로 메모리에 생성된다. a는 변수 이름으로서 3이라는 정수형 객체가 저장된 메모리 위치를 가리키게 된다. 즉, a는 레퍼런스(Reference)이다.

이해가 잘 되지 않는다면 이렇게 생각해보자.

```
a --> 3
```

즉, a라는 변수는 3이라는 정수형 객체를 가리키고 있다.

다음 예를 보자.

```
>>> a = 3  
>>> b = 3  
>>> a is b  
True
```

a가 3을 가리키고 b도 3을 가리킨다. 즉 a = 3을 입력하는 순간 3이라는 정수형 객체가 생성되고 변수 a는 3이란 객체의 메모리 번지를 가리킨다. 다음에 변수 b가 동일한 객체인 3을 가리킨다.

즉 3이라는 정수형 객체를 가리키고 있는 변수는 2개가 된다. 이 두 변수는 가리키고 있는 대상이 동일하다. 따라서 동일한 객체를 가리키고 있는지 아닌지에 대해서 판단하는 파이썬 내장함수인 is 함수를 a is b처럼 실행했을 때 참(True)을 리턴하게 된다. 3이라는 객체를 가리키고 있는 변수의 개수는 2개이다. 이것을 조금 어려운 말로 레퍼런스 카운트(Reference Count, 참조갯수)가 2개라고 한다. 만약 c = 3이라고 한번 더 입력한다면 레퍼런스 카운트는 3이 될 것이다.

## 변수 없애기

3이라는 정수형 객체가 메모리에 생성된다고 했다. 그렇다면 이 값을 메모리에서 없앨 수 있을까? 3이라는 객체를 가리키는 변수들의 개수를 레퍼런스 카운트라 하였는데, 이 레퍼런스 카운트가 0이 되는 순간 3이라는 객체는 자동으로 사라진다. 즉 3이라는 객체를 가리키고 있는 것이 하나도 없을 때 이 3이라는 객체는 메모리에서 사라지게 되는 것이다. 이것을 또한 어려운 말로 쓰레기 수집 - 가비지 컬렉션(Garbage collection)이라고도 한다.

다음의 예는 특정한 객체를 가리키는 변수를 없애는 예를 보여준다.

---

```
>>> a = 3
>>> b = 3
>>> del(a)
>>> del(b)
```

위의 예를 살펴보면 a와 b가 3이란 객체를 가리켰다가 del이란 파이썬 내장함수에 의해서 가리키는 변수 a, b가 사라지게 되고 따라서 레퍼런스 카운트가 0이 되어서 객체 3도 메모리에서 사라지게 된다.

※ 사용한 변수를 del 명령어를 이용하여 일일이 삭제할 필요는 없다. 파이썬이 이 모든것을 자동으로 해 주고 있기 때문이다. (가비지 콜렉션)

### 변수를 만드는 여러 가지 방법

방법 1)

```
>>> a, b = 'python', 'life'
```

방법 1처럼 터플로 a, b에 값을 대입할 수 있다. 이 방법은 아래 소개된 방법 2와 완전히 동일하다.

방법 2)

```
>>> (a, b) = ('python', 'life')
```

방법 1과 방법 2는 사실상 같다. 터플 부분에서도 언급했지만 터플은 괄호를 생략해도 된다.

방법 3)

```
>>> [a,b] = ['python', 'life']
```

방법 3처럼 리스트로 만들 수도 있다.

방법 4)

```
>>> a = b = 'python'
```

여러 개의 변수에 같은 값을 대입할 수도 있다.

위의 방법을 이용하여 파이썬에서는 두 변수 값을 바꾸는 매우 간단하고 쉬운 방법을 쓸 수 있다.

```
>>> a = 3  
>>> b = 5  
>>> a, b = b, a  
>>> a  
5  
>>> b  
3
```

처음에 a에는 3, b에는 5라는 값이 대입되어 있었지만 a, b = b, a라는 문장을 수행한 뒤 그 값이 서로 바뀌었음을 확인 할 수 있다.

## 리스트 복사

여기서는 리스트라는 자료형에서 가장 혼동하기 쉬운 부분을 설명하려고 한다. 예를 보면 알아보도록 하자.

---

```
>>> a = [1,2,3]
>>> b = a
>>> a[1] = 4
>>> a
[1, 4, 3]
>>> b
[1, 4, 3]
```

위의 예를 유심히 살펴보면 b라는 변수에 a가 가리키는 리스트를 대입하였다. 그런 다음, a 리스트의 a[1]을 4라는 값으로 바꾸었을 때, a리스트만이 바뀌는 것이 아니라 b리스트도 똑같이 바뀌게 된다. 그 이유는 a, b 모두 같은 리스트인 [1, 2, 3]을 가리키고 있었기 때문이다. a, b는 이름만 다를 뿐이지 완전히 동일한 리스트를 가리키고 있는 변수이다.

그렇다면 b 변수를 생성할 때 a와 같은 값을 가지면서 a 가 가리키는 리스트와는 다른 리스트를 가리키게 하는 방법은 없을까? 다음의 두 가지 방법이 있다.

#### 방법 1) [:] 이용

첫 번째 방법으로는 아래와 같이 리스트 전체를 가리키는 [:]을 이용하는 것이다.

```
>>> a = [1, 2, 3]
>>> b = a[:]
>>> a[1] = 4
>>> a
[1, 4, 3]
>>> b
[1, 2, 3]
```

위의 예에서 보듯이 a 리스트 값을 변형하더라도 b리스트에는 영향을 끼치지 않음을 볼수 있다.

### 방법 2) copy 모듈 이용

두 번째 방법은 copy 모듈을 이용하는 방법이다. 다음의 예제에서 보면 from copy import copy라는 처음 보는 형태가 나오는데 이것은 2장의 모듈 부분에서 자세히 다루게 된다. 여기서는 단순히 copy라는 함수를 쓰기 위해서 사용되는 것이라고만 알아두자.

```
>>> from copy import copy  
>>> b = copy(a)
```

위의 예 b = copy(a) 는 b = a[:]과 동일하다.

두 변수가 같은 값을 가지면서 다른 객체를 제대로 생성했는지를 확인하려면 다음과 같이 하면 된다. 즉, is 함수를 이용한다. 이 함수는 서로 동일한 객체인지 아닌지에 대한 판단을 하여 참과 거짓을 돌려준다.

```
>>> b is a  
False
```

위의 예에서 b is a 가 False를 돌려주므로 b와 a 가 다른 객체임을 알 수 있다.

## 5. 제어문

이번 장에서는 **if**, **while**, **for**등의 제어문에 대해서 알아볼 것이다.

이러한 것들을 알아보기 전에 집을 짓는 경우를 생각해 보자. 나무, 돌, 시멘트 등은 집을 짓기 위한 재료가 될 것이고, 철근 같은 것은 집의 뼈대가 될 것이다. 이것은 프로그램의 경우와 매우 비슷하다. 즉 나무, 돌, 시멘트 등의 재료는 바로 자료형이 될 것이고 집의 뼈대를 이루는 철근이 바로 우리가 이 곳에서 알아볼 제어문이 될 것이다. 즉, 자료형을 그 근간으로 하여 그것들의 흐름을 원활히 효율적으로 만들어 주는 것, 이것이 바로 지금부터 공부할 제어문이다.

## 1) if문

다음과 같은 상상을 해 보자.

“돈이 있으면 택시를 타고 가고 돈이 없으면 걸어 간다.”

위와 같은 상황은 우리 주변에서 언제든지 일어 날 수 있는 상황중의 하나이다. 프로그래밍이란 것도 사람이 만드는 것이라서 위와 같은 문장처럼 조건을 판단해서 그 상황에 맞게 처리해야 할 경우가 생기게 된다. 이렇듯 조건을 판단하여 해당 조건에 맞는 상황을 수행하는데 쓰이는 것이 바로 if문이다.

위와 같은 상황을 파이썬에서는 다음과 같이 만들 수 있다.

```
>>> money = 1
>>> if money:
...     print("택시를 타고 가라")
... else:
...     print("걸어가라")
...
택시를 타고 가라
```

## if문의 기본 구조

다음의 구조가 if와 else를 이용한 기본적인 구조이다.

```
if <조건문>:
    <수행할 문장1>
    <수행할 문장2>
    ...
else:
```

<수행할 문장A>

<수행할 문장B>

...

조건문을 테스트 해서 참이면 if문 바로 다음의 문장(if 블록)들을 수행하고 조건문이 거짓이면 else문 다음의 문장(else 블록)들을 수행하게 된다.

### 들여쓰기(indentation)

if 문을 만들 때는 다음처럼 if <조건문>: 다음의 문장부터 if문에 속하는 모든 문장들에 들여쓰기를 해 주어야 한다.

```
if <조건문>:  
    <수행할 문장1>  
    <수행할 문장2>  
    <수행할 문장3>
```

위에서 보는 것과 같이 조건문이 참일 경우 <수행할 문장1>을 들여쓰기 하였고 <수행할 문장2>, <수행할 문장3>도 들여쓰기를 해 주었다. 이것은 파이썬을 처음 배우는 사람들에게 매우 혼동스러운 부분이기도 하니 여러번 연습을 해 보는 것이 좋다.

다음처럼 하면 에러가 난다.

```
if <조건문>:  
    <수행할 문장1>  
    <수행할 문장2>  
    <수행할 문장3>
```

<수행할 문장2>가 들여쓰기가 되지 않았다.

또는

```
if <조건문>:  
    <수행할 문장1>  
    <수행할 문장2>  
    <수행할 문장3>
```

<수행할 문장3>이 들여쓰기는 되었지만 <수행할 문장1>이나 <수행할 문장2> 와의 들여쓰기의 깊이가 틀리다. 즉 들여쓰기는 언제나 같은 깊이로 해 주어야 한다.

그렇다면 들여쓰기는 공백으로 하는 것이 좋을까? 아니면 탭으로 하는 것이 좋을까? 이것에 대한 논란은 파이썬을 사용하는 사람들 사이에서 아직도 계속되고 있는 것인데 탭으로 하자는 쪽, 공백으로 하자는 쪽 모두다 일치하는 내용은 단 하나, 둘을 혼용해서 쓰지 말자는 것이다. 공백으로 할 거면 항상 공백으로 하고 탭으로 할 거면 항상 탭으로 하자는 말이다. 탭이나 공백은 프로그램 소스에서 눈으로 보이는 것이 아니기 때문에 혼용해서 쓰면 에러의 원인이 되기도 한다. 주의 하도록 하자.

(※ 요새는 들여쓰기 시 스페이스만을 사용하는 것이 파이썬 프로그램 커뮤니티에서 보편적으로 권장되는 방식이다.)

#### [참고] `:`을 잊지 말자

if(조건문) 다음에는 반드시 콜론(`:`)이 오게 된다. 이것은 특별한 의미라기보다는 파이썬의 문법 구조이다. 왜 하필이면 `:`인가에 대해서 궁금하다면 파이썬을 만든 Guido에게 직접 물어보아야 할 것이다. while이나 for, def, class 에도 역시 그 문장의 끝에 `:`이 항상 들어간다. 이 `:`을 가끔 빠뜨리는 경우가 많은데 주의하도록 하자.

파이썬이 다른 언어보다 보기 쉽고 소스코드가 간결한 이유가 바로 `:`을 사용하여 들여쓰기(indentation)를 하게끔 만들기 때문이다. 하지만 숙련된 프로그래머들이 파이썬을 접할 때 가장 혼란스러운 부분이기도 하다.

if 문을 다른 언어에서는 { } 이런 기호로 감싸지만 파이썬에서는 들여쓰기로 해결한다.

## 조건문이란 무엇인가?

if <조건문>에서 사용되는 조건문이란 참과 거짓을 판단하는 문장을 말한다. 자료형의 참과 거짓에 대해서는 이미 알아 보았지만 몇가지만 다시 알아보면 다음과 같은 것들이 있다.

### 자료형의 참과 거짓

자료형	참	거짓
숫자	3	0
문자열	“abc”	“”
리스트	[1,2,3]	[]
터플	(1,2,3)	()
딕셔너리	{“a”:“b”}	{}

따라서 위의 예에서 보았던

```
>>> money = 1
>>> if money:
```

에서 조건문은 money가 되고 money는 1이기 때문에 참이 되어 if문 다음의 문장을 수행하게 되는 것이다.

### 비교연산자

조건판단을 하는 경우는 자료형보다는 비교 연산자( $<$ ,  $>$ ,  $==$ ,  $!=$ ,  $>=$ ,  $<=$ )를 쓰는 경우가 훨씬 많다.

다음 표는 비교연산자를 잘 설명해 준다.

비교연산자      설명	
$x < y$	x가 y보다 작다
$x > y$	x가 y보다 크다
$x == y$	x와 y가 같다
$x != y$	x와 y가 같지 않다
$x >= y$	x가 y보다 크거나 같다
$x <= y$	x가 y보다 작거나 같다

예를 통해서 위의 연산자들에 대해서 알아보자.

```
>>> x = 3  
>>> y = 2  
>>> x > y  
True  
>>>
```

x에 3을 y에 2를 대입한 다음에  $x > y$ 라는 조건문을 수행하니까 True를 리턴한다. 그 이유는  $x > y$ 라는 조건문이 참이기 때문이다.

```
>>> x < y  
False
```

위의 조건문은 거짓이기 때문에 False를 리턴한다.

```
>>> x == y  
False
```

x와 y는 같지 않다. 따라서 위의 조건문은 거짓이다.

```
>>> x != y  
True
```

x와 y는 같지 않다. 따라서 위의 조건문은 참이다.

앞의 경우를 다음처럼 바꾸어 보자.

“만약 3000원 이상의 돈을 가지고 있으면 택시를 타고 그렇지 않으면 걸어가라”

위의 상황을 다음처럼 프로그래밍 할 수 있을 것이다.

```
>>> money = 2000  
>>> if money >= 3000:  
...     print("택시를 타고 가라")  
... else:  
...     print("걸어가라")
```

...

걸어가라

>>>

money  $\geq 3000$  이란 조건문이 거짓이 되기 때문에 else문 다음의 문장을 수행하게 된다.

### and, or, not

또 다른 조건 판단에 쓰이는 것으로 and, or, not이란 것이 있다.

각각의 연산자는 다음처럼 동작을 한다.

---

연산자	설명
x or y	x와 y 둘중에 하나만 참이면 참이다
x and y	x와 y 모두 참이어야 참이다
not x	x가 거짓이면 참이다

---

다음의 예를 통해 위의 사항을 반영해 보도록 하자.

“돈이 3000원 이상 있거나 카드가 있다면 택시를 타고 그렇지 않으면 걸어가라”

```
>>> money = 2000
>>> card = 1
>>> if money >= 3000 or card:
...     print("택시를 타고 가라")
... else:
...     print("걸어가라")
```

...  
 택시를 타고 가라  
 >>>

money는 2000이지만 card가 1이기 때문에 money  $\geq 3000$  or card라는 조건문이 참이 되기 때문에 if문 다음의 문장이 수행된다.

### x in s, x not in s

더 나아가서 파이썬에서는 조금 더 재미있는 조건문들을 제공한다. 바로 다음과 같은 것들이다.

---

in	not in
x in 리스트	x not in 리스트
x in 튜플	x not in 튜플
x in 문자열	x not in 문자열

---

‘in’이라는 영어단어가 ’~안에’라는 뜻을 가졌음을 상기해 보면 다음의 예들이 쉽게 이해가 될 것이다.

```
>>> 1 in [1, 2, 3]
True
>>> 1 not in [1, 2, 3]
False
```

위의 첫 번째 예는 “[1, 2, 3]이라는 리스트 안에 1이 있는가?”라는 조건문이다.

1은 [1, 2, 3]안에 있으므로 참이 되어 True를 리턴한다. 두 번째 예는 “[1, 2, 3]이라는 리스트 안에 1이 없는가?”라는 조건문이다. 1은 [1, 2, 3]안에 있으므로 거짓이 되어 False를 리턴한다.

다음은 튜플과 문자열의 적용예를 보여준다.

```
>>> 'a' in ('a', 'b', 'c')  
True  
>>> 'j' not in 'python'  
True
```

위의 것들을 이용하여 우리가 계속 사용해온 택시예제에 적용시켜 보자.

“만약 주머니에 돈이 있으면 택시를 타고, 없으면 걸어가라”

```
>>> pocket = ['paper', 'cellphone', 'money']  
>>> if 'money' in pocket:  
...     print("택시를 타고 가라")  
... else:  
...     print("걸어가라")  
...  
택시를 타고 가라  
>>>
```

['paper', 'cellphone', 'money']라는 리스트에 안에 'money'가 있으므로 'money' in pocket은 참이 되어서 if문 다음의 문장이 수행되었다.

## elif (다중 조건 판단)

if와 else만을 가지고서는 다양한 조건 판단을 하기가 어렵다. 다음과 같은 예만 하더라도 if와 else만으로는 조건 판단에 어려움을 겪게 된다.

“지갑에 돈이 있으면 택시를 타고, 지갑엔 돈이 없지만 카드가 있으면 택시를 타고, 돈도 없고 카드도 없으면 걸어가라”

위의 문장을 보면 조건을 판단하는 부분이 두 군데가 있다. 먼저 지갑에 돈이 있는지를 판단해야 하고 지갑에 돈이 없으면 다시 카드가 있는지를 판단한다.

if와 else만으로 위의 문장을 표현 하려면 다음과 같이 할 수 있을 것이다.

```
>>> pocket = ['paper', 'handphone']
>>> card = 1
>>> if 'money' in pocket:
...     print("택시를 타고가라")
... else:
...     if card:
...         print("택시를 타고가라")
...     else:
...         print("걸어가라")
...
택시를 타고가라
>>>
```

언뜻 보기에도 이해하기가 쉽지 않고 산만한 느낌이 든다. 위와 같은 점을 보완하기 위해서 파이썬에서는 다중 조건 판단을 가능하게 하는 elif라는 것을 사용한다.

위의 예를 elif를 이용하면 다음과 같이 할 수 있다.

```
>>> pocket = ['paper', 'cellphone']
>>> card = 1
>>> if 'money' in pocket:
...     print("택시를 타고가라")
... elif card:
...     print("택시를 타고가라")
... else:
...     print("걸어가라")
...
택시를 타고가라
```

즉, elif는 이전 <조건문>이 거짓일 때 수행되게 된다. if, elif, else의 기본 구조는 다음과 같다.

```
If <조건문>:
    <수행할 문장1>
    <수행할 문장2>
    ...
elif <조건문>:
    <수행할 문장1>
    <수행할 문장2>
    ...
elif <조건문>:
    <수행할 문장1>
    <수행할 문장2>
    ...
else:
    <수행할 문장1>
```

### <수행할 문장2>

...

위에서 보듯이 elif는 개수에 제한 없이 사용할 수 있다. (참고: 이것은 마치 C 언어의 switch문과 비슷한 것이다.)

### pass의 사용

가끔 조건문을 판단하고 참 거짓에 따라 행동을 정의 할 때 아무런 일도 하지 않게끔 설정을 하고 싶을 때가 생기게 된다. 다음의 예를 보자.

“지갑에 돈이 있으면 가만히 있고 지갑에 돈이 없으면 카드를 꺼내라”

위의 예를 pass를 적용해서 구현해 보자.

```
>>> pocket = ['paper', 'money', 'cellphone']
>>> if 'money' in pocket:
...     pass
... else:
...     print("카드를 꺼내라")
...
```

pocket이라는 리스트 안에 'money'란 문자열이 있기 때문에 if문 다음문장인 pass가 수행되었고 아무런 결과값도 보여주지 않는 것을 확인 할 수 있다.

### 한줄 짜리 if문

위의 예를 보면 if문 다음의 수행할 문장이 한줄이고 else문 다음에 수행할 문장도 한줄이다. 이렇게 수행할 문장이 한줄일때 조금 더 간편한 방법이 있다.

위에서 알아본 pass를 사용한 예는 다음처럼 간략화 할 수 있다.

```
>>> pocket = ['paper', 'money', 'cellphone']
>>> if 'money' in pocket: pass
... else: print("카드를 꺼내라")
...
```

if문 다음의 수행할 문장을 ':'뒤에 바로 적어 주었다. else문 역시 마찬가지이다. 이렇게 하는 이유는 때때로 이렇게 하는 것이 보기에 편하게 때문이다.

## 2) while문

반복해서 문장을 수행해야 할 경우 while문을 사용한다.

다음은 while문의 기본 구조이다.

```
while <조건문>:  
    <수행할 문장1>  
    <수행할 문장2>  
    <수행할 문장3>  
    ...
```

조건문이 참인 동안 while문에 속하는 문장들이 반복해서 수행된다.

“열 번 찍어 안 넘어 가는 나무 없다”라는 속담을 파이썬에 적용시켜 보면 다음과 같이 될 것이다.

```
>>> treeHit = 0  
>>> while treeHit < 10:  
...     treeHit = treeHit + 1  
...     print("나무를 %d번 찍었습니다." % treeHit)  
...     if treeHit == 10:  
...         print("나무 넘어갑니다.")  
... 
```

나무를 1번 찍었습니다.  
 나무를 2번 찍었습니다.  
 나무를 3번 찍었습니다.  
 나무를 4번 찍었습니다.  
 나무를 5번 찍었습니다.  
 나무를 6번 찍었습니다.

나무를 7번 찍었습니다.  
나무를 8번 찍었습니다.  
나무를 9번 찍었습니다.  
나무를 10번 찍었습니다.  
나무 넘어갑니다.

위의 예에서 while문의 조건문은 `treeHit < 10` 이다. 즉 `treeHit`가 10보다 작은 동안에 while 문 안의 문장들을 계속 수행하게 된다. whlie문 안의 문장을 보면 제일 먼저 `treeHit = treeHit + 1`로 `treeHit`값이 계속 1씩 증가한다. 그리고 나무를 `treeHit`번 만큼 찍었음을 알리는 문장을 출력하고 `treeHit`가 10이 되면 “나무 넘어갑니다”라는 문장을 출력하고 `treeHit < 10`라는 조건문이 거짓이 되어 while문을 빠져 나가게 된다.

여기서 `treeHit = treeHit + 1`은 프로그래밍을 할 때 매우 자주 쓰이는 기법으로 `treeHit`의 값을 1만큼씩 증가시킬 목적으로 쓰이는 것이다. 이것은 `treeHit += 1`처럼 쓰기도 한다.

## 무한루프(Loop)

이번에는 무한루프에 대해서 알아보기로 하자. 무한 루프라 함은 무한히 반복한다는 의미이다. 파이썬에서 무한루프는 while문으로 구현 할 수가 있다. 우리가 사용하는 프로그램들 중에서 이 무한루프의 개념을 사용하지 않는 프로그램은 하나도 없을 정도로 이 무한루프는 자주 사용된다. 다음은 무한루프의 기본적인 형태이다.

```
while True:  
    <수행할 문장1>  
    <수행할 문장2>  
    ...
```

while의 조건문이 True이므로 조건문은 항상 참이 된다. while은 조건문이 참인 동안에 while에 속해 있는 문장들을 계속해서 수행하므로 위의 예는 무한하게 while문 내의 문장들을 수행할 것이다.

다음의 예를 보자.

```
>>> while True:  
...     print("Ctrl-C를 눌러야 while문을 빠져 나갈 수 있습니다.")  
...  
Ctrl-C를 눌러야 while문을 빠져 나갈 수 있습니다.  
Ctrl-C를 눌러야 while문을 빠져 나갈 수 있습니다.  
Ctrl-C를 눌러야 while문을 빠져 나갈 수 있습니다.  
....
```

위의 문장이 영원히 출력될 것이다. Ctrl-C를 눌러서 빠져 나가도록 하자. 하지만 위처럼 무한루프를 돌리는 경우는 거의 없을 것이다. 보다 실용적인 예를 들어보자.

다음을 따라해 보도록 하자.

```
>>> prompt = """  
... 1. Add  
... 2. Del  
... 3. List  
... 4. Quit  
...  
... Enter number: """  
>>>
```

먼저 위와같이 여러줄짜리 문자열을 만든다.

```
>>> number = 0  
>>> while number != 4:  
...     print(prompt)  
...     number = int(input())  
...
```

1. Add
2. Del
3. List
4. Quit

Enter number:

다음에 number라는 변수에 0이라는 값을 먼저 대입한다. 이렇게 하는 이유는 다음에 나올 while문의 조건문이 number != 4인데 number라는 변수를 먼저 설정해 놓지 않으면 number라는 변수가 존재하지 않는다는 에러가 나기 때문이다. while문을 보면 number가 4가 아닌 동안에 prompt를 출력하게 하고 사용자로부터 입력을 받아 들인다. 위의 예는 사용자가 4라는 값을 입력하지 않으면 무한히 prompt를 출력하게 된다. 여기서 number = int(input())은 사용자의 숫자 입력을 받아들이는 것이라고만 알아두자. int나 input함수에 대한 사항은 뒤의 내장함수 부분에서 자세하게 다룰 것이다.

## while문 빠져 나가기(break)

while 문은 조건문이 참인 동안 계속해서 while문 안의 내용을 수행하게 된다. 하지만 강제로 while문을 빠져나가고 싶을 때가 생기게 된다.

커피 자판기를 생각해 보자. 커피가 자판기 안에 충분하게 있을 때는 항상 “돈을 받으면 커피를 줘라”라는 조건문을 가진 while문이 수행된다. 자판기가 제 역할을 하려면 커피의 양을 따로이 검사를 해서 커피가 다 떨어지면 while문을 멈추게 하고 “판매중지”란 문구를 자판기에 보여야 할 것이다. 이렇게 while문을 강제로 멈추게 하는 것을 가능하게 해 주는 것이 바로 break이다.

다음의 예는 위의 가정을 파이썬으로 표현해 본 것이다.

예) break의 사용

```
>>> coffee = 10
>>> money = 300
>>> while money:
...     print("돈을 받았으니 커피를 줍니다.")
...     coffee = coffee -1
...     print("남은 커피의 양은 %d 입니다." % coffee)
...     if not coffee:
...         print("커피가 다 떨어졌습니다. 판매를 중지합니다.")
...         break
...
...
```

money가 300으로 고정되어 있으니까 while money:에서 조건문인 money는 0이 아니기 때문에 항상 참이다. 따라서 무한 루프를 돌게 된다. 그리고 while 문의 내용을 한번 수행할 때마다 coffee = coffee - 1에 의해서 coffee의 개수가 한 개씩 줄어들게 된다. 만약 coffee가 0이 되면 if not coffee:라는 문장에서 not coffee가 참이 되므로 if문 다음의 문장들이 수행이 되고 break가 호출되어 while문을 빠져 나가게 된다.

하지만 실제 자판기는 위처럼 작동하지는 않을 것이다. 다음은 자판기의 실제 과정과 비슷하게 만들어 본 예이다. 이해가 안 되더라도 걱정하지 말자. 아래의

예는 조금 복잡하니까 대화형 인터프리터를 이용하지 말고 에디터를 이용해서 작성해 보자.

```
# -*- coding: utf-8 -*-
# coffee.py

coffee = 10
while True:
    money = int(input("돈을 넣어 주세요: "))
    if money == 300:
        print("커피를 줍니다.")
        coffee = coffee -1
    elif money > 300:
        print("거스름돈 %d를 주고 커피를 줍니다." % (money -300))
        coffee = coffee -1
    else:
        print("돈을 다시 돌려주고 커피를 주지 않습니다.")
        print("남은 커피의 양은 %d개 입니다." % coffee)
    if not coffee:
        print("커피가 다 떨어졌습니다. 판매를 중지 합니다.")
        break
```

위의 프로그램 소스를 따로 설명하지는 않겠다. 독자가 위의 것을 이해할 수 있다면 지금껏 알아온 if문이나 while문을 마스터 했다고 보면 된다. 다만 money = int(input("돈을 넣어 주세요: "))라는 문장은 사용자로부터 입력을 받는 부분이고 입력 받은 숫자를 money라는 변수에 대입하는 것이라고만 알아두자.

※ 파이썬 2.7 버전을 사용한다면 위 소스코드에서 input("돈을 넣어 주세요: ") 대신 raw\_input("돈을 넣어 주세요: ")으로 사용

해야 하며 소스코드 가장 첫줄에 다음과 같은 문자열을 반드시 넣어주어야만 한다. (참고: 파이썬 2.7 vs 파이썬 3)

```
# -*- coding: utf-8 -*-
```

만약 위의 프로그램 소스를 에디터로 작성해서 실행시키는 법을 모른다면 부록의 “에디터 사용법”을 참고하도록 하자.

## 조건문으로 돌아가기(continue)

while문 안의 문장을 수행할 때 어떤 조건을 검사해서 조건에 맞지 않는 경우 while문을 빠져나가는 것이 아니라 다시 while문의 맨 처음(조건문)으로 돌아가게 하고 싶을 경우가 생기게 된다. 만약 1부터 10까지의 수중에서 홀수만을 출력하는 것을 while문을 이용해서 작성한다고 생각해 보자. 어떤 방법이 좋을까?

예) continue의 사용

```
>>> a = 0
>>> while a < 10:
...     a = a+1
...     if a % 2 == 0: continue
...     print(a)
...
1
3
5
7
9
```

위의 예는 1부터 10까지의 수 중 홀수만을 출력하는 예이다. a가 10보다 작은

동안 a는 1만큼씩 계속 증가한다. if  $a \% 2 == 0$  (2로 나누었을 때 나머지가 0인 경우)이 참이 되는 경우는 a가 짝수일 때이다. 즉, a가 짝수이면 continue 문장을 수행한다. 이 continue문은 while문의 맨 처음(조건문:  $a < 10$ )으로 돌아 가게 하는 명령어이다. 따라서 위의 예에서 a가 짝수이면 print(a)는 수행되지 않을 것이다.

### 3) for문

파이썬의 특징을 가장 잘 대변해주는 것이 바로 이 for문이다. for문은 매우 유용하고 사용할 때 문장 구조가 한눈에 들어오며 이것을 잘만 쓰면 프로그래밍이 즐겁기까지 하다.

#### for문의 기본구조

for 문의 기본적인 구조는 다음과 같다.

```
for 변수 in 리스트(또는 튜플, 문자열):
    <수행할 문장1>
    <수행할 문장2>
    ...
    ...
```

리스트의 첫 번째 요소부터 마지막 요소까지 차례로 변수에 대입되어 <수행할 문장1>, <수행할 문장2> 등이 수행된다.

#### 예제를 통해 for 알아보기

for문은 예제를 통해서 보는 것이 가장 알기 쉽다. 예제를 따라해 보도록 하자.

예 1) 전형적인 for문

```
>>> test_list = ['one', 'two', 'three']
>>> for i in test_list:
...     print(i)
...
one
```

```
two  
three
```

[‘one’, ‘two’, ‘three’]라는 리스트의 첫 번째 요소인 ‘one’이 먼저 i변수에 대입된 후 print(i)라는 문장을 수행한다. 다음에 ‘two’라는 두 번째 요소가 i변수에 대입된 후 print(i)문장을 수행하고 리스트의 마지막 요소까지 이것을 반복한다.

for문의 쓰임새를 알기 위해서 다음을 가정해 보자.

“총 5명의 학생이 시험을 보았는데 시험점수가 60점이 넘으면 합격이고 그렇지 않으면 불합격이다. 합격인지 불합격인지에 대한 결과를 보여주시오.”

우선 5명의 학생의 시험성적을 리스트로 표현 해 보았다.

```
mark = [90, 25, 67, 45, 80]
```

1번 학생은 90점이고 5번 학생은 80점이다.

이런 점수를 차례로 검사해서 합격했는지 불합격했는지에 대한 통보를 해주는 프로그램을 만들어 보자. 역시 에디터로 만들어 보자.

```
# marks1.py  
marks = [90, 25, 67, 45, 80]  
  
number = 0  
for mark in marks:  
    number = number +1  
    if mark >= 60:  
        print("%d번 학생은 합격입니다." % number)  
    else:  
        print("%d번 학생은 불합격입니다." % number)
```

각각의 학생에게 번호를 붙이기 위해서 number라는 변수를 이용하였다. 점수 리스트인 marks에서 차례로 점수를 꺼내어 mark라는 변수에 대입하고 for안의 문장들을 수행하게 된다. 우선 for문이 한번씩 수행될 때마다 number는 1씩 증가하고 mark가 60이상이면 합격 메시지를 출력하고 60을 넘지 않으면 불합격 메시지를 출력한다.

## for와 continue

while문에서 알아보았던 continue가 for문에서도 역시 동일하게 적용이 된다. 즉, for문 안의 문장을 수행하는 도중에 continue문을 만나면 for문의 처음으로 돌아가게 된다.

위의 예제를 그대로 이용해서 60점 이상인 사람에게는 축하 메시지를 보내고 나머지 사람에게는 아무런 메시지도 전하지 않는 프로그램을 만들어 보자.

```
# marks2.py
marks = [90, 25, 67, 45, 80]

number = 0
for mark in marks:
    number = number +1
    if mark < 60: continue
    print("%d번 학생 축하합니다. 합격입니다. " % number)
```

점수가 60점 이하인 학생일 경우에는 `mark < 60` 참이 되어 continue문이 수행된다. 따라서 축하 메시지를 출력하는 부분인 print문을 수행하지 않고 for 문의 첫부분으로 돌아가게 된다.

## for와 range함수

for문은 range라는 숫자 리스트를 자동으로 만들어 주는 함수와 함께 사용되는 경우가 많다. 다음은 range함수의 간단한 사용법이다.

```
>>> a = range(10)
>>> a
range(0, 10)
```

위에서 보는 것과 같이 range(10)은 0부터 10미만의 숫자 range객체를 만들어 준다.

시작 번호와 끝 번호를 지정하려면 다음과 같이 해야 한다. 끝 번호는 포함되지 않는다.

```
>>> a = range(1, 11)
>>> a
range(1, 11)
```

위처럼 시작 숫자를 정해 줄 수도 있다.

for와 range를 이용하면 1부터 10까지 더하는 것을 다음과 같이 쉽게 구현할 수 있다.

예 ) 1부터 10까지의 합

```
>>> sum = 0
>>> for i in range(1, 11):
...     sum = sum + i
...
>>> print(sum)
```

range(1, 11)은 숫자 1부터 10까지의(1이상 11미만의) 숫자를 데이터로 갖는 자료형이다. 따라서 위의 예에서 i변수에 리스트의 숫자들이 하나씩 차례로 대입되면서 sum = sum + i라는 문장을 수행하게 되어 sum은 최종적으로 55가 되게 된다.

또한 우리가 앞서 살펴 보았던 60점 이상이면 합격인 예제도 range함수를 이용해서 적용시킬 수도 있다. 다음을 보자.

```
#marks3.py
marks = [90, 25, 67, 45, 80]

for number in range(len(marks)):
    if marks[number] < 60: continue
    print("%d번 학생 축하합니다. 합격입니다." % (number+1))
```

len이라는 함수가 처음 나왔는데 len함수는 리스트의 요소 개수를 돌려주는 함수이다. 따라서 위에서 len(marks)는 5가 될것이고 range(len(marks))는 range(5)가 될 것이다. 따라서 number변수에는 차례로 0부터 4까지의 숫자가 대입될 것이고 makrs[number]는 차례대로 90, 25, 67, 45, 80이라는 값을 갖게 될 것이다.

### 다양한 for문의 사용

```
>>> a = [(1,2), (3,4), (5,6)]
>>> for (first, last) in a:
...     print(first + last)
```

```
...  
3  
7  
11
```

위의 예는 a리스트의 요소 값이 튜플이기 때문에 각각의 요소들이 자동으로 (first, last)라는 변수에 대입되게 된다.

이것은 이미 살펴보았던 터플을 이용한 변수 값 대입방법과 매우 비슷한 경우이다.

```
>>> (first, last) = (1, 2)
```

### for와 range를 이용한 구구단

for와 range함수를 이용하면 단 4줄만으로 구구단을 출력해 볼 수가 있다.

```
>>> for i in range(2,10):  
...     for j in range(1, 10):  
...         print(i*j, end=" ")  
...     print(',')  
  
2 4 6 8 10 12 14 16 18  
3 6 9 12 15 18 21 24 27  
4 8 12 16 20 24 28 32 36  
5 10 15 20 25 30 35 40 45  
6 12 18 24 30 36 42 48 54  
7 14 21 28 35 42 49 56 63
```

```
8 16 24 32 40 48 56 64 72
9 18 27 36 45 54 63 72 81
```

위의 예를 보면 for가 두 번 사용되었다. range(2, 10)은 2부터 9까지의 숫자가 차례로 i에 대입된다. i가 처음 2일 때 두 번째 for 문을 만나게 된다. range(1, 10)은 1부터 10까지의 숫자가 j에 대입되고 그 다음 문장인 print(i\*j)를 수행한다. 따라서 i가 2일 때  $2*1, 2*2, 2*3, \dots, 2*9$  까지 차례로 수행되며 그 값을 출력하게 된다. 그 다음에는 i가 3일 때 역시 2일 때와 마찬가지로 수행될 것이고 i가 9일 때까지 계속 반복되게 된다.

위에서 print(i\*j, end=" ")처럼 end 파라미터를 넣어준 이유는 해당 결과값을 출력할 때 다음 줄로 넘어가지 않고 그 줄에 계속해서 출력하기 위한 것이다. 그 다음의 print(" ")은 2단, 3단, , 등을 구분하기 위해서 두 번째 for문이 끝나면 결과 값을 다음 줄부터 출력하게 해주는 문장이다.

※ 파이썬 2.7 버전의 경우 위 예제의 print(i\*j, end=" ") 대신  
`print i*j,`와 같이 콤마(.)를 마지막에 넣어주어야 한다. (참고:  
[파이썬 2.7 vs 파이썬 3](#))

지금껏 우리는 프로그램의 흐름을 제어하는 if, while, for등에 대해서 알아보았다. 독자는 while과 for를 보면서 두 가지가 아주 흡사하다는 느낌을 받았을 것이다. 실제로 for문을 쓰는 부분을 while로 바꿀 수 있는 경우가 많고 while문을 쓴 부분을 for문으로 바꾸어서 사용할 수 있는 경우가 많다.

## 리스트 내포(List comprehension)

리스트 안에 for문을 이용하면 좀 더 편리하고 직관적이게 프로그램을 만들 수 있다. 다음의 예제를 보자.

```
>>> a = [1,2,3,4]
```

```
>>> result = []
>>> for num in a:
...     result.append(num*3)
...
>>> print(result)
[3, 6, 9, 12]
```

위 예제는 a라는 리스트의 각 항목에 3을 곱한 결과를 result라는 리스트에 담는 예제이다. 이것을 리스트 내포를 이용하면 다음과 같이 간단히 해결할 수 있다.

```
>>> result = [num * 3 for num in a]
>>> print(result)
[3, 6, 9, 12]
```

만약 3을 곱한 수 중 짝수만 담고 싶다면 다음과 같이 할 수 있다.

```
>>> result = [num * 3 for num in a if num % 2 == 0]
>>> print(result)
[6, 12]
```

리스트 내포의 일반적인 문법은 다음과 같다.

[표현식 for 항목 in 반복가능객체 if 조건]

위 문법에서 조건식 부분은 위 예제에서 보듯이 생략이 가능하다.

조금 복잡하지만 for문을 2개이상 사용하는 것도 가능하다. for문을 여러개 사용할 경우의 문법은 다음과 같다.

---

```
[표현식 for 항목1 in 반복가능객체1 if 조건1
    for 항목2 in 반복가능객체2 if 조건2
    ...
    for 항목n in 반복가능객체n if 조건n]
```

위에서 살펴본 구구단의 모든 결과를 리스트에 담고 싶다면 리스트 내포를 이용하여 아래와 같이 구현할 수 있을 것이다.

```
>>> result = [x*y for x in range(2,10)
...           for y in range(1,10)]
>>> print(result)
[2, 4, 6, 8, 10, 12, 14, 16, 18, 3, 6, 9, 12, 15, 18, 21, 24, 27, 4, 8, 12,
20, 24, 28, 32, 36, 5, 10, 15, 20, 25, 30, 35, 40, 45, 6, 12, 18, 24, 30, 35,
, 48, 54, 7, 14, 21, 28, 35, 42, 49, 56, 63, 8, 16, 24, 32, 40, 48, 56, 64,
```

## 6. 입출력

지금껏 공부한 내용을 근간으로 하여 이제 함수, 입력과 출력, 파일처리방법 등에 대해서 알아 보기로 하자.

입출력은 프로그래밍 설계와 관련이 있다. 프로그래머는 프로그램을 어떤식으로 동작하게 만들어야겠다는 디자인을 먼저 하게되는데 그 때 가장 중요한 부분이 바로 입출력의 설계이다. 특정 프로그램만 사용하는 함수를 만들 것인지 아니면 모든 프로그램들이 공통으로 사용하는 함수를 만들것인지, 더 나아가 오픈API로 공개하여 모든 외부 프로그램들이 사용할 수 있게끔 만들것인지에 대한 그 모든것들이 전부 이 입출력과 관련이 있다.

## 1) 함수

함수를 설명하기 전에 믹서기를 생각해보자. 우리는 믹서기에 파일을 넣는다. 그리고 믹서를 이용해서 파일을 갈아서 파일 쥬스를 만들어 낸다. 우리가 믹서기에 넣는 파일은 입력이 되고 파일 쥬스는 그 출력(리턴값)이 된다.

그렇다면 믹서기는 무엇인가?



(by <http://www.wpclipart.com>)

바로 우리가 여기서 알고자 하는 함수이다. 입력을 가지고 어떤 일을 수행한 다음에 결과물을 내어놓는 것, 이것이 함수가 하는 일이다. 우리는 어려서부터 함수가 무엇인지에 대해서 공부했지만 이것에 대해서 깊숙이 고민해 본 적은 별로 없다. 하지만 우리는 함수에 대해서 조금 더 생각해 보는 시간을 가져야 한다. 프로그래밍에 있어서 함수란 것은 정말 중요하기 때문이다.

일차함수  $y = 2x + 3$  이것도 함수이다. 하지만 우리는 이것을 수학시간에 배웠던 직선그래프로만 알고 있지  $x$ 에 어떤 값을 넣었을 때 어떤 변화에 의해서  $y$  값이 나온다는 생각은 대부분 해보지 않았을 것이다.

생각을 달리하고 파이썬 함수에 대해서 깊이 들어가 보도록 하자.

## 함수를 사용하는 이유?

가끔 프로그래밍을 하다 보면 똑같은 내용을 자신이 반복해서 적고 있는 것을 발견할 때가 있다. 이 때가 바로 함수가 필요한 때이다. 여러 번 반복해서 사용된다는 것은 언제고 또다시 사용할 만한 가치가 있는 부분이라는 뜻이다. 즉, 이러한 경우 이것을 한 뭉치로 묶어서 “어떤 입력값을 주었을 때 어떤 리턴값을 돌려준다”라는 식의 함수를 작성하는 것이 현명한 일일 것이다.

함수를 사용하는 또 다른 이유로는 자신이 만든 프로그램을 함수화 시켜 놓으면 프로그램의 흐름을 일목요연하게 감지할 수 있게 된다. 마치 큰 공장에서 한 물건이 나올 때 여러 가지 공정을 거쳐가는 것처럼 자신이 원하는 결과값이 함수들을 거쳐가면서 변화되는 모습을 볼 수 있다.

이렇게 되면 프로그램의 흐름도 잘 파악할 수 있게 되고 예러가 어디에서 나는지도 금방 알아차릴 수 있게 된다.

## 파이썬 함수의 구조

파이썬 함수의 모습은 다음과 같다.

```
def 함수명(입력 인수):
    <수행할 문장1>
    <수행할 문장2>
    ...
    ...
```

`def`는 함수를 만들 때 사용하는 예약어이다. 함수명은 함수를 만드는 사람이 임의로 만드는 것이다. 마치 변수이름을 정하는 것과 같은 이치이다. 함수명 다음에 괄호 안에 있는 입력인수라는 것은 이 함수에 입력으로 넣어주는 값이다. 다음에 `if`, `while`, `for`문 등과 마찬가지로 수행할 문장을 수행한다.

가장 간단하지만 많은 것을 설명해 주는 다음의 예를 보도록 하자.

```
def sum(a, b):
    return a + b
```

위 함수의 의미는 다음과 같이 정의된다.

“*sum*이라는 것은 함수명이고 입력값으로 두개의 값을 받으며 리턴값은 두 개의 입력값을 더한 값이다.”

여기서 `return`은 함수의 결과 값을 돌려주는 명령어이다. 직접 위의 함수를 만들어 보고 사용해 보자.

```
>>> def sum(a, b):
...     return a+b
...
>>>
```

위와 같이 `sum`함수를 먼저 만들자.

```
>>> a = 3
>>> b = 4
>>> c = sum(a, b)
>>> print(c)
7
```

`a`에 3, `b`에 4를 대입한 다음 이미 만들었던 `sum`함수에 `a`와 `b`를 입력값으로 주어서 `c`라는 함수의 리턴값을 돌려 받는다. `print(c)`로 `c`의 값을 확인할 수 있다.

## 함수의 입력값과 리턴값

프로그래밍을 공부할 때 어려운 부분 중 하나가 용어의 혼용이라고 할 수 있다. 많은 원서들을 보기도 하고 누군가의 번역본을 보기도 하면서 우리는 갖가지 용어들을 익힌다. 입력 값을 다른 말로 함수의 인수, 입력인수 등으로 말하기도 하고 리턴 값을 출력 값, 결과 값, 돌려주는 값 등으로 말하기도 한다. 이렇듯 많은 용어들이 다른 말로 표현되지만 의미는 동일한 경우가 많다. 이런 것들에 대해 기억해 놓아야만 머리가 덜 아플 것이다.

함수는 들어온 입력값을 가지고 어떤 처리를 하여 적절한 리턴값을 돌려주는 블랙박스와 같다.

입력값 ---> 함수(블랙박스) ----> 리턴값

함수에 들어오는 입력값과 리턴값에 대해서 자세히 알아보도록 하자.

### 평범한 함수

입력 값이 있고 리턴값이 있는 함수가 평범한 함수이다. 앞으로 독자가 프로그래밍을 할 때 만들 함수의 대부분은 이러한 형태일 것이다.

```
def 함수이름(입력인수):
    <수행할 문장>
    ...
    return 결과값
```

평범한 함수의 전형적인 예를 한번 보도록 하자.

```
def sum(a, b):
```

```
result = a + b  
return result
```

두 개의 입력값을 받아서 서로 더한 결과값을 돌려주는 함수이다.

위의 함수를 사용하는 방법은 다음과 같다. 다음을 따라 해 보자. 우선 sum 함수를 만들자.

```
>>> def sum(a, b):  
...     result = a + b  
...     return result  
...  
>>>
```

다음에 입력값을 주고 리턴값을 돌려 받아 보자.

```
>>> a = sum(3, 4)  
>>> print(a)  
7
```

위처럼 입력값과 리턴값이 있는 함수는 다음처럼 사용된다.

리턴값받을변수 = 함수명(입력인수1, 입력인수2, , ,)

### 입력값이 없는 함수

입력값이 없는 함수가 존재할까? 당연히 그렇다. 다음을 보자.

```
>>> def say():
...     return 'Hi'
...
>>>
```

say라는 이름의 함수를 만들었다. 하지만 함수이름 다음의 입력 인수부분을 나타내는 괄호 안이 비어있다.

이 함수를 어떻게 쓸 수 있을까? 다음과 같이 따라해 보자.

```
>>> a = say()
>>> print(a)
Hi
```

위의 함수를 쓰기 위해서는 say()처럼 괄호 안에 아무런 값도 넣어주지 않고 써야 한다. 위의 함수는 입력값은 없지만 리턴값으로 'Hi'라는 문자열을 돌려준다. 따라서 a = say()처럼 하면 a에는 'Hi'라는 문자열이 대입되게 되는 것이다.

즉, 입력값이 없고 리턴값만 있는 함수는 다음과 같이 사용된다.

리턴값받을변수 = 함수명()

## 리턴값이 없는 함수

리턴값이 없는 함수 역시 존재한다. 다음의 예를 보자.

```
>>> def sum(a, b):
...     print("%d, %d의 합은 %d입니다." % (a, b, a+b))
...
```

---

```
>>>
```

리턴값이 없는 함수는 돌려주는 값이 없기 때문에 다음과 같이 사용한다.

```
>>> sum(3, 4)
3, 4의 합은 7입니다.
```

즉, 리턴값이 없는 함수는 다음과 같이 사용된다.

함수명(입력인수1, 입력인수2, , ,)

실제로 리턴값이 없는지 알아보기 위해 다음의 예를 따라해 보자.

```
>>> a = sum(3, 4)
3, 4의 합은 7입니다.
```

아마도 독자는 다음과 같은 질문을 할지도 모른다. “3, 4의 합은 7입니다.”라는 문장을 출력해 주었는데 리턴값이 없는 것인가? 이 부분이 초보자들이 혼동스러워 하는 부분이기도 한데 print 문은 함수내에서 사용되어지는 문장일 뿐이다. 돌려주는 값은 당연히 없다. 돌려주는 값은 return 명령어로만 가능하다.

이것을 확인해 보기 위해서 돌려받는 값을 a라는 변수에 대입하여 출력해 보면 리턴값이 있는지 알 수 있다.

```
>>> print(a)
None
```

a의 값이 None이다. 이 None이란 것은 거짓을 나타내는 자료형이라고 언급한 적이 있었다. sum함수처럼 리턴 값이 없을 때 a = sum(3, 4)처럼 쓰게 되면

함수 sum은 리턴 값으로 a변수에 None을 돌려주게 된다. 그렇다고 이것이 리턴 값이 있다는 걸로 생각하면 곤란하다.

### 입력값도 리턴값도 없는 함수

이것 역시 존재한다. 다음의 예를 보자.

```
>>> def say():
...     print('Hi')
...
>>>
```

입력 값을 받는 곳도 없고 return문도 없으니 입력값도 리턴값도 없는 함수이다.

이것을 사용하는 방법은 단 한가지이다.

```
>>> say()
Hi
```

즉, 입력값도 리턴값도 없는 함수는 다음과 같이 사용된다.

함수명()

### 입력값이 몇 개가 될 지 모를 때는 어떻게 해야 할까?

입력값을 주었을 때 그 입력값들을 모두 더해주는 함수를 생각해 보자. 하지만 몇 개가 입력으로 들어올지 알 수 없을 때는 어떻게 해야 할지 난감할 것이다. 이에 파이썬에서는 다음과 같은 것을 제공한다.

---

```
def 함수이름(*입력변수):
```

<수행할 문장>

...

입력인수 부분이 \*+입력변수로 바뀌었다.

다음의 예를 통해 사용법을 알아보자. sum\_many(1, 2) 하면 3을 sum\_many(1,2,3)이라고 하면 6을 sum\_many(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)은 55를 돌려주는 함수를 만들어 보자.

```
>>> def sum_many(*args):
...     sum = 0
...     for i in args:
...         sum = sum + i
...     return sum
...
>>>
```

위에서 만든 sum\_many라는 함수는 입력값이 몇 개든지 상관이 없다. 그 이유는 args라는 변수가 입력값들을 전부 모아서 튜플로 만들어 주기 때문이다. 만약 sum\_many(1, 2, 3)처럼 이 함수를 쓴다면 args는 (1, 2, 3)이 되고 sum\_many(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)처럼 하면 args는 (1, 2, 3, 4, 5, 6, 7, 8, 9, 10)이 된다. 여기서 \*args라는 것은 임의로 정한 변수명이다. \*pey, \*python처럼 아무이름으로 해도 된다. 단 args라는 것은 입력인수를 뜻하는 영어단어인 arguments라는 영어의 약자로 관례적인 표기법임을 알아두도록 하자.

실제로 한번 테스트를 해 보도록 하자.

```
>>> result = sum_many(1,2,3)
>>> print(result)
```

```
6
>>> result = sum_many(1,2,3,4,5,6,7,8,9,10)
>>> print(result)
55
```

def sum\_many(\*args) 처럼 \*args를 사용할 경우 \*args만 입력 인수로 올 수 있는 것은 아니다. 다음의 예를 보도록 하자.

```
>>> def sum_mul(choice, *args):
...     if choice == "sum":
...         result = 0
...         for i in args:
...             result = result + i
...     elif choice == "mul":
...         result = 1
...         for i in args:
...             result = result * i
...     return result
...
>>>
```

위의 예는 입력 인수로 choice와 \*args라는 것을 받는다. 따라서 다음과 같이 쓸 수 있을 것이다. sum\_mul('sum', 1,2,3,4) 또는 sum\_mul('mul', 1,2,3,4,5) 처럼 choice부분에는 'sum'이나 'mul'이라는 문자열을 그리고 그 뒤에는 개수에 상관없는 숫자를 입력으로 준다.

실제로 한번 테스트를 해 보도록 하자.

```
>>> result = sum_mul('sum', 1,2,3,4,5)
```

---

```
>>> print(result)
15
>>> result = sum_mul('mul', 1,2,3,4,5)
>>> print(result)
120
```

함수의 리턴값은 언제나 하나이다.

먼저 다음의 함수를 만들어 보자.

```
>>> def sum_and_mul(a,b):
...     return a+b, a*b
```

이것을 다음과 같이 호출하면 어떻게 될까?

```
>>> a = sum_and_mul(3,4)
```

리턴값은  $a+b$ 와  $a*b$  두 개인데 리턴값을 받아들이는 변수는  $a$  하나만 쓰였으니  
에러가 나지 않을까? 당연한 의문이다. 하지만 에러는 나지 않는다. 그 이유는  
리턴값이 두 개가 아니라 하나라는 데 있다. 함수의 리턴값은 튜플값 하나로 돌  
려주게 된다.

즉  $a$  변수는 위의 `sum_and_mul` 함수에 의해서 다음과 같은 값을 가지게 된다.

```
a = (7, 12)
```

즉, 리턴값으로  $(7, 12)$ 라는 튜플 값을 갖게 되는 것이다. 이것을 두 개의 리턴  
값처럼 받고 싶다면 다음과 같이 호출하면 될 것이다.

```
>>> sum, mul = sum_and_mul(3, 4)
```

이렇게 호출한다면 이것의 의미는  $\text{sum}, \text{mul} = (7, 12)$ 가 되어서  $\text{sum}$ 은 7이 되고  $\text{mul}$ 은 12가 될 것이다.

또, 다음과 같은 의문이 생길 수도 있다.

```
>>> def sum_and_mul(a,b):  
...     return a+b  
...     return a*b  
...  
>>>
```

위와 같이 하면 두 개의 리턴값을 돌려주지 않을까? 하지만 파이썬에서 위와 같은 함수는 참 어리석은 함수이다.

위의 함수는 다음과 완전히 동일하다.

```
>>> def sum_and_mul(a,b):  
...     return a+b  
...  
>>>
```

즉, 함수는 `return`문을 만나는 순간 `return`값을 돌려준 다음에 함수를 빠져나가게 된다.

## return의 또 다른 쓰임새

특별한 경우에 함수를 빠져나가기를 원할 때 return만 단독으로 써서 함수를 즉시 빠져나갈 수 있다. 다음 예를 보자.

```
>>> def say_nick(nick):
...     if nick == "바보":
...         return
...     print("나의 별명은 %s 입니다." % nick)
...
>>>
```

위의 함수는 입력값으로 nick이란 변수를 받아서 문자열을 출력하는 함수이다. 이 함수 역시 리턴값은 없다. (문자열을 출력한다는 것과 리턴값이 있다는 것은 전혀 다른 말이다. 혼동하지 말도록 하자, 함수의 리턴값은 오로지 return문에 의해서만 생성된다.) 만약에 입력값으로 '바보'라는 값이 들어오면 문자열을 출력하지 않고 함수를 즉시 빠져나간다. return문으로 함수를 빠져나가는 방법은 실제 프로그래밍에서 자주 쓰인다.

## 입력값에 초기치 설정하기

이젠 좀 더 다른 함수의 인수 전달 방법에 대해서 알아보자. 인수에 초기치를 미리 설정해 주는 경우를 보자.

```
def say_myself(name, old, man=True):
    print("나의 이름은 %s 입니다." % name)
    print("나이는 %d살입니다." % old)
    if man:
        print("남자입니다.")
```

```
else:  
    print("여자입니다.")
```

(※ 알아두기 - 앞으로 나올 프로그램 소스는 '>>>' 표시가 없으면 항상 에디터로 작성하기로 하자.)

위의 함수를 보면 입력 인수가 세 개임을 알수 있다. 그런데 이상한 점이 나왔다. man=True처럼 입력 인수에 미리 값을 넣어준 것이다. 이것이 바로 함수의 인수 초기치를 설정하는 방법이다. 항상 변하는 것이 아니라 아주 가끔 변하는 것일 때 이렇게 함수의 초기치를 미리 설정해 주는 것은 매우 유용하다.

위의 함수는 다음처럼 사용할 수 있다.

```
say_myself("박응용", 27)  
say_myself("박응용", 27, True)
```

즉 인수값으로 “박응용”, 27처럼 두 개를 주게 되면 name에는 “박응용”이 old에는 27이 man이라는 변수에는 입력값을 주지 않았지만 초기값인 True라는 값을 갖게 되는 것이다. 따라서 위에서 함수를 사용한 두가지 방법은 모두 동일한 결과를 출력하게 된다.

```
say_myself("박응선", 27, False)
```

초기값설정된 부분을 False로 바꾸었으므로 이번에는 man 변수에 False라는 값이 대입되게 된다.

### 함수 입력값 초기치 설정시 주의사항

함수의 초기치를 설정할 때 주의해야 할 것이 하나 있다.

만약 위의 함수를 다음과 같이 만들면 어떻게 될까?

```
def say_myself(name, man=True, old):
    print("나의 이름은 %s 입니다." % name)
    print("나이는 %d살입니다." % old)
    if man:
        print("남자입니다.")
    else:
        print("여자입니다.")
```

위의 함수와 바뀐 부분은 초기치를 설정한 인수의 위치이다. 결론을 미리 말하자면 이것은 실행시에 에러가 난다.

그냥 얼핏 생각하기에 위의 함수를 호출하려면 다음과 같이 하면 될 것 같다.

```
say_myself("박응용", 27)
```

하지만 위와 같이 함수를 호출한다면 name변수에는 “박응용”이 들어가겠지만 파이썬 인터프리터는 man에 27을 대입해야 할지 old에 27을 대입해야 할지 알 수 없을 것이다.

에러메시지를 보면 다음과 같다.

```
SyntaxError: non-default argument follows default argument
```

위의 에러메시지는 초기 치를 설정해 놓은 입력 인수 뒤에 초기 치를 설정해 놓지 않은 입력 인수는 사용할 수 없다는 말이다. 즉 입력인수로 (name, old, man=True)은 되지만 (name, man=True, old)은 안된다는 것이다. 결론은 초기화 시키고 싶은 입력 변수들은 항상 뒤쪽에 위치시키라는 것이다.

## 함수 내에서 선언된 변수의 효력 범위

함수안에서 사용할 변수의 이름을 함수 밖에서 사용한 이름과 동일하게 사용한다면 어떻게 될까? 이런 궁금증이 떠올랐던 독자라면 이곳에서 확실하게 알 수 있을 것이다.

아래의 예를 보자.

```
a = 1
def vartest(a):
    a = a +1

vartest(a)
print(a)
```

먼저 a라는 변수를 생성하고 1을 대입한다. 그리고 입력으로 들어온 값을 1만큼 더해주고 리턴값은 돌려주지 않는 vartest 함수에 입력 값으로 a를 주었다. 그 다음에 a의 값을 출력하게 하였다. 당연히 vartest에서 a를 1만큼 더했으니까 2가 출력되어야 할 것 같지만 프로그램 소스를 작성해서 실행시켜 보면 결과 값이 1이 나온다.

그 이유는 함수안에서 새로 만들어진 변수는 함수안에서만 쓰여지는 함수만의 변수이기 때문이다. 즉 def vartest(a)처럼 하면 이때 입력 인수를 뜻하는 변수 a는 함수안에서만 쓰이는 변수이지 함수 밖의 변수 a가 아니라는 말이다.

위에서 변수이름을 a로 한 vartest함수는 다음처럼 변수이름을 b로 한 vartest와 완전히 동일한 것이다.

```
def vartest(b):
    b = b + 1
```

즉 함수에서 쓰이는 변수는 함수 밖의 변수이름들과는 전혀 상관 없다는 말이다. 따라서 위에서 보았던 vartest의 a는 1이란 값을 입력으로 받아서 1만큼 더해주어 a가 2가 되지만 그것은 함수 내의 a를 뜻하는 것이지 함수 밖의 변수 a와는 전혀 다르다는 말이다.

다음의 예를 보면 더욱 정확하게 이해할 수 있을 것이다.

```
def vartest(a):
    a = a + 1

vartest(3)
print(a)
```

위의 프로그램 소스를 에디터로 작성해서 실행시키면 어떻게 될까? 여러가 날 것이라 생각한 독자는 모든 것을 이해한 독자이다. vartest(3)을 수행하면 vartest라는 함수내에서 a는 4가 되지만 함수를 끝내고 난뒤에 print(a)라는 문장은 여러가 나게 된다. 그 이유는 a라는 변수는 어디에도 찾아 볼 수가 없기 때문이다. 다시 말하지만 함수 내에서 쓰이는 변수는 함수내에서 쓰일 뿐이지 함수 밖에서 사용될 수 있는 것이 아니다. 이것에 대해서 이해하는 것은 매우 중요하다.

그렇다면 vartest라는 함수를 이용해서 함수 외부의 a를 1만큼 증가시킬 수 있는 방법은 없을까?라는 의문이 떠오르게 된다. 두 가지 해결 방법이 있을 수 있다.

### 첫 번째 방법

```
a = 1
def vartest(a):
    a = a +1
    return a
```

```
a = vartest(a)
print(a)
```

첫 번째 방법은 return을 이용하는 방법이다. vartest함수는 입력으로 들어온 값을 1만큼 더한 값을 돌려준다. 따라서 a = vartest(a)처럼 하면 a가 vartest 함수의 리턴값으로 바뀌게 되는 것이다. 여기서도 물론 vartest함수 안의 a변수는 함수 밖의 a와는 다른 것이다.

## 두 번째 방법

```
a = 1
def vartest():
    global a
    a = a+1

vartest()
print(a)
```

두 번째 방법은 global이라는 명령어를 이용하는 방법이다. 위의 예에서 보듯이 vartest안의 global a라는 문장은 함수 안에서 함수 밖의 a변수를 직접 사용하겠다는 말이다. 보통 프로그래밍을 할 때 global이라는 것은 쓰지 않는 것이 좋다. 왜냐하면 함수는 독립적으로 존재하는 것이 좋기 때문이다. 외부 변수에 종속적인 함수는 그다지 좋은 함수가 아니다. 독자는 가급적 이 global을 쓰는 방식을 피해야 할 것이다. 따라서 당연히 두 번째 방법보다는 첫 번째 방법이 좋다.

## 2) 입력과 출력

우리들이 사용하는 대부분의 완성된 프로그램은 사용자의 입력에 따라서 그에 맞는 출력값을 내어주게끔 되어 있다. 예를 들어보면 게시판에 우리가 작성한 글을 입력한다는 “확인” 버튼을 눌러야만(입력) 우리가 작성한 글이 게시판에 올라가는 것(출력)을 확인 할 수 있게 되는 것이다.

사용자 입력 ---> 처리(프로그램, 함수 등) ---> 출력

우리는 이미 함수 부분에서 입력과 출력이 어떤 의미인지에 대해서 알아보았다. 지금부터는 좀 더 다양하게 사용자의 입력을 받는 방법과 파일을 읽고 쓰는 방법 등에 대해서 알아보도록 하자.

### 사용자 입력

어떤 변수에 사용자로부터 입력받은 값을 대입하고 싶을 때는 어떻게 해야 할까?

#### input의 사용

```
>>> a = input()
Life is too short, you need python
>>> a
Life is too short, you need python
>>>
```

input은 입력되는 모든 것을 문자열로 취급한다.

※ 파이썬 2.7 버전의 경우 위 예제의 `input` 대신 `raw_input`을 사용해야 한다. (참고: [파이썬 2.7 vs 파이썬 3](#))

## 프롬프트 추가하기

`input(prompt)`

사용자로부터 입력을 받을 때 “숫자를 입력하세요.” 라던지 “이름을 입력하세요”라는 질문을 포함하고 싶을 때도 있다. 그럴 때는 `prompt`를 이용하면 된다. 다음의 예를 따라해 보자.

```
>>> number = input("숫자를 입력하세요: ")  
숫자를 입력하세요:
```

위와 같은 질문을 볼 수 있을 것이다.

## print 자세히 알기

우리가 앞서서 계속 사용해 왔던 `print`가 하는 일은 자료형을 출력하는 것이다. 지금껏 알아보았던 `print`의 사용예는 다음과 같다.

```
>>> a = 123  
>>> print(a)  
123  
>>> a = "Python"  
>>> print(a)  
Python  
>>> a = [1, 2, 3]
```

---

```
>>> print(a)
[1, 2, 3]
```

이제 이것보다 조금 더 자세하게 print에 대해서 알아 보기로 하자.

### 따옴표(“)로 둘러싸인 문자열은 + 연산과 동일

```
>>> print("life" "is" "too short") ----- ①
lifeistoo short
>>> print("life"+"is"+"too short") ----- ②
lifeistoo short
```

위에서 ①과 ②는 완전히 동일한 결과값을 보여준다. 즉, 따옴표로 둘러싸인 문자열을 연속해서 쓰면 '+'연산을 한 것과 마찬가지이다.

### 문자열 띄어쓰기는 콤마로

```
>>> print("life", "is", "too short")
life is too short
```

콤마(,)를 이용하면 문자열간에 띄어쓰기가 된다.

### 한 줄에 출력하기

앞서 보았던 구구단 프로그램에서 보았듯이 한 줄에 결과값을 계속 출력하려면 end파라미터를 이용하여 끝문자를 지정해야 한다.

```
>>> for i in range(10):
...     print(i, end=' ')
...
0 1 2 3 4 5 6 7 8 9
```

※ 파이썬 2.7 버전의 경우 위 예제의 `print(i, end=' ')` 대신 `print i,`로 사용해야 한다. (참고: [파이썬 2.7 vs 파이썬 3](#))

### 3) 파일 읽고 쓰기

우리는 지금껏 입력으로는 사용자가 입력하게 하는 방식을 사용하였고, 출력으로는 모니터 화면에 결과값을 출력하는 방식의 프로그래밍만 해 왔다. 하지만 입출력의 방법이 꼭 그것만 있는 것은 아니다. 이번에는 파일을 이용한 입출력 방법에 대해서 알아 볼 것이다. 먼저 파일을 새롭게 만들어서 프로그램에 의해 만들어진 결과 값을 파일에 한번 적어보고, 또 적은 내용을 읽어보는 프로그램을 만드는 것으로 시작해 보자.

#### 파일 생성하기

다음을 에디터로 작성해서 실행해 보면 프로그램을 실행한 디렉토리에 새로운 파일이 하나 생성되는 것을 확인할 수 있을 것이다.

```
f = open("새파일.txt", 'w')
f.close()
```

파일을 생성하기 위해서 우리는 open이란 파일 내장 함수를 썼다.

open함수는 다음과 같이 파일이름과 파일열기모드를 입력으로 받고 리턴값으로 파일 객체를 돌려준다.

```
파일객체 = open(파일이름, 파일열기모드)
```

파일 열기 모드에는 다음과 같은 것들이 있다.

##### 파일열기모드 설명

r	읽기모드 - 파일을 읽기만 할 때 사용
---	-----------------------

---

## 파일열기모드 설명

---

w	쓰기모드 - 파일에 내용을 쓸 때 사용
a	추가모드 - 파일의 마지막에 새로운 내용을 추가 시킬 때 사용

---

파일을 쓰기 모드로 열게 되면 해당 파일이 존재할 경우에는 원래있던 내용이 모두 사라지게 되고, 해당 파일이 존재하지 않으면 새로운 파일이 생성된다. 위의 예에서는 없는 파일인 “새파일.txt”를 쓰기 모드로 열었기 때문에 새로운 파일인 “새파일.txt”라는 이름의 파일이 현재디렉토리에 생성되는 것이다.

만약 ”새파일.txt“라는 파일을 C:/Python이란 디렉토리에 생성하고 싶다면 다음과 같이 해야 할 것이다.

```
f = open("C:/Python/새파일.txt", 'w')  
f.close()
```

위의 예에서보면 f.close()라는 것이 있는데 이것은 열린 파일 객체를 닫아주는 것이다. 사실 이 문장은 생략해도 된다. 왜냐하면 파이썬 프로그램이 종료할 때 열린 파일 객체를 자동으로 닫아주기 때문이다. 하지만 직접 열린 파일을 닫아주는 것이 좋다. 쓰기모드로 열었던 파일을 닫지 않고 다시 사용하려고 할 경우에는 에러가 나기 때문이다.

## 파일을 쓰기 모드로 열어서 출력값 적기

위의 예에서는 파일을 쓰기모드로 열기만 했지 정작 아무런 것도 쓰지 않았다. 이번에는 프로그램의 출력값을 파일에 적어 보도록 하자.

```
f = open("새파일.txt", 'w')
```

```
for i in range(1, 11):
    data = "%d 번째 줄입니다.\n" % i
    f.write(data)

f.close()
```

위의 프로그램을 다음의 프로그램과 비교해 보자.

```
for i in range(1, 11):
    data = "%d 번째 줄입니다.\n" % i
    print(data)
```

두 프로그램의 서로 다른 점은 data를 출력시키는 방법이다. 두 번째 방법은 우리가 지금껏 계속 사용해 왔던 모니터 화면에 출력하는 방법이고 첫 번째 방법은 모니터 화면대신에 파일에 결과값을 적는 방법이다. 차이점이 금방 눈에 들어 올 것이다. 두번째 방법의 print대신에 파일객체 f의 write라는 함수를 이용했을 뿐이다. 첫 번째 예제를 에디터로 작성해서 실행시킨 다음 그 프로그램을 실행시킨 디렉토리를 살펴보면 ”새파일.txt“라는 파일이 생성되었음을 확인 할 수 있을 것이다. 그 파일이 어떤 내용을 담고 있는지 확인해 보자. 아마 다음과 같을 것이다.

새파일.txt의 내용

- 1 번째 줄입니다.
- 2 번째 줄입니다.
- 3 번째 줄입니다.
- 4 번째 줄입니다.
- 5 번째 줄입니다.

- 6 번째 줄입니다.
- 7 번째 줄입니다.
- 8 번째 줄입니다.
- 9 번째 줄입니다.
- 10 번째 줄입니다.

즉 두 번째 방법을 사용했을 때 모니터 화면에 출력될 내용이 파일에 고스란히 들어가 있는 것을 알 수 있다.

## 파일을 읽는 여러가지 방법

파이썬에는 파일을 읽는 여러가지 방법이 있다. 이번에는 그것들에 대해서 자세히 알아보도록 하자.

### 첫 번째 방법

첫 번째 방법은 `readline()`을 이용하는 방법이다. 다음의 예를 보자.

```
f = open("새파일.txt", 'r')
line = f.readline()
print(line)
f.close()
```

이전에 만들었던 “새파일.txt”를 수정하거나 지우지 않았다면 위의 프로그램을 실행시켰을 때 “새파일.txt”의 가장 첫 번째 줄을 읽어서 화면에 출력해 줄 것이다. 위 예제는 `f.open("새파일.txt", 'r')`로 파일을 읽기 모드로 연 후 `readline()`을 이용해서 파일의 첫 번째 줄을 읽어 출력하는 예제이다.

만약 모든 라인을 읽어서 화면에 출력하고 싶다면 다음과 같은 프로그램을 작성해야 한다.

```
f = open("새파일.txt", 'r')

while True:
    line = f.readline()
    if not line: break
    print(line)

f.close()
```

while True라는 무한루프를 이용해서 f.readline()을 이용해서 파일을 계속해서 한 줄씩 읽어 들인다. 만약 더 이상 읽을 라인이 없으면 break가 수행되어 while 문을 빠져나간다. (f.readline()은 더 이상 읽을 라인이 없을 경우 None을 리턴한다.)

위의 프로그램을 다음 프로그램과 비교해 보자.

```
while 1:
    data = input()
    if not data: break
    print(data)
```

위의 예는 사용자의 입력을 받아서 그 내용을 출력하는 예이다. 위의 파일을 읽어서 출력하는 예제와 비교해 보자. 입력을 받는 방식만 틀리다는 것을 금방 알 수 있을 것이다. 두 번째 예는 키보드를 통한 입력방법이고 첫 번째 방법은 파일을 이용한 입력 방법이다.

## 두 번째 방법

두 번째 방법은 `readlines()`를 이용하는 방법이다. 다음의 예를 보기로 하자.

```
f = open("새파일.txt", 'r')
lines = f.readlines()

for line in lines:
    print(line)

f.close()
```

`f.readlines()`는 파일의 모든 라인을 읽어서 각각의 줄을 요소로 갖는 리스트를 리턴한다. 따라서 위의 예에서 `lines`는 [“1 번째 줄입니다.”, “2 번째 줄입니다.”, , , “10 번째 줄입니다.”]라는 리스트가 된다. `f.readlines()`는 `f.readline()`과는 달리 's'가 하나 더 붙어 있음에 유의하도록 하자.

## 세 번째 방법

세 번째 방법은 `read()`를 이용하는 방법이다. 다음의 예를 보기로 하자.

```
f = open("새파일.txt", 'r')
data = f.read()

print(data)

f.close()
```

f.read()는 파일의 내용 전체를 문자열로 리턴한다. 따라서 위의 예의 data는 파일의 전체내용이다.

## 파일에 새로운 내용 추가하기

'w' 모드로 파일을 연 경우에는 이미 존재하는 파일을 열 경우 그 파일의 내용이 모두 사라지게 된다고 했는데 원래 있던 값을 유지하면서 단지 새로운 값만 추가해야 할 경우도 있다. 이런 경우에는 파일을 추가 모드('a')로 열면 된다. 다음의 예를 보도록 하자.

```
f = open("새파일.txt", 'a')

for i in range(11, 20):
    data = "%d번째 줄입니다.\n" % i
    f.write(data)

f.close()
```

위 예는 “새파일.txt”라는 파일을 추가모드('a')로 열고 write를 이용해서 결과값을 파일에 적는다. 여기서 추가 모드로 파일을 열었기 때문에 ”새파일.txt“라는 파일이 원래 가지고 있던 내용 바로 다음부터 결과값을 적기 시작한다. “새파일.txt”라는 파일을 읽어서 확인해 보면 원래있던 파일 뒷부분에 새로운 부분이 추가 되었음을 확인 할 수 있을 것이다.

## with문과 함께 사용하기

지금까지 진행했던 예제를 보면 항상 다음과 같은 스타일로 파일을 열고 닫아 왔다.

```
f = open("foo.txt", 'w')
f.write("Life is too short, you need python")
f.close()
```

파일을 열면 위와 같이 항상 close해 주는 것이 좋은 스타일이다. 하지만 이렇게 파일을 열고 닫는 것을 자동으로 해 줄 수 있다면 편리하지 않겠는가?

파이썬의 with문이 바로 이런 역할을 해 준다.

다음의 예는 with문을 이용해서 위 예제를 다시 작성 한 모습이다.

```
with open("foo.txt", "w") as f:
    f.write("Life is too short, you need python")
```

위와 같이 with문을 이용하면 with 블록을 벗어나는 순간 열린 파일객체 f가 자동으로 close되게 되어 편리하다.

※ with문은 파일에만 쓰이는 것은 아니다. 보통 파일, 락, 연결등의 시스템 자원을 연결하고 해제할 경우 함께 사용된다.

## sys모듈 입력

이전에 도스를 사용해 본 독자라면 다음과 같은 명령어를 사용해 본적이 있을 것이다.

```
C:\> type a.txt
```

위의 type명령어는 뒤에 파일이름을 인수로 받아서 그 내용을 출력해 주는 도스 명령어이다.

## 도스명령어 [인수1 인수2]

많은 도스 명령어가 위와 같은 방식을 따른다. 즉 명령행(도스창)에서 입력인수를 직접 주어서 프로그램을 실행시키는 방식이다. 이러한 기능을 파이썬 프로그램에도 적용시킬 수가 있다.

파이썬에서는 sys란 모듈을 이용하여 이것을 가능하게 한다. sys라는 모듈을 쓰려면 아래의 예에서 같이 import sys처럼 import라는 명령어를 사용해야 한다. 모듈을 사용하고 만드는 방법에 대해서는 뒤에서 자세히 다룰 것이다.

```
#sys1.py
import sys

args = sys.argv[1:]
for i in args:
    print(i)
```

위의 프로그램을 C:/Python이란 디렉토리에 저장하고 윈도우즈 도스창을 열고 다음과 같이 입력해 보자.

```
C:/Python>c:/python34/python sys1.py aaa bbb ccc
```

다음과 같은 결과 값을 볼 수 있을 것이다.

결과값:

```
aaa
bbb
ccc
```

sys모듈의 argv는 명령창에서 입력한 인수들을 의미한다. 즉, argv[0]는 파일 이름인 sys1.py가 되고 argv[1]부터는 뒤에 따라오는 인수들이 차례로 argv의 요소가 된다. 위의 예는 입력받은 인수들을 for문을 이용해 차례대로 하나씩 출력하는 예이다.

위의 예를 이용해서 간단한 스크립트를 하나 만들어 보자.

```
# sys2.py
import sys
args = sys.argv[1:]
for i in args:
    print(i.upper(), end=' ')
```

문자열 관련함수인 upper()를 이용한 명령행에서 입력된 소문자를 대문자로 바꾸어 주는 간단한 프로그램이다. 도스창에서 다음과 같이 입력해 보자. (주의: sys2.py 파일이 C:/Python이란 디렉토리내에 있어야만 한다.)

```
C:/Python> c:\python34\python sys2.py life is too short, you need python
```

결과값:

```
LIFE IS TOO SHORT, YOU NEED PYTHON
```

## 7. 파이썬 날개달기

이제 프로그래밍의 꽃이라고 할 수 있는 클래스에 대해서 알아보고 모듈, 예외처리 및 파이썬 라이브러리에 대해서 알아보자. 이번장의 내용을 끝으로 여러분은 파이썬 프로그램을 작성하기 위해 배워야 할 대부분의 것들을 숙지했다고 보면 될 것이다.

## 1) 클래스

초보 개발자에게 클래스(class)는 넘기 힘든 장벽과도 같은 존재이다. 여기서는 최대한 쉬운것부터 차근차근 가 보도록 하자.

### 클래스는 도대체 왜 필요하게 되었을까?

가장 많이 사용되는 언어중 하나인 C언어에는 클래스가 없다. 이 말은 굳이 클래스 없이도 프로그램을 충분히 만들 수 있다는 말과도 같다. 하지만 요새 새로이 등장하는 언어들은 모두 클래스를 포함하고 있다.

그렇다면 도대체 이 클래스라는 것은 왜 필요하게 되었을까?

예제를 통해 한번 생각 해 보자.

여러분은 계산기를 사용해 봤을 것이다. 3이라는 숫자를 계산기에 입력한 후 '+' 기호를 입력한 후 4를 입력하면 결과값으로 7을 보여준다. 다시 한번 '+' 기호를 입력한 후 3을 입력하면 기존 결과값 7에 3을 더해 10을 보여준다. 즉, 계산기는 이전에 계산된 결과값을 항상 메모리 어딘가에 저장하고 있어야 한다.

이러한 내용을 우리가 먼저 익힌 함수를 이용하여 구현 해 보자. 계산기의 “더하기” 기능을 구현한 파이썬 코드는 다음과 같다:

```
result = 0

def adder(num):
    global result
    result += num
    return result

print(adder(3))
```



```
print(adder(4))
```

이전에 계산된 결과값을 유지하기 위해서 result라는 전역변수(global)를 사용했다.

실행하면 예상한대로 다음과 같은 결과값이 출력된다.

3

7

그런데 다음과 같은 문제가 생겼다. 한 프로그램에서 두개의 계산기가 필요한 상황이 되어 버린 것이다. 각각의 계산기는 각각의 결과값을 유지해야 하기 때문에 위와 같이 adder함수 1개 만으로는 결과값을 따로 유지할 수 없게 되었다.

이런 상황을 해결하려면 다음과 같이 함수를 각각 따로 만들어야 한다.

```
result1 = 0
result2 = 0

def adder1(num):
    global result1
    result1 += num
    return result1

def adder2(num):
    global result2
    result2 += num
    return result2

print(adder1(3))
print(adder1(4))
print(adder2(3))
print(adder2(7))
```

adder1과 adder2라는 함수가 만들어졌고 또 각각의 함수의 결과값을 저장하기 위한 전역변수 result1, result2가 필요하게 되었다.

결과 값은 다음과 같이 의도한대로 출력된다.

```
3
7
3
10
```

계산기 1의 결과 값이 계산기 2에 아무런 영향을 끼치지 않음을 확인 할 수

있다.

하지만 계산기가 3개, 5개, 10개로 점점 더 많이 필요하게 된다면 어떻게 할 것인가? 그 때마다 전역변수와 함수를 추가할 것인가?

아직 클래스에 대해서 배우진 않았지만 위와 같은 경우 클래스를 이용하면 다음과 같이 간단하게 해결 할 수 있다. (※ 아래 클래스를 아직은 이해하려고 하지는 말자. 곧 자세하게 배울 것이다. 여기서는 클래스를 개념적으로만 이해했으면 좋겠다.)

```
class Calculator:  
    def __init__(self):  
        self.result = 0  
  
    def adder(self, num):  
        self.result += num  
        return self.result  
  
cal1 = Calculator()  
cal2 = Calculator()  
  
print(cal1.adder(3))  
print(cal1.adder(4))  
print(cal2.adder(3))  
print(cal2.adder(7))
```

실행하면 함수 2개를 사용했을 때와 동일한 다음과 같은 결과가 출력된다.

3  
7

3

10

Calculator 클래스로 만들어진 cal1, cal2라는 인스턴스가 각각의 계산기 역할을 수행하게 된다. 그리고 계산기(cal1, cal2)의 결과값 역시 다른 계산기의 결과값과 상관없이 독립적인 결과값을 유지하게 된다. 클래스를 이용하면 계산기의 갯수가 늘어나더라도 인스턴스를 생성하기만 하면 되기 때문에 함수를 사용하는 경우와 달리 매우 간단 해 진다.

클래스의 이점이 단순히 이것 하나만은 아니지만 도대체 왜 클래스가 필요하게 되었는지에 대한 근본적인 물음에 대한 해답이 되었을 것이다.

이제부터 본격적으로 파이썬 클래스에 대해서 알아보도록 하자.

## 클래스(class) 개념잡기

다음은 파이썬 클래스의 가장 간단한 예이다.

```
class Simple:  
    pass
```

위의 클래스는 아무런 기능도 갖고 있지 않은 껍질 뿐인 클래스이다. 하지만 이렇게 껍질 뿐인 클래스도 인스턴스(instance)라는 것을 생성하는 기능은 가지고 있다. (인스턴스와 객체는 같은 말이다. 클래스에 의해서 생성된 객체를 인스턴스라고 부른다)

인스턴스는 클래스에 의해서 만들어진 객체로 한개의 클래스는 무수히 많은 인스턴스를 만들어 낼 수가 있다. 위에서 만든 Simple 클래스의 인스턴스를 만드는 방법은 다음과 같다.

---

```
a = Simple()
```

바로 Simple()의 결과값을 돌려 받은 a가 인스턴스이다. 마치 함수를 사용해서 그 결과 값을 돌려 받는 모습과 비슷하다.

## 조금 쉽게 이해해 보기

어린시절 뽕기를 해 본적이 있다면 아래 그림과 비슷한 모양의 뽕기 틀을 본적이 있을 것이다. 뽕기 아저씨가 뽕기를 불에 달군 후 평평한 바닥에 “탁”하고 소리나게 떨어뜨려 넓적하고 동그랗게 만든후에 아래와 비슷한 틀로 모양을 찍어준다. 찍어준 모양대로 깨뜨리지 않고 만들어 오면 아저씨는 뽕기 한개를 더 해 주었다.



이곳에서 설명할 클래스라는 것이 마치 위 뽕기의 틀(별모양, 하트모양)과 비슷하다. 별 모양의 틀(클래스)로 찍으면 별모양의 뽕기(인스턴스)가 생성되고 하트 모양의 틀(클래스)로 찍으면 하트모양의 뽕기(인스턴스)가 나오는 것이다.

클래스란 똑같은 무엇인가를 계속해서 만들어 낼 수 있는 설계도면 같은 것이고

(뽑기 틀), 인스턴스란 클래스에 의해서 만들어진 피조물(별 또는 하트가 찍혀진 뽑기)을 뜻하는 것이다.

### ※ [참고] 객체와 인스턴스의 차이

클래스에 의해서 만들어진 객체를 인스턴스라고도 한다. 그렇다면 객체와 인스턴스의 차이는 무엇일까? 이렇게 생각 해 보자. `cat = Animal()` 이렇게 만들어진 `cat`은 객체이다. 그리고 `cat`이라는 객체는 `Animal`의 **인스턴스(instance)**이다. 즉 인스턴스라는 말은 특정 객체(`cat`)가 어떤 클래스(`Animal`)의 객체인지를 **관계**위주로 설명할 때 사용된다. 즉, “`cat`은 인스턴스” 보다는 “`cat`은 객체”라는 표현이 “`cat`은 `Animal`의 객체” 보다는 “`cat`은 `Animal`의 인스턴스”라는 표현이 훨씬 잘 어울린다.

## 이야기 형식으로 클래스 기초쌓기

이곳에서는 클래스의 개념을 아직 잡지 못한 독자들을 위한 곳이다. 매우 쉽게 설명하려고 노력하였고 최대한 클래스의 핵심 개념에 근접할 수 있도록 배려를 하였다. 클래스에 대해서 잘 아는 독자라도 재밌게 한 번 읽어 보기를 바란다.

## 클래스 변수

좀 더 이해를 쉽게 하기 위해서 다음의 클래스를 보자.

```
>>> class Service:  
...     secret = "영구는 배꼽이 두 개다."
```

위의 클래스의 이름은 `Service`이다. 우리는 위의 `Service` 클래스를 어떤 유용한 정보를 제공해 주는 한 인터넷 업체라고 해 두자. 이 인터넷 업체는 가입한 고

객에게만 유용한 정보를 제공하려 한다. 자 그렇다면 가입을 해야만 이 인터넷 업체의 유용한 정보를 얻을 수 있을 것이다.

가입을 하는 방법은 다음과 같다.

```
>>> pey = Service()
```

위 처럼 하면 pey라는 아이디로 인터넷 서비스 업체인 Service클래스를 이용할 수가 있게 된다. 위의 Service 클래스는 마음이 좋아서 돈도 필요 없고 비밀 번호도 필요 없다고 한다.

자 이제 pey라는 아이디로 위 서비스 업체가 제공하는 정보를 얻어내 보자.

```
>>> pey.secret  
"영구는 배꼽이 두 개다"
```

아이디 이름에다가 서비스 업체가 제공하는 secret라는 변수를 '.'(도트 연산자)를 이용해서 호출하였더니 실제로 어마어마한 정보를 얻을 수 있었다.

## 클래스 함수

하지만 이 Service라는 이름의 인터넷 서비스 제공업체는 신생 벤처 기업이라서 위처럼 단 하나만의 정보만을 제공하고 있다고 한다. 하지만 제공해 주는 정보의 양이 너무 적은 듯하여 가입한 사람들을 대상으로 설문조사를 하였더니 모두들 더하기를 잘 못한다고 대답을 했다. 그래서 이 서비스 업체는 더하기를 해주는 서비스를 제공하기로 마음을 먹었다.

두 수를 더하는 서비스를 제공해 주기 위해서 이 서비스 업체는 다음과 같이 클래스를 업그레이드하였다.

```
>>> class Service:  
...     secret = "영구는 배꼽이 두 개다"  
...     def sum(self, a, b):  
...         result = a + b  
...         print("%s + %s = %s 입니다." % (a, b, result))  
...  
>>>
```

클래스가 업그레이드되어 이제 서비스에 가입한 모든 사람들이 더하기 서비스를 제공받을 수 있게 되었다.

서비스를 사용하는 방법은 다음과 같았다.

먼저 서비스 업체에 가입을 해서 아이디를 받는다.

```
>>> pey = Service()
```

다음에 더하기 서비스를 이용한다.

```
>>> pey.sum(1,1)  
1 + 1 = 2 입니다.
```

더하기의 결과값이 정상적으로 출력되었다.

그렇다면 이번에는 서비스 업체의 입장에서 생각해 보도록 하자. 서비스 업체는 오직 가입한 사람들에게만 서비스를 제공하고 싶어한다. 이를 위해서 그들은 더하기 제공하는 서비스에 트릭을 가했다. 위에서 보았던 더하기 해주는 함수를 다시 보면 다음과 같다.

```
...     def sum(self, a, b):
```

---

```

...         result = a + b
...         print("%s + %s = %s 입니다." % (a, b, result))

```

누군가 이 서비스 업체의 더하기 서비스를 쓰고 싶다고 요청했을 때 이 사람이 가입을 한 사람인지 안한 사람인지 가리기 위해서 위처럼 sum이라는 함수의 첫 번째 입력 값으로 self라는 것을 집어넣었다.

누군가 다음처럼 더하기 서비스를 이용하려 한다고 생각을 해 보자.

```

>>> pey = Service()
>>> pey.sum(1, 1)

```

이렇게 하면 pey라는 아이디를 가진 사람이 이 서비스 업체의 sum이라는 서비스를 이용하겠다고 요청을 한다는 말이다. 위와 같이 했을 때 Service라는 인터넷 제공 업체의 더하기 함수(sum)는 다음처럼 생각한다.

“어 누군가 더하기 서비스를 해 달라고 하네. 그럼 먼저 서비스를 해 주기 전에 이 사람이 가입을 한 사람인지 아닌지 판단해야 겠군. 자 그럼 첫 번째 입력 값으로 뭐가 들어오나 보자. 음... pey라는 아이디를 가진 사람이군. 음, 가입한 사람이군. 서비스를 제공해 주자”

위에서 보듯이 서비스 업체는 sum함수의 첫 번째 입력 값을 통해서 가입 여부를 판단했다. 다시 sum 함수를 보자.

```

...     def sum(self, a, b):
...         result = a + b
...         print("%s + %s = %s 입니다." % (a, b, result))

```

위의 sum함수는 첫 번째 입력 값으로 self라는 것을 받고 두 번째 세 번째로 더 할 숫자를 받는 함수이다. 위의 sum함수는 입력으로 받는 입력 인수의 갯수가

3개이다.

따라서 pey라는 아이디를 가진 사람은 다음처럼 sum함수를 사용해야 할 것이다.

```
pey.sum(pey, 1, 1)
```

sum함수는 첫 번째 입력값을 가지고 가입한 사람인지 아닌지를 판단할 수 있다고 했었다. 따라서 첫 번째 입력 인수로 pey라는 아이디를 주어야지 sum함수는 이 사람이 pey라는 아이디를 가지고 있는 사람임을 알고 서비스를 제공해 줄 것이다. 하지만 위의 sum함수를 호출하는 방법을 보면 pey라는 것이 중복해서 사용되었음을 볼 수 있다.

따라서 다음과 같은 문장만으로 sum함수는 이 사람이 pey라는 아이디를 가지고 있음을 알 수 있을 것이다.

```
>>> pey.sum(1, 1)
```

그래서 pey.sum(pey, 1, 1)이 아닌 pey.sum(1, 1)이 가능하게 된 것이다. 앞의 방법보다는 뒤의 방법이 쓰기도 쉽고 보기도 쉽지 않은가? pey.sum(1, 1)이라는 호출이 발생하면 self는 호출 시 이용했던 인스턴스(즉, pey라는 아이디)로 바뀌게 된다.

※ pey.sum(1, 1) 은 Service.sum(pey, 1, 1) 처럼 사용해도 동일한 결과를 얻는다.

[참고] self라는 변수를 클래스 함수의 첫번째 인자로 받아야 하는 것은 파이썬만의 (좋지않은?) 특징이다. 굳이 왜 'self'가 필요했는지는 파이썬 언어 그 자체를 살펴보아야 한다. 파이썬 언어 개발자가 아닌 우리는 그저 클래스내의 함수의 첫번째 인자는 “무조건 self로 사용을 해야 인스턴스의 함수로 사용할 수 있다.”라고만 알아두자.

## self 제대로 알기

시간이 흘러 더하기 서비스를 계속 제공받던 사람들은 매우 똑똑해 졌다고 한다. 그들은 매우 자신감에 넘치게 되어서 매우 잘난 척을 하기에 이르렀고 서비스 업체에 뭔가 색다른 서비스를 제공해 줄 것을 요구하기에 이르렀다. 그 요구는 황당하게도 더하기 서비스를 제공할 때 “홍길동님  $1 + 2 = 3$  입니다.”처럼 “홍길동”이라는 자신의 이름을 넣어달라는 것이었다.

인터넷 서비스 업체인 Service는 가입자들의 요구가 참 우스웠지만 그래도 자신의 서비스를 이용해 주는 것에 고마운 마음으로 그러한 서비스를 제공해 주기로 마음을 먹었다.

그래서 다음과 같이 또 Service 클래스를 업그레이드 하게 되었다.

이름을 입력받아서 sum 함수를 제공할 때 앞부분에 그 이름을 넣어 주기로 했다.

```
>>> class Service:
...     secret = "영구는 배꼽이 두 개다"
...     def setname(self, name):
...         self.name = name
...     def sum(self, a, b):
...         result = a + b
...         print("%s님 %s + %s = %s입니다." % (self.name, a, b, result))
...
>>>
```

그리고 바뀐 점과 사용법에 대해서 가입한 사람들에게 일러주었다. 그래서 사람들은 다음처럼 위의 서비스를 이용할 수 있었다.

먼저 서비스에 가입을 해서 pey라는 아이디를 얻는다.

```
>>> pey = Service()
```

다음에 pey라는 아이디를 가진 사람의 이름이 “홍길동”임을 서비스에 알려준다.

```
>>> pey.setname("홍길동")
```

다음에 더하기 서비스를 이용한다.

```
>>> pey.sum(1, 1)
```

홍길동님 1 + 1 = 2 입니다.

이제 서비스를 제공해 주는 서비스 업체의 입장에서 다시 한번 생각해 보도록 하자.

우선 이름을 입력 받는 함수 setname을 보자.

```
...     def setname(self, name):  
...         self.name = name
```

아래처럼 pey라는 아이디를 부여 받은 사람이

```
>>> pey = Service()
```

이름을 설정하겠다는 요구를 다음과 같이 하였다.

```
>>> pey.setname("홍길동")
```

위와 같은 상황이 발생 했을 때 서비스 제공 업체의 setname함수는 다음과 같아 생각한다.

“*pey*라는 아이디를 가진 사람이 자신의 이름을 ‘홍길동’으로 설정하려고 하는 구나. 그렇다면 앞으로 *pey*라는 아이디로 접근하면 이 사람의 이름이 ‘홍길동’이라는 것을 잊지 말아야 겠다.”

위와 같은 것을 가능하게 해 주는 것이 바로 **self**이다.

일단 *pey*라는 아이디를 가진 사람이 “‘홍길동’이라는 이름을 `setname` 함수에 입력으로 주면 다음의 문장이 수행된다.

```
self.name = name
```

`self`는 첫 번째 입력 값으로 *pey*라는 아이디를 받게 되므로 다음과 같이 바뀔 것이다.

```
pey.name = name
```

`name`은 두 번째로 입력받은 “‘홍길동’이라는 값”으로 위의 문장은 다시 다음과 같이 바뀔 것이다.

```
pey.name = "홍길동"
```

위 코드의 의미는 *pey*라는 아이디를 가진 사람의 이름은 이제 항상 “‘홍길동’이라는 의미”이다.

이제 아이디 *pey*에 그 사람의 이름을 부여하는 과정이 끝이 났다.

다음으로 “‘홍길동님  $1 + 1 = 2$  입니다.’라는 서비스를 가능하게 해 주기 위한 더하기 함수를 보도록 하자.

```
...     def sum(self, a, b):
...         result = a + b
```

---

```
...           print("%s님 %s + %s = %s입니다." % (self.name, a, b, result))
```

“ $1 + 1 = 2$  입니다.”라는 서비스만을 제공했던 이전의 sum함수와 비교해 보면 다른 점은 단 하나, self.name이라는 것을 출력 문자열에 삽입한 것 뿐이다. 그렇다면 self.name은 무엇일까? 설명하기 전에 sum함수를 이용하기 까지의 과정을 다시 한번 보자.

pey라는 이름의 아이디를 가진 사람이 자신의 이름을 “홍길동”이라고 설정한 다음에 sum함수를 다음과 같이 쓰려고 요청했다고 해 보자.

```
>>> pey = Service()
>>> pey.setname("홍길동")
>>> pey.sum(1, 1)
```

이 때 서비스 업체의 sum함수는 다음과 같이 생각한다.

“*pey*라는 아이디를 가진 사람이 더하기 서비스를 이용하려 하는군. 가입한 사람이 맞군. 아, 그리고 이 사람의 이름은 어디 보자 ‘홍길동’이군, 이름을 앞에 넣어 준 다음 결과 값을 돌려주도록 하자.”

여기서 우리가 알아야 할 사항은 “sum함수가 어떻게 *pey*라는 아이디를 가진 사람의 이름을 알아내게 되었을까?”이다. 먼저 setname함수에 의해서 “홍길동”이라는 이름을 설정해 주었기 때문에 *pey.name*이 “홍길동”이라는 값을 갖게 된다는 사실을 이미 알아보았다. 따라서 sum함수에서도 *self.name*은 *pey.name*으로 치환되기 때문에 sum함수는 *pey*라는 아이디를 가진 사람의 이름을 알아채게 되는 것이다.

※ [중요] 즉, *self*는 Service에 의해서 생성된 인스턴스(예:*pey*)를 지칭한다는 사실을 잊지말자.

## `__init__` 이란 무엇인가?

자, 이제 또 시간이 흘러 가입자 수가 많아지게 되었다. 더하기 해 주는 서비스는 너무 친절하게 서비스를 제공해 주기 때문이었다. 그런데 가끔 이런 문제가 발생한다고 항의 전화가 빗발치듯 들어온다. 그 사람들의 말을 들어보면 다음과 같은 것이다.

```
>>> babo = Service()
>>> babo.sum(1, 1)
```

위와 같이 하면 `babo.setname("나바보")`와 같은 과정이 빠졌기 때문에 여러가 나는 것이라고 골백번 얘기를 하지만 항상 이런 실수를 하는 사람들로부터 항의 전화가 와서 서비스 업체에서는 여간 귀찮은 게 아니었다. 그래서 다음과 같은 아이디어를 떠올렸다. 지금까지는 사람들이 서비스 가입 시 바로 아이디를 주는 방식이었는데 아이디를 줄 때 그 사람의 이름을 입력받아야만 아이디를 주는 방식으로 바꾸면 `babo.setname("나바보")`와 같은 과정을 생략할 수 있을 거란 생각이었다.

위와 같이 해주기 위한 방법을 찾던 중 서비스업체의 실력자 한사람이 `__init__()`란 함수를 이용하자고 제의를 한다. 그 방법은 다음과 같았다.

```
>>> class Service:
...     secret = "영구는 배꼽이 두 개다"
...     def __init__(self, name):
...         self.name = name
...     def sum(self, a, b):
...         result = a + b
...         print("%s님 %s + %s = %s입니다." % (self.name, a, b, result))
...
>>>
```

위의 Service클래스를 이전의 클래스와 비교해 보면 바뀐 부분은 딱 한가지이다. 바로 setname함수의 이름인 setname이 `__init__`으로 바뀐 것이다. 클래스에서 이 `__init__`이란 함수는 특별한 의미를 갖는다. 의미는 다음과 같다.

“인스턴스를 만들 때 항상 실행된다.” 즉, 아이디를 부여받을 때 항상 실행된다 는 말이다.

따라서 이제는 위의 서비스에 가입을 하기 위해서 다음처럼 해야 한다.

```
>>> pey = Service("홍길동")
```

이전에는 `pey = Service()`만을 하면 되었지만 이제는 `__init__`함수 때문에 `pey = Service("홍길동")`처럼 아이디를 부여받을 때 이름 또한 써 주어야 한다.

이전에 했던 방식은 다음과 같다.

```
>>> pey = Service()  
>>> pey.setname("홍길동")  
>>> pey.sum(1, 1)
```

이것이 `__init__` 함수를 이용하면 다음처럼 간략하게 쓸 수 있게 된다.

```
>>> pey = Service("홍길동")  
>>> pey.sum(1, 1)
```

따라서 빗발치던 항의 전화도 멈추게 되고 세 번 입력하던 것을 두 번만 입력하면 되니 모두들 기뻐하게 되었다고 한다.

이상과 같이 인스턴스와 self의 의미를 알기 위해서 이야기 형식으로 클래스에 대해서 알아보았다. 위의 내용은 클래스에 대한 정확한 설명은 아니지만 초보자가 인스턴스와 self의 의미, 그리고 \_\_init\_\_ 함수에 대해서 보다 쉽게 접근할 수 있었을 것이다. 위에서 알아본 pey = Service()로 해서 생성된 pey를 아이디라고 하였는데 이것이 바로 인스턴스라고 불리우는 것임을 잊지 말자.

이제 위에서 알아보았던 기초적인 사항을 바탕으로 클래스에 대해서 자세하게 알아보기로 하자.

## 클래스 자세히 알기

클래스란 인스턴스(Instance)를 만들어 내는 공장과도 같다. 이 인스턴스를 어떻게 사용할 수 있는지를 알려면 클래스의 구조를 보면 알 수 있다. 즉, 클래스는 해당 인스턴스의 청사진(설계도)이라고 할 수 있다. 사실 지금껏 알아온 자료형, 제어문, 함수들만으로도 우리가 원하는 프로그램을 작성하는데는 문제가 없다. 하지만 클래스를 이용하면 보다 우아하게 프로그램을 만들 수 있게 된다.

이제부터 객체지향 프로그래밍의 가장 중심이 되는 클래스에 대해서 자세히 알아보기로 하자. 잘 이해가 안되더라도 낙심하지는 말자. 파이썬에 익숙해지다 보면 반드시 쉽게 이해가 될 것이다.

여러 가지 클래스를 만들어 보면서 클래스에 대해서 자세히 알아보도록 하자.

## 클래스의 구조

클래스는 다음과 같은 모습이다.

```
class 클래스이름[(상속 클래스명)]:  
    <클래스 변수 1>  
    <클래스 변수 2>
```

```
...
def 클래스함수1(self[, 인수1, 인수2,,,]):  
    <수행할 문장 1>  
    <수행할 문장 2>  
    ...  
def 클래스함수2(self[, 인수1, 인수2,,,]):  
    <수행할 문장1>  
    <수행할 문장2>  
    ...  
...
```

위에서 보듯이 class라는 키워드는 클래스를 만들 때 쓰이는 예약어이고 그 바로 뒤에는 클래스이름을 써 주어야 한다. 클래스 이름 뒤에 상속할 클래스가 있다면 괄호(())안에 상속할 클래스 이름을 쓴다. 클래스 내부에는 클래스 변수와 클래스 함수들이 있다.

클래스가 무엇인지 감이 안 오더라도 걱정하지 말고 차근차근 다음의 예를보며 이해해 보자.

## 사칙연산 하는 클래스 만들기

사칙연산을 쉽게 해주는 클래스를 만들어 보자. 사칙연산은 더하기, 빼기, 나누기, 곱하기를 말한다.

FourCal이란 사칙연산을 가능하게 하는 클래스가 다음처럼 동작한다고 가정 해 보자.

먼저 a = FourCal()처럼해서 a라는 객체를 만든다.

```
>>> a = FourCal()
```

다음에 a.setdata(4, 2)처럼해서 4와 2라는 숫자를 a에 지정해 주고

```
>>> a.setdata(4, 2)
```

a.sum()을 하면 두 수의 합한 결과( $4 + 2$ )를 돌려주고

```
>>> print(a.sum())
6
```

a.mul()하면 두수의 곱한 결과( $4 * 2$ )를 돌려주고

```
>>> print(a.mul())
8
```

a.sub()은 두 수를 뺀 결과( $4 - 2$ )를 돌려주고

```
>>> print(a.sub())
2
```

a.div()는 두 수를 나눈 결과( $4 / 2$ )를 돌려준다.

```
>>> print(a.div())
2
```

위와 같은 동작을 하는 FourCal 클래스를 만드는 것이 바로 우리의 목표이다.

※ 참고 - 위와 같은 방식은 클래스를 먼저 만드는 것이 아니라 클래스에 의해서 만들어진 객체를 중심으로 어떤 식으로 동작하게 할 것인지 미리 구상을 해 보

는 방식이다. 그리고 생각했던 것을 하나씩 해결해 나가면서 클래스를 완성하게 된다.

자! 그렇다면 위처럼 동작하는 클래스를 지금부터 만들어 보자. 제일 먼저 할 일은 `a = FourCal()`처럼 객체를 만들 수 있게 해야 한다. 이것은 매우 간단하다. 다음을 따라해 보자.

```
>>> class FourCal:  
...     pass  
...  
>>>
```

우선 대화형 인터프리터에서 위와 같이 `pass`란 문장만을 포함한 `FourCal` 클래스를 만든다. 위 클래스는 아무런 변수나 함수도 포함하지 않지만 우리가 원하는 객체 `a`를 만들 수 있는 기능을 가지고 있다. 확인 해보자.

```
>>> a = FourCal()  
>>> type(a)  
<class '__main__.FourCal'>
```

위에서 보듯이 `a = FourCal()`로 `a`라는 객체를 먼저 만들고 그 다음에 `type(a)`로 `a`라는 객체가 어떤 타입인지 알아보았다. 역시 객체 `a`가 `FourCal` 클래스의 인스턴스임을 보여준다. (※ 참고 - `type`함수는 파이썬이 자체적으로 가지고 있는 내장함수로 객체의 타입을 출력한다.)

하지만 우리가 만들어낸 객체 `a`는 아무런 기능도 가지고 있지 않다. 우리는 더 하기, 나누기, 곱하기, 빼기 등의 기능을 가진 객체를 만들어야 한다. 이러한 기능을 갖춘 객체를 만들기 전에 우선적으로 해 주어야 할 일은 `a`라는 객체에 더하기나 곱하기를 할 때 쓰일 두 개의 숫자를 먼저 부여해 주는 일이다.

다음과 같이 연산을 수행할 대상(4, 2)을 지정할 수 있게 만들어 보자.

```
>>> a.setdata(4, 2)
```

위의 사항이 가능하도록 하기 위해서는 다음과 같이 해야 한다.

```
>>> class FourCal:
...     def setdata(self, first, second):
...         self.first = first
...         self.second = second
...
>>>
```

이전에 만들었던 FourCal클래스에서 pass란 것은 당연히 없어져야 하고 그 대신에 setdata라는 함수를 만들었다. 클래스 함수를 다른 말로 메쏘드(Method)라고도 한다. 어려운 용어이지만 익혀두도록 하자. (즉 setdata라는 함수는 FourCal클래스의 메쏘드이다. ) 그렇다면 이제 setdata메서드에 대해서 자세하게 알아보기로 하자.

보통 우리가 일반적인 함수를 만들 때는 다음과 같이 만든다.

```
def sum(a, b):
    return a+b
```

즉 입력 값이 있고 돌려주는 리턴 값이 있다. 메쏘드도 마찬가지이다. setdata 메쏘드를 다시 적어보면 아래와 같다.

```
def setdata(self, first, second):
    self.first = first
    self.second = second
```

입력 인수로 self, first, second이란 세 개의 입력 값을 받는다. 하지만 일반적인 함수와는 달리 메소드의 첫 번째 입력 인수는 특별한 의미를 갖는다. 위에서 보면 바로 self가 특별한 의미를 갖는 변수이다.

다음의 예를 보면서 자세히 알아 보기로 하자.

```
>>> a = FourCal()  
>>> a.setdata(4, 2)
```

위에서 보는 것처럼 a라는 객체를 만든 다음에 a.setdata(4,2)처럼 하면 FourCal 클래스의 setdata 메소드가 호출되고 setdata 메소드의 첫 번째 인수에는자동으로 a라는 인스턴스가 입력으로 들어가게 된다.

즉 setdata의 입력 인수는 self, first, second처럼 총 세 개이지만 a.setdata(4,2)처럼 두 개의 입력 값만 주어도 a라는 인스턴스가 setdata 함수의 첫 번째 입력을 받는 변수인 self에 대입되게 된다.

```
self: 객체 a, first: 4, second: 2
```

파이썬 클래스에서 가장 헷갈리는 부분이 바로 이 부분이다. setdata라는 함수는 입력 인수로 3개를 받는데 왜 a.setdata(4,2)처럼 두 개만을 주어도 되는가?라는 부분이다. 이것에 대한 답변을 여러분은 이제 알았을 것이다.

사실 거의 사용하지는 않지만 다음과 같이 호출하는 것도 가능하다.

```
FourCal.setdata(a, 4, 2)
```

클래스명.메소드 형태로 호출할 때는 위와 같이 객체 a를 입력인자로 꼭 넣어주어야 한다. 단, 위에서 보았듯이 객체.메소드 형태로 호출할 때는 첫번째 입

력인자(self)를 생략해야 한다.

그 다음으로 중요한 다음의 사항을 보자.

setdata 함수에는 두 개의 수행할 문장이 있다.

```
self.first = first  
self.second = second
```

이것이 뜻하는 바에 대해서 알아보자. 입력인수로 받은 first는 4이고 second는 2라는 것은 이미 알아 보았다.

그렇다면 위의 문장은 다음과 같이 바뀔 것이다.

```
self.first = 4  
self.second = 2
```

여기서 중요한 것은 바로 self이다. self는 a.setdata(4, 2)처럼 호출했을 때 자동으로 첫 번째 입력 인수로 들어오는 인스턴스 a라고 했다. 그렇다면 self.first의 의미는 무엇이겠는가? 당연히 a.first가 될 것이다. self.second는 당연히 a.second가 될 것이다.

따라서 위의 두 문장을 풀어서 쓰면 다음과 같이 될 것이다.

```
a.first = 4  
a.second = 2
```

정말 이런지 확인 해 보도록 하자.

```
>>> a = FourCal()  
>>> a.setdata(4, 2)
```

```
>>> print(a.first)
4
>>> print(a.second)
2
```

b라는 객체 하나를 더 만들어 보자.

```
>>> b = FourCal()
>>> b.setdata(3, 7)
>>> print(b.first)
3
>>> print(a.first)
4
```

a와 b라는 객체는 모두 first라는 변수값을 가지고 있지만 그 값은 각각 다르다. b객체의 first 변수에 3이라는 값을 대입하더라도 a의 first 값이 3으로 변경되지 않는다. a, b 객체는 모두 고유한 저장영역(네임스페이스-Namespace)을 가지고 있기 때문이다.

객체의 변수(예: self.first)는 그 객체에서만 사용되는 값이다. 다른 객체들과 별도로 그 값을 유지한다는 점을 꼭 기억하도록 하자. (클래스에서는 이 부분을 이해하는 것이 가장 중요하다.)

지금껏 완성된 클래스를 다시 써보면 다음과 같다.

```
>>> class FourCal:
...     def setdata(self, first, second):
...         self.first = first
...         self.second = second
...
```

```
>>>
```

지금까지 한 내용이 바로 위의 4줄을 설명하기 위한 것이었다. 위에서 설명한 것들이 이해가 잘 되지 않는다면 다시 한번 읽어보는 것이 좋다. 여기까지를 이해하지 못하면 다음으로 넘어갈 수 없기 때문이다.

자! 그럼 이제 두 개의 숫자 값을 설정해 주었으니 두 개의 숫자를 더하는 기능을 위의 클래스에 추가해 보도록 하자.

우리는 다음과 같이 더하기를 할 수 있는 기능을 갖춘 클래스를 만들어야 한다.

```
>>> a = FourCal()  
>>> a.setdata(4, 2)  
>>> print(a.sum())  
6
```

이것을 가능하게 하기 위해서 FourCal클래스를 다음과 같이 만들었다.

```
>>> class FourCal:  
...     def setdata(self, first, second):  
...         self.first = first  
...         self.second = second  
...     def sum(self):  
...         result = self.first + self.second  
...         return result  
...<br>>>>
```

추가된 것은 sum이란 메쏘드이다. 이 sum 메쏘드만 따로 떼어내서 생각해 보

도록 하자.

```
def sum(self):  
    result = self.first + self.second  
    return result
```

입력으로 받는 값은 self밖에 없고 돌려주는 값은 result이다. a.sum()처럼 하면 sum함수에 자동으로 객체 a가 첫 번째 입력 인수로 들어가게 된다는 것을 명심하자.

그러면 이번에는 돌려주는 값을 보자.

```
result = self.first + self.second
```

위의 내용은 아래와 같이 해석 될 것이다.

```
result = a.first + a.second
```

위의 내용은 a.setdata(4, 2)에서

```
a.first = 4  
a.second = 2
```

라고 이미 설정되었기 때문에 다시 다음과 같이 해석 될 것이다.

```
result = 4 + 2
```

따라서

```
>>> print(a.sum())
```

위처럼 하면 6이란 값을 화면에 출력하게 한다.

여기까지 모두 이해한 독자라면 클래스에 대한 것 80% 이상을 알았다고 보아도 된다. 파이썬의 클래스는 그다지 어렵지 않다.

그렇다면 이번에는 곱하기, 빼기, 나누기 등을 할 수 있게 해보자.

```
>>> class FourCal:  
...     def setdata(self, first, second):  
...         self.first = first  
...         self.second = second  
...     def sum(self):  
...         result = self.first + self.second  
...         return result  
...     def mul(self):  
...         result = self.first * self.second  
...         return result  
...     def sub(self):  
...         result = self.first - self.second  
...         return result  
...     def div(self):  
...         result = self.first / self.second  
...         return result  
...  
>>>
```

mul, sub, div 모두 sum 함수에서 했던 것과 마찬가지 방법이니 따로 설명을 하지는 않겠다.

정말로 모든 것이 제대로 동작하는지 확인해보자.

```
>>> a = FourCal()
>>> b = FourCal()
>>> a.setdata(4, 2)
>>> b.setdata(3, 7)
>>> a.sum()
6
>>> a.mul()
8
>>> a.sub()
2
>>> a.div()
2
>>> b.sum()
10
>>> b.mul()
21
>>> b.sub()
-4
>>> b.div()
0
```

우리가 목표로 했던 사칙연산을 해내는 클래스를 만들어 낸 것이다. a객체와 b 객체는 서로 다른 저장공간을 가지고 있기 때문에 위에서 보듯이 완전히 독립적으로 동작함을 알 수 있다. 즉 클래스에 의해서 생성된 객체들은 다른 객체들과 완전히 다른 저장공간을 가지고 독립적으로 동작함을 알 수 있다.

※ 이러한 의미에서 객체지향 프로그래밍(Object oriented Programming)이라

---

는 말이 생겨나게 된 것이다.

## “박씨네 집” 클래스

이번에는 전혀 다른 내용의 클래스를 한 번 만들어 보자. 사칙 연산을 하는 클래스보다는 조금 재미있게 만들어 보자. “박씨네 집”이라는 클래스를 만들어 보겠다. 먼저 클래스가 어떤 식으로 동작하게 할 지 생각해 보자.

클래스 이름은 HousePark으로 하기로 하자. pey라는 인스턴스를 다음처럼 만든다.

```
>>> pey = HousePark()
```

pey.lastname을 출력하면 “박씨네 집” 이기 때문에 “박”이라는 성을 출력하게 만들기로 하자.

```
>>> print(pey.lastname)
박
```

이름을 설정하면 pey.fullname이 성을 포함한 값을 가지게 하도록 만들자.

```
>>> pey.setname("응용")
>>> print(pey.fullname)
박응용
```

travel이란 함수에 여행을 가고 싶은 곳을 입력으로 주면 다음과 같이 출력해 주는 travel함수도 만들어 보자.

```
>>> pey.travel("부산")
```

박응용, 부산여행을 가다.

우선 여기까지만 만들어 보기로 하자. 어렵지 않을 것이다. 먼저 객체만 단순히 생성할 수 있는 클래스는 다음처럼 만들 수 있다.

```
>>> class HousePark:  
...     pass  
...  
>>>
```

이렇게 하면 `pey = HousePark()` 처럼 해서 객체를 만들 수 있게 된다. 이번에는 `pey.lastname`하면 “박”을 출력하게 하기 위해서 다음처럼 해보자.

```
>>> class HousePark:  
...     lastname = "박"  
...  
>>>
```

`lastname`은 클래스 변수이다. 이 클래스 변수 `lastname`은 `HousePark`클래스에 의해서 생성되는 인스턴스 모두에 `lastname`은 “박”이라는 값을 갖게 하는 기능을 한다. 다음의 예를 보자.

```
>>> pey = HousePark()  
>>> pes = HousePark()  
>>> print(pey.lastname)  
박  
>>> print(pes.lastname)  
박
```

위에서 보듯이 HousePark클래스에 의해서 생긴 인스턴스는 모두 lastname이 “박”으로 설정되는 것을 확인 할 수 있다.

※ 참고 - 클래스 변수를 쓸 수 있는 또 하나의 방법

```
>>> print(HousePark.lastname)
박
```

다음에는 이름을 설정하고 print(pey.fullname)하면 성을 포함한 이름을 출력하도록 만들어 보자.

```
>>> class HousePark:
...     lastname = "박"
...     def setname(self, name):
...         self.fullname = self.lastname + name
...
>>>
```

우선 이름을 설정하기 위해서 setname이란 메쏘드를 만들었다.

위의 setname 메쏘드는 다음처럼 사용될 것이다.

```
>>> pey = HousePark()
>>> pey.setname("응용")
```

위의 예에서 보듯이 setname함수에 “응용”이란 값을 인수로 주어서 결국 self.fullname에는 “박” + “응용”이란 값이 대입되게 된다.

이 과정을 살펴 보면 다음과 같다.

```
self.fullname = self.lastname + name
```

우선 두 번째 입력 값 name 은 “응용” 이므로 다음과 같이 바뀔 것이다.

```
self.fullname = self.lastname + "응용"
```

다음에 self는 setname 함수의 첫 번째 입력으로 들어오는 pey라는 인스턴스이기 때문에 다음과 같이 다시 바뀔 것이다.

```
pey.fullname = pey.lastname + "응용"
```

pey.lastname은 클래스 변수로 항상 “박”이란 값을 갖기 때문에 다음과 같이 바뀔 것이다.

```
pey.fullname = "박" + "응용"
```

따라서 위와 같이 pey.setname(“응용”)을 한 다음에는 다음과 같은 결과를 볼 수 있을 것이다.

```
>>> print(pey.fullname)  
박응용
```

이제 우리가 만들려고 했던 클래스의 기능 중 단 한가지만 남았다.

“박응용”을 여행 보내게 하는 메소드 travel을 HousePark 클래스에 구현해 보자.

```
>>> class HousePark:  
...     lastname = "박"  
...     def setname(self, name):  
...         self.fullname = self.lastname + name
```

---

```

...     def travel(self, where):
...         print("%s, %s여행을 가다." % (self.fullname, where))
...
>>>

```

travel이란 메소드를 추가했다. 입력 값으로 인스턴스(self)와 where를 받는다. 그리고 해당 값을 문자열 포맷팅 연산자를 이용하여 문자열에 삽입한 후 출력한다.

위 클래스는 이제 다음과 같이 사용할 수 있다.

```

>>> pey = HousePark()
>>> pey.setname("응용")
>>> pey.travel("부산")
박응용, 부산여행을 가다.

```

위의 과정을 travel함수의 입장에서 살펴보면 다음과 같다. 우선 travel함수의 입력변수인 self와 where은 다음과 같을 것이다.

```

self: pey
where : "부산"

```

따라서 self.fullname은 pey.fullname이 될 것이고 이 pey.fullname은 pey.setname("응용")에 의해서 만들어진 "박응용"이 될 것이다. 따라서 pey.travel("부산")처럼 하게 되면 위의 예처럼 출력되게 되는 것이다.

## 초기값 설정하기

우리는 위에서 HousePark이라는 클래스를 이용해서 인스턴스를 만들었는데 이 인스턴스에 setname함수를 이용해서 이름을 설정해 주는 방식을 사용했었다.

하지만 위에서 만든 함수를 다음과 같이 실행해 보자.

```
>>> pey = HousePark()
>>> pey.travel("부산")
```

에러가 나게 된다. 에러의 이유는 travel함수에서 self.fullname이란 변수를 필요로 하는 데 self.fullname이란 것은 setname이란 함수에 의해 생성되는 것 이기 때문에 위의 경우 setname을 해주는 부분이 생략되어서 에러가 나게 되는 것이다. 클래스 설계시 이렇게 에러가 날 수 있는 상황을 만들면 좋은 클래스라고 보기 어렵다. 따라서 이런 에러가 나는 것을 방지하기 위해서 pey라는 객체를 만드는 순간 setname 메쏘드가 동작하게 한다면 상당히 편리할 것이다. 이러한 생각으로 나오게 된 것이 바로 `__init__` 이란 메쏘드이다.

### `__init__` 메쏘드, 초기치를 설정한다.

자 그렇다면 위의 클래스를 다음과 같이 바꾸어 보자.

```
>>> class HousePark:
...     lastname = "박"
...     def __init__(self, name):
...         self.fullname = self.lastname + name
...     def travel(self, where):
...         print("%s, %s여행을 가다." % (self.fullname, where))
...
```

---

```
>>>
```

setname 메쏘드의 이름이 `__init__`으로 바뀌기만 하였다.

이것이 어떤 차이를 불러오는지 알아보자. 다음처럼 해보자.

```
>>> pey = HousePark()
TypeError: __init__() takes exactly 2 arguments (1 given)
```

위와 같은 에러 메시지를 볼 수 있을 것이다. 에러 메시지의 원인을 보면 입력 인수로 두 개를 받아야 하는데 1개 만 받았다는 에러이다. 여기서 1개를 받았다는 것은 바로 `pey`라는 객체이다.

에러의 메시지를 보면 알 수 있듯이 `pey = HousePark()`이라고 하는 순간 `__init__` 메쏘드가 호출 된다. `__init__` 메쏘드는 두 개의 입력값(`self, name`)을 필요로 하고 있다.

그렇다면 어떻게 객체를 만들 때 `__init__` 함수에 두 개의 입력값을 줄 수 있을까?

그것은 다음과 같다.

```
>>> pey = HousePark("응용")
```

마치 이전에 보았던 `setname` 메쏘드를 썼던 것과 마찬가지 방법이다. 다만 객체를 생성하는 순간에 입력값으로 “응용”이란 값을 주는 점이 다르다.

이렇게 `__init__` 메쏘드를 이용하면 만들어지는 인스턴스에 초기값을 줄 수 있기 때문에 편리할 때가 많다.

※ `__init__` 메쏘드는 생성자(Constructor)라고도 말한다.

다음과 같이 하면 에러 없이 잘 실행되는 것을 확인 할 수 있을 것이다.

```
>>> pey = HousePark("응용")
>>> print(pey.travel("태국"))
박응용, 태국여행을 가다.
```

## 상속 (Inheritance)

상속이란 말은 “물려받다” 라는 뜻이다. “재산을 상속받다” 할 때의 상속과 같은 의미이다. 클래스에도 이런 개념을 쓸 수가 있다. 어떤 클래스를 만들 때 다른 클래스의 기능을 상속받을 수 있는 것이다. 우리는 “박씨네 집”이라는 HousePark 클래스를 만들었었는데 이번엔 “김씨네 집”이라는 HouseKim 클래스를 만들어 보자.

상속의 개념을 이용하면 다음과 같이 간단하게 할 수 있다. HousePark 클래스는 이미 만들어 놓았다고 가정을 한다. 다음의 예는 HouseKim이라는 클래스가 HousePark 클래스를 상속하는 예제이다.

```
>>> class HouseKim(HousePark):
...     lastname = "김"
...
>>>
```

성(lastname)을 “김”으로 고쳐 주었다. 그리고 아무런 것도 추가 시키지 않았다.

이 HouseKim이란 클래스는 위와 같아

```
class HouseKim(HousePark):
```

클래스명 다음에 팔호안에 다른 클래스를 넣어주면 상속을 하게된다.

위 클래스는 마치 다음처럼 상속을 사용하지 않고 코딩한 것과 완전히 동일하게 동작한다.

```
>>> class HouseKim:  
...     lastname = "김"  
...     def __init__(self, name):  
...         self.fullname = self.lastname + name  
...     def travel(self, where):  
...         print("%s, %s여행을 가다." % (self.fullname, where))  
...  
>>>
```

즉, HousePark클래스와 완전히 동일하게 행동하는 것이다.

다시 다음과 같이 HouseKim이라는 클래스를 만들자.

```
>>> class HouseKim(HousePark):  
...     lastname = "김"  
...  
>>>
```

그리고 다음과 같이 따라해 보자.

```
>>> juliet = HouseKim("줄리엣")  
>>> juliet.travel("독도")  
김줄리엣, 독도여행을 가다.
```

HousePark 클래스의 모든 기능을 상속받았음을 확인할 수 있다. 재미있지 않은가? 상속의 개념을 이용해서 독자는 “양씨네 집”, “이씨네 집” 등을 쉽게 만들 수 있을 것이다.

상속의 개념중 하나 더 알아야 할 것이 있는데 그것은 상속대상이 되는 클래스의 메쏘드 중에서 그 행동을 달리 하고 싶을 때가 있을 것이다. 이럴땐 어떻게 해야 할까?

“김씨네 집” 클래스는 “박씨네 집” 클래스를 상속받았는데 우리는 여기서 “김씨네집” 클래스가 상속받은 travel함수를 “박씨네집” 클래스와 다르게 동작하도록 만들어 보자.

```
>>> juliet = HouseKim("줄리엣")
>>> juliet.travel("독도", 3)
김줄리엣, 독도여행 3일 가네.
```

위처럼 행동할 수 있게끔 HouseKim 클래스를 만들어 보자.

```
>>> class HouseKim(HousePark):
...     lastname = "김"
...     def travel(self, where, day):
...         print("%s, %s여행 %d일 가네." % (self.fullname, where, day))
...
>>>
```

위처럼 travel함수를 다르게 설정하고 싶으면 동일한 이름의 travel함수를 HouseKim 클래스내에서 다시 구현해 주면 된다. 간단하다.

※ 이렇게 메쏘드명을 동일하게 재 구현하는 것을 메쏘드 오버라이딩(Overriding)이라고 말한다.

## 연산자 오버로딩(Overloading)

연산자 오버로딩(Overloading)이란 연산자(+, -, \*, /,,,) 등을 객체끼리 사용할 수 있게 하는 기법을 말한다. 쉽게 말해서 다음과 같은 것을 가능하게 만드는 것이다.

```
>>> pey = HousePark("응용")
>>> juliet = HouseKim("줄리엣")
>>> pey + juliet
박응용, 김줄리엣 결혼했네
```

즉, 객체끼리 연산자 기호를 사용하는 방법을 말하는 것이다.

우선 이미 만들어 보았던 클래스들에 몇가지 사항을 추가하여 에디터를 이용해서 다음처럼 작성해 보자.

```
# house.py

class HousePark:
    lastname = "박"
    def __init__(self, name):
        self.fullname = self.lastname + name
    def travel(self, where):
        print("%s, %s 여행을 가다." % (self.fullname, where))
    def love(self, other):
        print("%s, %s 사랑에 빠졌네" % (self.fullname, other.fullname))
    def __add__(self, other):
        print("%s, %s 결혼했네" % (self.fullname, other.fullname))
```

```

class HouseKim(HousePark):
    lastname = "김"
    def travel(self, where, day):
        print("%s, %s여행 %d일 가네." % (self.fullname, where, day))

pey = HousePark("응용")
juliet = HouseKim("줄리엣")
pey.love(juliet)
pey + juliet

```

위의 프로그램을 실행시키면 다음과 같은 결과를 보여 준다.

```

박응용, 김줄리엣 사랑에 빠졌네
박응용, 김줄리엣 결혼했네

```

위의 프로그램의 실행 부분인 다음을 보자.

```

pey = HousePark("응용")
juliet = HouseKim("줄리엣")
pey.love(juliet)
pey + juliet

```

먼저 pey = HousePark("응용")으로 pey라는 객체를 만들고 juliet이라는 객체 역시 하나 생성한다. 다음에 pey.love(juliet)처럼 love 메소드가 호출되었다.

love 메소드를 보면

```
def love(self, other):
```

---

```
print("%s, %s 사랑에 빠졌네" % (self.fullname, other.fullname))
```

입력 인수로 두 개의 객체를 받는 것을 알 수 있다. 따라서 pey.love(juliet)과 같이 호출하면 self에는 pey가 other에는 juliet이 들어가게 되는 것이다. 따라서 “박응용, 김줄리엣 사랑에 빠졌네”라는 문장을 출력하게 된다.

자! 이제 다음과 같은 문장이다.

```
pey + juliet
```

더하기 표시기호인 ‘+’를 이용해서 객체를 더하려고 시도한다. 이렇게 ‘+’연산자를 객체에 사용하게 되면 클래스의 `__add__`라는 함수가 호출되게 된다.

HousePark에서 사용된 부분을 보면 다음과 같다.

```
def __add__(self, other):
    print("%s, %s 결혼했네" % (self.fullname, other.fullname))
```

pey + juliet처럼 호출되면 `__add__(self, other)` 메쏘드의 `self`는 pey가 되고 `other`는 juliet이 된다. 따라서 “박응용, 김줄리엣 결혼했네”라는 문자열을 출력한다.

자, 이젠 지금껏 만들어본 클래스로 이야기를 만들어 보자. 스토리는 다음과 같다.

```
박응용은 부산에 놀러가고
김줄리엣도 우연히 3일 동안 부산에 놀러간다.
둘은 사랑에 빠져서 결혼하게 된다.
그러다가 바로 싸우고 이혼을 하게 된다.
```

참으로 슬픈 이야기이지만 연산자 오버로딩을 공부하기에 더없이 좋은 이야기이다. 다음처럼 동작시키면

```
pey = HousePark("응용")
juliet = HouseKim("줄리엣")
pey.travel("부산")
juliet.travel("부산", 3)
pey.love(juliet)
pey + juliet
pey.fight(juliet)
pey - juliet
```

결과값을 다음과 같이 나오게 만드는 클래스를 작성하는 것이 목표이다.

박응용 부산여행을 가다.  
 김줄리엣 부산여행 3일 가네.  
 박응용, 김줄리엣 사랑에 빠졌네  
 박응용, 김줄리엣 결혼했네  
 박응용, 김줄리엣 싸우네  
 박응용, 김줄리엣 이혼했네

위에서 보면 `fight`라는 메소드를 추가시켜야 하고 `pey - juliet`을 수행하기 위해서 `__sub__`이라는 ‘-’ 연산자가 쓰였을 때 호출되는 메소드를 만들어 주어야 한다.

자, 최종적으로 만들어진 “박씨네 집” 클래스를 공개한다.

```
class HousePark:
    lastname = "박"
    def __init__(self, name):
```

```

        self.fullname = self.lastname + name
    def travel(self, where):
        print("%s, %s 여행을 가다." % (self.fullname, where))
    def love(self, other):
        print("%s, %s 사랑에 빠졌네" % (self.fullname, other.fullname))
    def fight(self, other):
        print("%s, %s 싸우네" % (self.fullname, other.fullname))
    def __add__(self, other):
        print("%s, %s 결혼했네" % (self.fullname, other.fullname))
    def __sub__(self, other):
        print("%s, %s 이혼했네" % (self.fullname, other.fullname))

class HouseKim(HousePark):
    lastname = "김"
    def travel(self, where, day):
        print("%s, %s 여행 %d일 가네." % (self.fullname, where, day))

pey = HousePark("응용")
juliet = HouseKim("줄리엣")
pey.travel("부산")
juliet.travel("부산", 3)
pey.love(juliet)
pey + juliet
pey.fight(juliet)
pey - juliet

```

결과값은 예상한 대로 다음처럼 나올 것이다.

박응용, 부산여행을 가다.  
김줄리엣, 부산여행 3일 가네.  
박응용, 김줄리엣 사랑에 빠졌네  
박응용, 김줄리엣 결혼했네  
박응용, 김줄리엣 싸우네  
박응용, 김줄리엣 이혼했네

## 2) 모듈

모듈이란 함수나 변수들, 또는 클래스들을 모아놓은 파일이다. 다른 파이썬 프로그램에서 불러쓸 수 있게끔 만들어진 파이썬 파일을 모듈이라 부른다.

우리는 파이썬으로 프로그래밍을 할 때 굉장히 많은 모듈을 사용한다. 물론 이미 다른 사람들에 의해서 만들어진 파이썬 라이브러리들이 그 대표적인 것이 되겠지만 우리가 직접 만들어서 사용해야 할 경우도 생길 것이다. 여기서는 모듈을 어떻게 만들고 또 사용할 수 있는지에 대해서 자세하게 알아보기로 하자.

### 모듈 만들고 불러보기

우선 모듈에 대해서 자세히 살펴보기 전에 간단한 모듈을 한번 만들어 보기로 하자.

```
# mod1.py
def sum(a, b):
    return a + b
```

위와 같이 sum 함수만을 가지고 있는 파일 mod1.py를 만들고 저장하여 보자. 그리고 아래와 같이 도스창(cmd 명령 수행시 나오는 창)을 열고 mod1.py를 저장한 디렉토리로 이동(예: 만약 mod1.py 파일이 c:/python 디렉토리에 저장되어 있다면 cd c:/python)한 다음에 대화형 인터프리터를 실행(python 명령을 수행)한다.

```
C:\Users\pahkey>cd c:/python
c:\python>dir
...
2014-09-23 오후 01:53          49 mod1.py
```

```
...
C:\Python>c:\Python34\python
Python 3.4.3 (v3.4.3:9b73f1c3e601, Feb 24 2015, 22:44:40) [MSC v.1600 ...
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

그리고 아래와 같이 따라해 보자. 꼭 mod1.py가 저장한 위치(위 예에서는 c:/python 디렉토리)로 이동한 다음 다음을 실행해야 한다. 그래야만 대화형 인터프리터에서 mod1.py를 읽을 수 있다. import 는 현재 디렉토리에 있는 파일이나 파일 쌍 라이브러리가 저장되어진 디렉토리에 있는 파일 모듈만을 불러올 수 있다. 이 사항에 대해서는 잠시 후에 알아보도록 하자.

우리가 만든 mod1.py라는 파일을 파일에서 불러서 쓰려면 어떻게 할까? 다음은 import의 사용법이다.

### import 모듈이름

여기서 모듈이름은 mod1.py에서 “.py”라는 확장자를 제거한 mod1만을 가리키는 것이다.

```
>>> import mod1
>>> print(mod1.sum(3,4))
7
```

위처럼 mod1.py를 불러오기 위해서 import mod1과 같이 하였다. import mod1.py 과 같이 사용하는 실수를 하지 않도록 주의 하자. import는 이미 만들어진 파일 모듈을 사용할 수 있게 해주는 것이다. mod1.py파일에 있는 sum함수를 이용하기 위해서는 위의 예에서와 같이 mod1.sum처럼 모듈이름 뒤에 ‘.’(도트 연산자)를 붙이고 함수이름을 써서 사용할 수 있다.

이번에는 mod1.py에 다음의 함수를 추가 시켜 보자.

```
def safe_sum(a, b):
    if type(a) != type(b):
        print("더할수 있는 것이 아닙니다.")
        return
    else:
        result = sum(a, b)
    return result
```

위의 함수가 하는 역할은 서로 다른 타입의 객체끼리 더하는 것을 방지해 준다. 만약 서로 다른 형태의 객체가 입력으로 들어오면 “더할 수 있는 값이 아닙니다”라는 메시지를 출력하고 return문만 단독으로 사용되어 None값을 돌려주게 된다.

이 함수를 mod1.py에 추가 시킨다음 다시 대화형 인터프리터를 열고 다음과 같이 따라하자.

```
>>> import mod1
>>> print(mod1.safe_sum(3, 4))
7
```

import mod1으로 mod1.py파일을 불러온다. 다음에 mod1.safe\_sum(3, 4)로 safe\_num 함수를 호출한다.

```
>>> print(mod1.safe_sum(1, 'a'))
더할 수 있는 값이 아닙니다.
None
>>>
```

만약 서로 다른 형태의 객체가 입력으로 들어오면 에러 메시지를 출력하고 단독으로 쓰인 return에 의해서 None이라는 것을 돌려주게 된다.

또한 sum함수도 다음처럼 바로 호출할 수 있다.

```
>>> print(mod1.sum(10, 20))  
30
```

때로는 mod1.sum, mod1.safe\_sum처럼 쓰기 보다는 그냥 sum, safe\_sum처럼 함수를 쓰고 싶은 사람도 있을 것이다. 이럴때는 “from 모듈이름 import 모듈함수”를 사용하면 된다.

```
from 모듈이름 import 모듈함수
```

다음과 같이 따라해 보자.

```
>>> from mod1 import sum  
>>> sum(3, 4)  
7
```

from ~ import ~를 이용하면 위에서처럼 모듈이름을 붙이지 않고 바로 해당 모듈의 함수를 쓸 수 있다. 하지만 위와 같이 하면 mod1.py파일의 sum함수만을 사용할 수 있게 된다. 그렇다면 sum함수와 safe\_sum함수를 둘다 사용하고 싶을 땐 어떻게 해야 할까?

두가지 방법이 있다.

```
from mod1 import sum, safe_sum
```

첫 번째 방법은 위에서 보는 것과 같이 from 모듈이름 import 모듈함수1, 모듈함수2 처럼 사용하는 방법이다. 콤마로 구분하여 필요한 함수를 불러올 수 있는 방법이다.

```
from mod1 import *
```

두 번째 방법은 위와같이 \* 문자를 이용하는 방법이다. '\*'가 모든 것을 뜻한다고 대부분 알고 있는데 파이썬에서도 마찬가지이다. 위의 from mod1 import \*가 뜻하는 말은 mod1.py의 모든 함수를 불러서 쓰겠다는 말이다. mod1.py에는 함수가 2개 밖에 존재하지 않기 때문에 위의 두가지 방법은 동일하게 적용될 것이다.

### if \_\_name\_\_ == “\_\_main\_\_”: 의 의미

이번에는 mod1.py 파일에 다음과 같이 추가하여 보자.

```
# mod1.py
def sum(a, b):
    return a+b

def safe_sum(a, b):
    if type(a) != type(b):
        print("더할수 있는 것이 아닙니다.")
        return
    else:
        result = sum(a, b)
    return result
```

```
print(safe_sum('a', 1))
print(safe_sum(1, 4))
print(sum(10, 10.4))
```

위와 같은 mod1.py를 에디터로 작성해서 C:/Python이란 디렉토리에 저장을 했다면 위의 프로그램 파일을 다음처럼 실행할 수 있다.

```
C:/Python> c:\Python34\python mod1.py
더할수 있는 것이 아닙니다.
None
5
20.4
```

결과값은 위처럼 나올 것이다. 하지만 문제는 이 mod1.py라는 파일을 import 해서 쓰려고 할 때 생긴다.

도스창을 열고 다음을 따라해 보자.

```
C:\WINDOWS> cd \Python
C:/Python> c:\Python34\python
>>> import mod1
더할수 있는 것이 아닙니다.
None
5
20.4
```

위와 같은 결과를 볼 수 있을 것이다. 우리는 단지 mod1.py파일의 sum과 safe\_sum 함수만을 쓰려고 했는데 위처럼 import mod1을 하는 순간 mod1.py가 실행이 되어서 결과값을 출력한다. 이러한 것을 방지하기 위한 것이 있다.

mod1.py파일에서 마지막 부분을 다음과 같이 수정해 보자.

```
if __name__ == "__main__":
    print(safe_sum('a', 1))
    print(safe_sum(1, 4))
    print(sum(10, 10.4))
```

이것이 뜻하는 의미는 C:/Python> c:\Python34\python mod1.py 처럼 직접 이 파일을 실행시켰을 때는 `__name__ == "__main__"` 이 참이 되어 if문 다음 문장들이 수행되고 대화형 인터프리터나 다른 파일에서 이 모듈을 불러서 쓸때는 `__name__ == "__main__"`이 거짓이 되어 if문 아래문장들이 수행되지 않도록 한다는 뜻이다.

파이썬 모듈을 만든 다음 보통 그 모듈을 테스트하기 위해서 위와 같은 방법을 많이 사용한다. 실제로 그런지 대화형 인터프리터를 열고 실행해 보자.

```
>>> import mod1
>>>
```

위와 같이 고친 다음에는 아무런 결과값을 출력하지 않는 것을 볼 수 있다.

## 클래스나 변수등을 포함한 모듈

위에서 알아본 모듈은 함수만을 포함하고 있지만 클래스나 변수등을 포함할 수도 있다. 다음의 프로그램을 작성해 보자.

```
# mod2.py
PI = 3.141592
```

```
class Math:  
    def solv(self, r):  
        return PI * (r ** 2)  
  
def sum(a, b):  
    return a+b  
  
if __name__ == "__main__":  
    print(PI)  
    a = Math()  
    print(a.solv(2))  
    print(sum(PI , 4.4))
```

클래스와 함수, 변수등을 모두 포함하고 있는 파일이다. 이름을 mod2.py로 하고 C:/Python이란 디렉토리에 저장했다고 가정을 해 보자.

다음과 같이 실행할 수 있다.

```
C:/Python> c:\Python34\python mod2.py  
3.141592  
12.566368  
7.541592
```

이번에는 대화형 인터프리터를 열고 다음과 같이 따라해 보자.

```
C:/Python> c:\Python34\python  
>>> import mod2  
>>>
```

---

`__name__ == "__main__"`이 거짓이 되므로 아무런 값도 출력되지 않는다.

```
>>> print(mod2.PI)
3.141592
```

mod2.PI 처럼 mod2.py에 있는 PI라는 변수값을 사용할 수 있다.

```
>>> a = mod2.Math()
>>> print(a.solv(2))
12.566368
```

위의 예는 mod2.py에 있는 클래스 Math를 쓰는 방법을 보여준다. 위에서 보듯이 모듈내에 있는 클래스를 이용하기 위해서는 '.'(도트연산자)를 이용하여 클래스이름 앞에 모듈이름을 먼저 써 주어야 한다.

```
>>> print(mod2.sum(mod2.PI, 4.4))
7.541592
```

mod2.py에 있는 함수 sum 역시 당연히 사용할 수 있다.

## 다른 프로그램 파일에서 만든 모듈 불러오기

만들어놓은 모듈 파일을 써 먹기 위해서 지금까지 대화형 인터프리터 만을 이용했는데 이번에는 새롭게 만들 파일 내에서 이전에 만들어 놓았던 모듈을 불러서 쓰는 방법에 대해서 알아보기로 하자. 바로 이전에 만든 모듈인 mod2.py라는 파일을 새롭게 만들 파일 프로그램 파일에서 불러보도록 하자. 자! 그럼 에디터로 다음을 함께 작성해 보자.

```
# modtest.py  
import mod2  
  
result = mod2.sum(3, 4)  
print(result)
```

위에서 보듯이 파일에서도 대화형 인터프리터에서 한 것과 마찬가지 방법으로 import mod2로 mod2 모듈을 불러와서 쓰면 된다. 여기서 중요한 것은 modtest.py라는 파일과 mod2.py라는 파일이 동일한 디렉토리에 있어야만 한다는 점이다.

## 모듈 불러오는 또 다른 방법

우리는 지금껏 만든 모듈을 써먹기 위해서 도스창을 열고 모듈이 있는 디렉토리로 이동한 다음이나 쓸 수 있었다. 하지만 항상 이렇게 해야 하는 불편함을 해소할 수 있는 방법이 있다.

만약 독자가 만든 파일 모듈을 항상 C:/Python/Mymodules 라는 디렉토리에 저장했다면 그 디렉토리로 이동할 필요없이 모듈을 불러서 쓸 수 있는 방법에 대해서 알아보자. 우선 위에서 만든 mod2.py 모듈을 C:/Python/Mymodules 라는 디렉토리에 저장한 다음에 다음의 예를 따라해 보도록 하자.

먼저 sys 모듈을 불러보자.

```
>>> import sys
```

sys 모듈은 파일을 설치할 때 함께 오는 라이브러리 모듈이다. sys에 대한 얘기는 뒤에서 또다시 다룰 것이다. 우리는 sys 모듈을 이용해서 파일 라이브러리가 설치되어 있는 디렉토리를 확인 할 수 있다.

다음과 같이 해보자.

```
>>> sys.path
[‘’, ‘C:\\Windows\\SYSTEM32\\python34.zip’, ‘c:\\Python34\\DLLs’, ‘c:\\Pyt
```

sys.path는 파일 라이브러리들이 설치되어 있는 디렉토리를 보여준다. 또한 파일 모듈이 위의 디렉토리에 들어있는 경우에는 해당 디렉토리로 이동할 필요 없이 바로 불러서 쓸 수가 있다.

그렇다면 sys.path에 C:/Python/Mymodules라는 디렉토리를 추가하면 아무데서나 불러쓸 수 있을까?

당연하다.

sys.path의 결과값이 리스트였으므로 우리는 다음과 같이 할 수 있을 것이다.

```
>>> sys.path.append("C:/Python/Mymodules")
>>> sys.path
[‘’, ‘C:\\Windows\\SYSTEM32\\python34.zip’, ‘c:\\Python34\\DLLs’, ‘c:\\Pyt
>>>
```

sys.path.append를 이용해서 C:/Python/Mymodules라는 디렉토리를 sys.path에 추가시키고 다시 sys.path를 보았더니 가장 마지막 요소에 C:/Python/Mymodules라고 추가 된 것을 확인 할 수 있었다.

그렇다면 실제로 모듈을 불러서 쓸 수 있는지 확인해보자.

```
>>> import mod2
>>> print(mod2.sum(3,4))
7
```

이상 없이 불러서 쓸 수 있다. 이렇게 특정한 디렉토리에 있는 모듈을 불러서 쓰고 싶을 때 사용하는 것이 바로 sys.path.append(모듈디렉토리)의 방법이다.

또 다른 방법으로는 PYTHONPATH 환경변수를 사용하는 방법이 있다.

다음과 같이 따라 해 보자.

```
C:\Users\home>set PYTHONPATH=C:\Python\Mymodules
```

```
C:\Users\home>c:\Python34\python.exe
Python 3.4.3 (v3.4.3:9b73f1c3e601, Feb 24 2015, 22:44:40) ...
Type "help", "copyright", "credits" or "license" for more information.
>>> import mod2
>>> print(mod2.sum(3,4))
7
```

set 명령어를 이용하여 PYTHONPATH 환경변수에 mod2.py 파일이 있는 C:\Python\Mymodules 디렉토리를 설정하면 디렉토리 이동이나 별도의 모듈추가 작업 없이 mod2 모듈을 불러서 사용할 수 있게 된다.

### 3) 패키지

패키지(Packages)는 도트(‘.’)를 이용하여 파일 모듈을 계층적(디렉토리 구조)으로 관리할 수 있게 해 준다. 예를 들어 모듈명이 A.B 인 경우 A는 패키지명이 되고 B는 A패키지의 B모듈이 된다.

파이썬 패키지는 물리적으로 다음과 같이 디렉토리와 파일 모듈로 이루어진다.

가상의 *game* 패키지 예

```
game/
    __init__.py
    sound/
        __init__.py
        echo.py
        wav.py
    graphic/
        __init__.py
        screen.py
        render.py
    play/
        __init__.py
        run.py
        test.py
```

game, sound, graphic, play는 디렉토리 명이고 .py 확장자를 가지는 파일은 파일 모듈이다. (`__init__.py` 파일은 조금 특이한 용도로 쓰이는 데 이것에 대해서는 뒤에서 자세하게 다룰 것이다.) game 디렉토리가 이 패키지의 루트 디렉토리이며 sound, graphic, play는 서브디렉토리이다.

보통 간단한 파이썬 프로그램이 아니라면 위와같이 패키지 구조로 파이썬 프로그램을 만드는 것이 공동작업이나 유지보수등 여러면에서 유리할 것이다. 또한 모듈 제작 시 다른 모듈들과 이름이 겹치더라도 패키지 구조의 도움으로 작성한 모듈을 보다 안전하게 사용할 수 있게 해준다.

## 패키지 만들기

이제 위와 비슷한 game 패키지를 만들어 보면 패키지에 대해서 알아보도록 하자.

c:/python이라는 디렉토리 밑에 game 및 기타 서브 디렉토리들을 생성하고 .py 파일들을 다음과 같이 생성 해 보자. (만약 c:/python 이라는 디렉토리가 없다면 먼저 생성하고 진행하도록 하자.)

```
c:/python/game/__init__.py  
c:/python/game/sound/__init__.py  
c:/python/game/sound/echo.py  
c:/python/game/graphic/__init__.py  
c:/python/game/graphic/render.py
```

\_\_init\_\_.py 파일은 파일만 생성하고 내용은 비워두도록 하자.

echo.py 파일은 다음과 같이 만들자.

```
# echo.py  
def echo_test():  
    print ("echo")
```

render.py 파일은 다음과 같이 만들자.

```
# render.py
```

---

```
def render_test():
    print ("render")
```

여기까지 준비가 되었다면 다음을 따라 해 보도록 하자.

우선 아래의 예제들을 수행하기 전에 먼저 우리가 만든 game 패키지를 참조할 수 있도록 다음과 같이 도스창(cmd 명령어 수행시 나오는 창)에서 set명령을 이용하여 PYTHONPATH 환경변수에 c:/python 디렉토리를 추가하고 파이썬 인터프리터(Interactive shell)를 실행하도록 하자.

```
C:\> set PYTHONPATH=c:/python
```

```
C:\> c:\Python34\python
Python 3.4.3 (v3.4.3:9b73f1c3e601, Feb 24 2015, 22:44:40) [MSC v.1600 ...
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

자, 이제 패키지를 이용하여 echo.py의 echo\_test 함수를 실행 해 보자.

### 첫번째 방법

echo 모듈을 import 하여 다음과 같이 실행할 수 있다.

```
>>> import game.sound.echo
>>> game.sound.echo.echo_test()
echo
```

[주의] 아래의 예제들은 import 예제들이므로 하나의 예제를 실행하고 다음 예제를 실행할 때에는 반드시 인터프리터를 종료하고 다시 실행하도록 하자. 인터프리터를 재 시작하지 않을 경우 이전에

import 했던 것들이 메모리에 남아 있으므로 영뚱한 결과가 나올 수 있다. (윈도우즈의 경우 인터프리터 종료는 Ctrl+Z)

## 두번째 방법

echo 모듈이 있는 디렉토리까지를 from ... import 하여 다음과 같이 실행 할 수 있다.

```
>>> from game.sound import echo  
>>> echo.echo_test()  
echo
```

## 세번째 방법

다음과 같이 echo모듈의 echo\_test 함수를 직접 import 하여 실행 할 수 있다.

```
>>> from game.sound.echo import echo_test  
>>> echo_test()  
echo
```

하지만 다음과 같이 사용하는 것은 불가능하다.

```
>>> import game  
>>> game.sound.echo.echo_test()  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
AttributeError: 'module' object has no attribute 'sound'
```

import game 을 수행하면 game 디렉토리의 모듈 또는 game 디렉토리의 \_\_init\_\_.py 에 정의된 것들만 참조가 가능하다.

또 다음처럼 사용하는 것도 불가능하다.

```
>>> import game.sound.echo.echo_test
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ImportError: No module named echo_test
```

import a.b.c 처럼 import 할 때 가장 마지막 항목인 c는 모듈 또는 패키지여야만 한다.

## `__init__.py` 의 용도

`__init__.py` 파일은 해당 디렉토리가 패키지의 일부임을 알려주는 역할을 한다. 만약 game, sound, graphic 등의 패키지에 포함된 디렉토리 중 `__init__.py` 파일이 없다면 패키지로 인식되지 않는다.

시험삼아 sound 디렉토리의 `__init__.py` 를 제거해 보고 다음을 수행 해 보자.

```
>>> import game.sound.echo
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ImportError: No module named sound.echo
```

sound 디렉토리에 `__init__.py` 파일이 없어서 `ImportError`가 발생하게 된다.

[참고] python3.3 버전부터는 `__init__.py` 파일 없이도 패키지로 인식이 가능하게 변경되었다.([PEP 420](#)) 하지만 하위 버전 호환을 위해 `__init__.py` 파일을 생성하는 것이 안전한 방법이다.

## \_\_all\_\_ 의 용도

다음을 따라 해 보자.

```
>>> from game.sound import *
>>> echo.echo_test()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'echo' is not defined
```

이상하지 않은가? 분명 game.sound 패키지에서 모든것(\*)을 import 하였으므로 echo 모듈을 사용할 수 있어야 할 것 같은데 오류가 발생했다. 이렇게 특정 디렉토리의 모듈을 \*를 이용하여 import 할 경우에는 다음과 같이 해당 디렉토리의 \_\_init\_\_.py 파일에 \_\_all\_\_이라는 변수를 설정하고 import 가능한 모듈을 정의해 주어야 한다.

c:/python/game/sound/\_\_init\_\_.py

```
__all__ = ['echo']
```

여기서 \_\_all\_\_ 이 의미하는 것은 sound 디렉토리에서 \* 를 이용하여 import 할 경우 이 곳에 정의된 echo 모듈만 import 된다는 의미이다.

위와 같이 \_\_init\_\_.py 파일을 변경한 후 위 예제를 수행하면 원하던 결과가 출력되는 것을 확인 할 수 있을 것이다.

```
>>> from game.sound import *
>>> echo.echo_test()
echo
```

(※ 착각하기 쉬운데 `from game.sound.echo import *` 는 `__all__` 과 상관 없이 무조건 된다. 이렇게 상관없이 무조건 되는 경우는 `from a.b.c import *`에서 `from`의 마지막 항목인 `c`가 모듈인 경우이다.)

## relative 패키지

만약 graphic 디렉토리의 render.py 모듈이 sound 디렉토리의 echo.py 모듈을 사용하고 싶다면 어떻게 해야 할까?

다음과 같이 render.py 를 수정하면 가능하다.

*render.py*

```
from game.sound.echo import echo_test

def render_test():
    print ("render")
    echo_test()
```

`from game.sound.echo import echo_test`라는 문장을 추가하여 `echo-test()` 함수를 사용할 수 있도록 수정하였다.

이렇게 수정한 후 다음과 같이 수행 해 보자.

```
>>> from game.graphic.render import render_test
>>> render_test()
render
echo
```

이상없이 잘 수행이 된다.

from game.sound.echo import echo\_test 와 같이 전체 경로를 이용하여 import 할 수도 있지만 다음과 같이 relative 하게 import 하는 것도 가능하다.

(※ 이 기능은 Python 2.5부터 지원되기 시작하였다.)

### render.py

```
from ..sound.echo import echo_test

def render_test():
    print ("render")
    echo_test()
```

from game.sound.echo import echo\_test 으로 변경되었다. 여기서 ..은 부모 디렉토리를 의미한다. graphic과 sound 디렉토리는 동일한 depth 이므로 부모디렉토리(..)을 이용하여 위와 같은 import가 가능하게 된 것이다.

※ [참고] “..”과 같은 relative 접근자는 render.py와 같이 모듈에서만 사용해야 한다. 파이썬 인터프리터에서 relative 접근자를 사용하면 “SystemError: cannot perform relative import”와 같은 오류가 발생할 것이다.

relative한 접근자에는 다음과 같은 것들이 있다.

- .. - 부모디렉토리
- . - 현재디렉토리

## 4) 예외처리

프로그램을 만들다 보면 수없이 많은 에러가 난다. 물론 에러가 나는 이유는 프로그램이 오동작을 하기 하기 위한 파이썬의 배려이다. 하지만 때때로 이러한 에러를 무시하고 싶을 때도 있고, 에러가 날 때 그에 맞는 적절한 처리를 하고 싶을 때가 생기게 된다. 이에 파이썬에는 try, except를 이용해서 에러를 처리할 수 있게 해준다.

### 에러는 어떤 때 일어나는가?

에러를 처리하는 방법을 알기 전에 어떤 상황에서 에러가 나는지 한번 보자. 오타를 쳤을 때 나는 구문 에러 같은 것이 아닌 실제 프로그램에서 잘 발생하는 에러를 보기로 하자. 먼저 없는 파일을 열려고 시도해 보자.

```
>>> f = open("나없는파일", 'r')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
FileNotFoundError: [Errno 2] No such file or directory: '나없는
파일'
```

위의 예에서 보듯이 없는 파일을 열려고 시도하면 `FileNotFoundError`라는 이름의 에러가 발생하게 된다.

※ python 2.7 버전에서는 `FileNotFoundError`가 아닌 `IOError`라는 이름의 에러가 발생한다.

이번에는 또 하나 자주 발생하는 에러로 0으로 어떤 다른 숫자를 나누는 경우를 생각해 보자.

```
>>> 4 / 0
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
```

4를 0으로 나누려니까 ZeroDivisionError라는 이름의 에러가 발생한다.

마지막으로 한가지 에러만 더 들어 보자. 다음의 에러는 정말 빈번하게 일어난다.

```
>>> a = [1,2,3]
>>> a[4]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

a는 [1, 2, 3]이란 리스트인데 a[4]는 a 리스트에서 얻을 수 없는 값이기 때문에 IndexError가 나게 된다. 파일은 이런 에러가 나면 프로그램을 중단하고 에러메시지를 보여준다.

## 에러 처리하기

자, 이제 유연한 프로그래밍을 위한 에러처리의 기법에 대해서 살펴보자.

다음은 에러 처리를 위한 try, except문의 기본 구조이다.

```
try:
    ...
except [발생에러 [as 에러메시지변수]] :
    ...
```

try 블록에서 에러가 나지 않는다면 except 블록은 수행이 되지 않는다. 하지만 try 블록 수행 중 에러가 발생하면 except 블록이 수행된다.

except 구문을 자세히 보자.

`except [발생에러 [as 에러메시지변수]] :`

위에서 보면 [발생에러 [as 에러메시지변수]]의 [] 기호는 생략이 가능하다는 관례적인 표기법이다. 즉 다음처럼 try, except만 써도 되고

첫 번째

`try:`

...

`except:`

...

다음과 같이 발생에러만 포함한 except문을 써도 되고

두 번째

`try:`

...

`except 발생에러:`

...

다음과 같이 발생에러와 에러메시지변수까지 포함한 except문을 써도 된다.

세 번째

```
try:  
    ...  
except 발생에러 as 에러메시지변수:  
    ...
```

이중에서 한가지를 택해서 쓰게 되는데 첫 번째의 경우는 에러 종류에 상관없이 에러가 발생하기만 하면 except 블록을 수행한다는 말이고 두 번째 경우는 에러가 발생했을 때 except문에 미리 정해놓은 에러이름과 일치할 때만 except 블록을 수행한다는 말이고 세 번째의 경우는 두 번째 경우에 플러스하여 에러 메시지까지 알고 싶을 때 사용하는 방법이다.

세 번째 방법의 예를 잠시 들어 보면 다음과 같다.

```
try:  
    4 / 0  
except ZeroDivisionError as e:  
    print(e)
```

위처럼 4를 0으로 나누려고 하면 ZeroDivisionError가 발생하여 except 블록이 실행되고 e라는 에러메시지를 다음과 같이 출력한다.

결과값: division by zero

※ 파이썬 2.7 버전의 경우 위 예제의 except ZeroDivisionError as e: 대신 except ZeroDivisionError, e:와 같이 사용해야 한다. (참고: [파이썬 2.7 vs 파이썬 3](#))

## try .. else

try문은 else절을 지원한다. else절은 예외가 발생하지 않은 경우 실행되게 되며 except절 바로 다음에 나와야 한다.

다음의 예제를 보자.

```
try:
    f = open('foo.txt', 'r')
except FileNotFoundError as e:
    print(str(e))
else:
    data = f.read()
    f.close()
```

만약 foo.txt라는 파일이 없다면 except절이 수행되고 foo.txt파일이 있을경우 else절이 수행될 것이다.

## try .. finally

try문에는 finally절을 사용할 수 있다. finally절은 try문 수행 도중 예외의 발생여부에 상관없이 항상 수행된다. 보통 finally절을 사용하는 이유는 사용한 리소스를 close 해야 할 경우에 많이 사용된다.

다음의 예를 보자.

```
f = open('foo.txt', 'w')
try:
    # 무언가를 수행한다.
finally:
```

```
f.close()
```

foo.txt라는 파일을 쓰기모드로 연 후에 try문이 수행된 후 예외발생 여부에 상관없이 finally절에서 f.close()로 열린파일을 닫아주는 예제이다.

## 에러 회피하기

프로그래밍을 하다보면 특정 에러가 발생할 경우 이 에러는 그냥 통과시켜야 할 경우가 생기곤 한다.

다음의 예를 보자.

```
try:  
    f = open("나없는파일", 'r')  
except FileNotFoundError:  
    pass # 파일이 없더라도 오류가 아니다.
```

try문 내에서 FileNotFoundError가 발생할 경우 pass를 사용하여 에러를 그냥 지나치도록 한 예제이다.

## 에러 발생시키기(raise)

좀 이상하긴 하지만 프로그래밍을 하다보면 종종 에러를 일부러 발생시켜야 할 경우도 생기게 된다. 파이썬은 raise라는 명령어를 이용하여 에러를 강제로 발생시킬 수 있다.

예를 들어 Bird라는 클래스를 상속받는 자식 클래스는 반드시 fly라는 함수를 구현하게 만들고 싶은 경우(강제로 그렇게 하고 싶은 경우)가 있을 수 있다.

```
class Bird:  
    def fly(self):  
        raise NotImplementedError
```

[참고] `NotImplementedError`는 파이썬 내장 에러로 구현되지 않았을 때 발생 시키는 에러로 사용한다.

위 예제는 `Bird` 클래스를 상속하는 자식 클래스는 반드시 `fly`라는 함수를 구현 해야 한다는 의지를 보여주는 예제이다. 만약 위 `Bird` 클래스를 상속한 자식 클래스가 `fly` 함수를 구현하지 않은 상태로 `fly` 함수를 호출한다면 어떻게 될까?

```
class Eagle(Bird):  
    pass  
  
eagle = Eagle()  
eagle.fly()
```

`Eagle` 클래스는 `Bird` 클래스의 `fly` 함수를 구현하지 않았기 때문에 `Bird` 클래스의 `fly()` 함수가 호출되어 `raise` 문에 의해 다음과 같은 `NotImplementedError` 오류가 발생할 것이다.

```
Traceback (most recent call last):  
  File "...", line 33, in <module>  
    eagle.fly()  
  File "...", line 26, in fly  
    raise NotImplementedError  
NotImplementedError
```

`NotImplementedError`를 발생시키지 않으려면 다음과 같이 `Eagle` 클래스에 `fly`

함수를 구현해야 한다.

```
class Eagle(Bird):
    def fly(self):
        print("very fast")

eagle = Eagle()
eagle.fly()
```

위 예제처럼 fly함수를 구현한 후 수행하면 오류없이 다음과 같은 문장이 출력될 것이다.

```
very fast
```

## 5) 내장함수

이제 우리는 파이썬에 대한 대부분의 것들을 알게 되었다. 이제 여러분은 자신이 원하는 프로그램을 직접 만들어볼 수 있을 것이다. 하지만 무엇인가를 만들기 전에 살펴보아야 할 것이 있다. 그것은 자신이 만들려는 프로그램을 이미 누군가가 만들어 놓았을지도 모른다는 사실이다.

물론 공부를 목적으로 누군가가 만들어 놓은 프로그램을 또 만들 수는 있지만 그런 목적이 아니라면 이미 만들어진 것을 다시 만드는 것은 어리석은 행동일 것이다. 그리고 이미 만들어진 것들은 테스트과정을 무수히 거친 훌륭한 것들이기도 하다. 따라서 무엇인가 새로운 프로그램을 만들기 전에 이미 만들어진 것들, 그 중에서도 특히 파이썬 배포본에 함께 들어 있는 파이썬 라이브러리들에 대해서 살펴보는 것은 매우 중요한 일일 것이다.

파이썬 라이브러리는 매우 광범위하고 훌륭한 것들이다. 이 책에서는 모든 라이브러리를 살펴볼 수는 없고 자주 쓰이는 유용한 것들에 대해서만 다루기로 한다.

우선 라이브러리를 살펴보기 전에 파이썬 내장함수를 먼저 보도록 하자.

우리는 이미 몇 가지의 내장 함수들을 사용해 왔다. print, del, type 등이 바로 그것이다. 이러한 파이썬 내장 함수들은 외부 모듈과는 달리 import를 필요로 하지 않는다. 아무런 설정 없이 바로 사용할 수가 있다.

이곳에서 우리는 모든 내장함수에 대해서 알아보지는 않을 것이다. 다만 활용빈도가 높고 중요한 것들에 대해서만 간략히 알아볼 것이다.

### abs

abs(x)는 숫자값을 입력값으로 받았을 때, 그 숫자의 절대값을 돌려주는 함수이다.

```
>>> abs(3)
3
>>> abs(-3)
3
>>> abs(1+2j)
2.2360679774997898
>>>
```

복소수의 절대값은 다음과 같이 구해진다.

$$\text{abs}(a + bj) = \sqrt{a^2 + b^2}$$

## all

all(x)은 반복가능한(iterable) x가 모두 참인경우 True, 거짓이 하나라도 있을 경우 False를 리턴한다. (※ 반복가능한 x는 for문으로 x값을 출력할 수 있는 형태의 자료형을 의미한다.)

다음의 예를 보자.

```
>>> all([1,2,3])
True
```

반복가능한 리스트 자료형 [1,2,3]은 모든 항목이 참이므로 True를 리턴한다.

```
>>> all([1,2,3,0])
False
```

---

반복가능한 리스트 자료형 [1,2,3,0] 중에서 0은 거짓이므로 False를 리턴한다.

### any

any(x)는 반복가능한(iterable) x중 어느것이라도 참이 있을 경우 True를 리턴하고 모두 거짓일 경우에만 False를 리턴한다. all(x)의 반대 경우라고 할 수 있다.

다음의 예를 보자.

```
>>> any([1,2,3,0])  
True
```

반복가능한 리스트 자료형 [1,2,3,0] 중에서 1은 참이므로 True를 리턴한다.

```
>>> any([0, "")  
False
```

반복가능한 리스트 자료형 [0, ""] 의 항목 0과 ""은 모두 거짓이므로 False를 리턴한다.

### chr

chr(i)는 정수 형태의 아스키코드값을 입력으로 받아서 그에 해당하는 문자를 출력하는 함수이다.

```
>>> chr(97)  
'a'  
>>> chr(48)
```

```
'0'
```

```
>>>
```

## dir

dir은 객체가 가지고 있는 변수나 함수를 리스트 형태로 보여준다. 아래의 예는 리스트와 딕셔너리 객체의 관련 함수들(메소드)을 보여주는 예이다. 우리가 앞서 살펴보았던 관련함수들을 구경할 수 있을 것이다.

```
>>> dir([1,2,3])
['append', 'count', 'extend', 'index', 'insert', 'pop', ...]
>>> dir({'1':'a'})
['clear', 'copy', 'get', 'has_key', 'items', 'keys', ...]
```

## divmod

divmod(a,b)는 두 개의 숫자를 입력값으로 받았을 때 그 몫과 나머지를 튜플의 형태로 반환하는 함수이다.

```
>>> divmod(7,3)
(2,1)
>>> divmod(1.3, 0.2)
(6.0, 0.09999999999999978)
```

## enumerate

`enumerate`는 입력값으로 시퀀스 자료형(리스트, 튜플, 문자열)을 입력으로 받아 `enumerate` 객체를 리턴한다. `enumerate` 객체는 첫번째로 그 순서값, 두번째로 그 순서값에 해당되는 시퀀스 자료형의 실제값을 갖는 객체이다.

`enumerate`는 아래의 예와같이 보통 `for`문과 함께 사용된다.

```
>>> for i, name in enumerate(['boby', 'foo', 'bar']):
...     print(i, name)
...
0 boby
1 foo
2 bar
```

반복구간에서 시퀀스 자료형의 값 뿐만 아니라 현재 어떤 위치에 있는지에 대한 인덱스값이 필요한 경우에 `enumerate`함수는 매우 유용하다.

## eval

`eval(expression)`은 입력값으로 실행가능한 문자열(`1+2`, `'hi' + 'a'` 같은 것)을 입력으로 받아서 문자열을 실행한 결과값을 반환하는 함수이다.

```
>>> eval('1+2')
3
>>> eval("'hi' + 'a'")
'hia'
>>> eval('divmod(4,3)')
(1, 1)
```

## filter

filter(function, iterable)는 함수와 반복가능한 자료형을 입력으로 받아서 반복자료형의 값이 하나씩 함수에 인수로 전달될 때, 참을 반환시키는 값만을 따로 모아서 리턴하는 함수이다. filter의 뜻은 무엇인가를 걸러낸다는 뜻이다. 이 의미가 filter 함수에서도 그대로 사용된다.

다음의 예를 보자.

```
#positive.py
def positive(l):
    result = []
    for i in l:
        if i > 0:
            result.append(i)
    return result

print(positive([1,-3,2,0,-5,6]))
```

결과값: [1, 2, 6]

즉, 위의 positive함수는 리스트를 입력값으로 받아서 각각의 요소를 판별해 양수값만 따로 리스트에 모아 그 결과값을 돌려주는 함수이다.

filter함수를 이용하면 아래와 같이 위의 내용을 간단하게 쓸 수 있다.

```
#filter1.py
def positive(x):
```

```
return x > 0

print(list(filter(positive, [1,-3,2,0,-5,6])))
```

결과값: [1, 2, 6]

filter 함수는 첫 번째 인수로 함수명을, 두 번째 인수로는 그 함수에 차례로 들어갈 반복 가능한 자료형(예:리스트, 튜플, 문자열)을 받는다. filter 함수는 두 번째 인수인 각 리스트의 요소들이 첫 번째 인수인 함수에 들어갔을 때 리턴 값이 참인 것만을 끌어서 돌려준다. 위의 예에서는 1, 2, 6 만이 양수로  $x > 0$ 이라는 문장이 참이 되므로 [1, 2, 6]이라는 결과 값을 돌려주게 된다.

lambda를 쓰면 더욱 간편하게 쓸 수 있다. (lambda함수는 잠시 후에 설명한다.)

```
>>> print(list(filter(lambda x: x > 0, [1,-3,2,0,-5,6])))
[1, 2, 6]
```

## hex

hex(x)는 입력으로 정수값을 받아서 그 값을 십육진수값(hexadecimal)로 변환하여 돌려주는 함수이다.

```
>>> hex(234)
'0xea'
>>> hex(3)
'0x3'
```

## id

`id(object)`는 객체를 입력값으로 받아서 객체의 고유값(레퍼런스)을 반환하는 함수이다.

```
>>> a = 3
>>> id(3)
135072304
>>> id(a)
135072304
>>> b = a
>>> id(b)
135072304
```

a, b 3이 모두 같은 객체를 가리키고 있음을 보여 준다.

4는 다른 객체이므로 당연히 `id(4)`는 다른 값을 보여 준다.

```
>>> id(4)
135072292
```

## input

`input([prompt])`은 사용자 입력을 받는 함수이다.

입력 인수로 문자열을 주면 아래의 세번째 예에서 보듯이 그 문자열은 프롬프트가 된다.

```
>>> a = input()
```

---

```

hi
>>> a
'hi'
>>> b = input("Enter: ")
Enter: hi

```

위에서 입력 받은 문자열을 확인해 보면 다음과 같다.

```

>>> b
'hi'

```

※ 파이썬 2.7 버전의 경우 위 예제의 `input` 대신 `raw_input`으로 사용해야 한다. (참고: [파이썬 2.7 vs 파이썬 3](#))

## int

`int(x)`는 스트링 형태의 숫자나 소수점 숫자 등을 정수의 형태로 반환시켜 돌려준다. 정수를 입력으로 받으면 그대로 돌려준다.

```

>>> int('3')
3
>>> int(3.4)
3

```

`int(x, radix)`는 `x`라는 문자열을 `radix`(진수)형태로 계산한 값을 리턴한다.

'11'이라는 이진수 값에 대응되는 십진수 값은 다음과 같이 구한다.

```

>>> int('11', 2)

```

'1A'라는 십육진수 값에 대응되는 십진수 값은 다음과 같이 구한다.

```
>>> int('1A', 16)
26
```

## isinstance

isinstance(object, class)는 입력값으로 인스턴스와 클래스 이름을 받아서 입력으로 받은 인스턴스가 그 클래스의 인스턴스인지를 판단하여 참이면 True, 거짓이면 False를 반환한다.

```
>>> class Person: pass
...
>>> a = Person()
>>> b = 3
>>> isinstance(a, Person)
True
```

위의 예는 a가 Person 클래스에 의해서 생성된 인스턴스임을 확인시켜 준다.

```
>>> isinstance(b, Person)
False
```

b는 Person 클래스에 의해 생성된 인스턴스가 아니므로 isinstance의 결과가 False로 리턴된다.

## lambda

lambda는 함수를 생성할 때 사용되는 예약어로 def와 동일하나 보통 한줄로 간결하게 만들어 사용할 때 사용한다. lambda는 “람다”라고 읽으며 보통 def를 쓸 정도로 복잡하지 않거나 def를 쓸 수 없는 곳에 쓰인다. lambda는 다음과 같이 정의된다.

```
lambda 인수1, 인수2, ... : 인수를 이용한 표현식
```

한 번 만들어 보자.

```
>>> sum = lambda a, b: a+b
>>> sum(3,4)
7
```

lambda를 이용한 sum함수는 인수로 a, b를 받고 a와 b를 합한 값을 돌려준다. 위의 lambda를 이용한 sum함수는 다음의 def를 이용한 함수와 하는 일이 완전히 동일하다.

```
>>> def sum(a, b):
...     return a+b
...
>>>
```

그렇다면 def가 있는데 왜 lambda라는 것이 나오게 되었을까? 이유는 간단하다. lambda는 def 대신 간결하게 사용할 수 있고 def로 쓸 수 없는 곳에 lambda는 쓰일 수 있기 때문이다. 리스트 내에 lambda가 들어간 경우를 살펴보자.

```
>>> l = [lambda a,b:a+b, lambda a,b:a*b]
```

```
>>> l  
[at 0x811eb2c>, at 0x811eb64>]
```

즉 리스트 각각의 요소에 lambda 함수를 만들어 쓸 수 있다. 첫 번째 요소 l[0]은 두개의 입력값을 받아서 합을 돌려주는 lambda 함수이다.

```
>>> l[0]  
at 0x811eb2c>  
>>> l[0](3,4)  
7
```

두 번째 요소 l[1]은 두개의 입력값을 받아서 곱을 돌려주는 lambda 함수이다.

```
>>> l[1](3,4)  
12
```

여러분은 파이썬에 익숙해질수록 lambda 함수를 더욱 더 사랑하게 될 것이다.

## len

len(s)은 인수로 반복가능한(iterable) 자료형(문자열, 리스트, 튜플, 딕셔너리, 집합등)을 입력으로 받아 그 길이(요소의 개수)를 돌려주는 함수이다.

```
>>> len("python")  
6  
>>> len([1,2,3])  
3  
>>> len((1, 'a'))
```

2

## list

list(s)는 인수로 반복가능한(iterable) 자료형을 입력받아 그 요소를 똑같은 순서의 리스트로 만들어 돌려주는 함수이다. 리스트를 입력으로 주면 똑같은 리스트를 복사하여 돌려준다.

```
>>> list("python")
['p', 'y', 't', 'h', 'o', 'n']
>>> list((1,2,3))
[1, 2, 3]
>>> a = [1,2,3]
>>> b = list(a)
>>> b
[1, 2, 3]
>>> id(a)
9164780
>>> id(b)
9220284
```

list(a)는 리스트를 복사해서 다른 리스트를 돌려주는 것을 위의 예에서 확인할 수 있다. 즉 a와 b의 id값이 서로 다르다.

## map

map(f, iterable)은 함수(f)와 반복가능한 자료형(iterable)을 입력으로 받아 입력 자료형의 각각의 요소가 함수 f에 의해 수행된 결과를 뒤어서 리턴하는 함수이다.

다음의 함수를 보자.

```
def two_times(l):
    result = []
    for i in l:
        result.append(i*2)
    return result
```

이 함수는 리스트를 입력받아서 각각의 요소에 2를 곱한 결과값을 돌려주는 함수이다.

다음과 같이 쓰일 것이다.

```
# two_times.py
def two_times(l):
    result = []
    for i in l:
        result.append(i*2)
    return result

result = two_times([1,2,3,4])
print(result)
```

결과값: [2, 4, 6, 8]

이것을 다음과 같이 해보자.

```
>>> def two_times(x): return x*2
...
>>> list(map(two_times, [1,2,3,4]))
[2, 4, 6, 8]
```

즉 map이란 함수는 입력값으로 함수명과 그 함수에 들어갈 인수로 리스트 등의 시퀀스 자료형을 받는다. 이것은 다음과 같이 해석된다. 리스트의 첫 번째 요소인 1이 two\_times 함수의 입력값으로 들어가서  $1 * 2$ 의 과정을 거쳐 2의 값이 결과값 리스트 []에 추가된다. 이 때 결과값은 [2]가 되고, 다음에 리스트의 두 번째 요소인 2가 two\_times 함수의 입력값으로 들어가서 4가 된 다음 이 값이 또 결과값인 [2]에 추가되어 결과값은 [2, 4]가 된다. 총 4개의 요소값이 반복되면 결과값은 [2, 4, 6, 8]이 될 것이고 더 이상의 입력값이 없으면 최종 결과값인 [2, 4, 6, 8]을 돌려준다. 이것이 map 함수가 하는 일이다.

위의 예는 lambda를 쓰면 다음처럼 간략화된다.

```
>>> list(map(lambda a: a*2, [1,2,3,4]))
[2, 4, 6, 8]
```

[map을 이용해서 리스트의 값을 1씩 추가하는 예]

```
# map_test.py
def plus_one(x):
    return x+1
print(list(map(plus_one, [1,2,3,4,5])))
```

결과값: [2,3,4,5,6]

※ 위 예에서 map의 결과를 리스트로 보여주기 위하여 list함수를 이용하여 출력하였다. (파이썬 2.7은 map의 결과가 리스트이므로 위 예에서 list함수를 이용하여 리스트로 변환하지 않아도 된다.)

### max

max(iterable)는 인수로 반복가능한 자료형을 입력 받아 그 최대값을 돌려주는 함수이다.

```
>>> max([1,2,3])  
3  
>>> max("python")  
'y'
```

### min

min(iterable)은 max와는 반대로 반복가능한 자료형을 입력 받아 그 최소값을 돌려주는 함수이다.

```
>>> min([1,2,3])  
1  
>>> min("python")  
'h'
```

### oct

oct(x)는 정수 형태의 숫자를 8진수 문자열로 바꾸어 돌려주는 함수이다.

---

```
>>> oct(34)
'0o42'
>>> oct(12345)
'0o30071'
```

## open

`open(filename, [mode])`은 파일 이름과 읽기 방법을 입력받아 파일 객체를 돌려주는 함수이다. 읽기 방법(mode)이 생략되면 기본적으로 읽기 전용 모드('r')로 파일객체를 만들어 리턴한다.

---

mode	설명
w	쓰기 모드로 파일 열기
r	읽기 모드로 파일 열기
a	추가 모드로 파일 열기
b	바이너리 모드로 파일 열기

---

※ w+, r+, a+ 는 파일을 업데이트할 용도로 사용된다.

b는 w, r, a와 함께 사용된다.

```
>>> f = open("binary_file", "rb")
```

위 예의 “rb”는 바이너리 읽기모드임을 의미한다.

아래 예의 fread와 fread2는 동일한 값을 나타낸다.

```
>>> fread = open("read_mode.txt", 'r')
>>> fread2 = open("read_mode.txt")
```

즉, 읽기모드 부분이 생략되면 기본값으로 'r'을 갖게 된다.

다음은 추가 모드로 파일을 여는 예이다.

```
>>> fappend = open("append_mode.txt", 'a')
```

## ord

ord(c)는 문자의 아스키 값을 돌려주는 함수이다. (chr 함수의 반대 케이스이다.)

```
>>> ord('a')
97
>>> ord('0')
48
```

## pow

pow(x, y)는 x의 y승을 한 결과값을 돌려주는 함수이다.

```
>>> pow(2, 4)
16
>>> pow(3, 3)
27
```

## range

range([start,] stop [,step])는 for문과 잘 사용되는 것으로 인수로 정수값을 주어 그 숫자에 해당되는 범위의 값을 iterable의 형태로 돌려주는 함수이다.

인수가 하나일 경우

```
>>> list(range(5))
[0, 1, 2, 3, 4]
```

인수가 두 개일 경우 (입력으로 주어지는 두 개의 숫자는 시작과 끝을 나타낸다. 끝번호가 포함이 안된다는 것에 주의 하자. )

```
>>> list(range(5, 10))
[5, 6, 7, 8, 9]
```

인수가 세 개일 경우 - 세 번째 인수는 시작번호부터 끝번호까지 가는데 숫자 사이의 거리를 말한다.

```
>>> list(range(1, 10, 2))
[1, 3, 5, 7, 9]
>>> list(range(0, -10, -1))
[0, -1, -2, -3, -4, -5, -6, -7, -8, -9]
```

## repr

repr(object)은 객체를 출력할 수 있는 문자열 형태로 변환하여 돌려주는 함수이다. 이 변환된 값은 주로 eval 함수의 입력으로 쓰인다. str 함수와의 차이점이라면 str으로 변환된 값은 eval의 입력값이 될 수 없는 경우가 있다는 것이다.

```
>>> repr("hi".upper())
"'HI'"
>>> eval(repr("hi".upper()))
'HI'
>>> eval(str("hi".upper()))
Traceback (innermost last):
File "", line 1, in ? eval(str("hi".upper()))
File "", line 0, in ?
NameError: There is no variable named 'HI'
```

위의 경우처럼 str을 쓸 경우 eval 함수의 입력값이 될 수 없는 경우가 있다.

## sorted

sorted(iterable) 함수는 입력으로 받은 반복가능한 자료형을 소트한 후 그 결과를 리스트로 리턴하는 함수이다.

```
>>> sorted([3,1,2])
[1, 2, 3]
>>> sorted(['a','c','b'])
['a', 'b', 'c']
>>> sorted("zero")
['e', 'o', 'r', 'z']
```

```
>>> sorted((3,2,1))  
[1, 2, 3]
```

리스트 자료형에도 sort라는 함수가 있다. 하지만 리스트 자료형의 sort함수는 리스트 객체 그 자체를 소트할 뿐이지 소트된 결과를 리턴하지는 않는다.

다음의 예제로 sorted와 리스트 자료형의 sort함수와의 차이점을 확인해 보자.

```
>>> a = [3,1,2]  
>>> result = a.sort()  
>>> print(result)  
None  
>>> a  
[1, 2, 3]
```

## str

str(object)은 객체를 출력할 수 있는 문자열 형태로 변환하여 돌려주는 함수이다. 단 문자열 그 자체로만 돌려주는 함수이다. 위의 repr함수와의 차이점을 살펴보자.

```
>>> str(3)  
'3'  
>>> str('hi')  
'hi'  
>>> str('hi'.upper())  
'HI'
```

## tuple

tuple(iterable)은 인수로 반복가능한 자료형을 입력 받아 튜플 형태의 자료로 바꾸어 돌려준다. 튜플형이 입력으로 들어오면 그대로 돌려준다.

```
>>> tuple("abc")
('a', 'b', 'c')
>>> tuple([1,2,3])
(1, 2, 3)
>>> tuple((1,2,3))
(1, 2, 3)
```

## type

type(object)은 인수로 객체를 입력받아 그 객체의 자료형이 무엇인지 알려주는 함수이다.

```
>>> type("abc")
<class 'str'>
>>> type([])
<class 'list'>
>>> type(open("test", 'w'))
<class '_io.TextIOWrapper'>
```

## zip

zip(iterable) 함수는 동일한 개수의 요소 값을 갖는 반복가능한 자료형을 묶어 주는 역할을 한다. 예제로 확인해 보자.

```
>>> list(zip([1,2,3], [4,5,6]))  
[(1, 4), (2, 5), (3, 6)]  
>>> list(zip([1,2,3], [4,5,6], [7,8,9]))  
[(1, 4, 7), (2, 5, 8), (3, 6, 9)]  
>>> list(zip("abc", "def"))  
[('a', 'd'), ('b', 'e'), ('c', 'f')]
```

## 6) 외장함수

파이썬 사용에 날개를 달아보자. 전세계의 파이썬 사용자들에 의해서 이미 만들어진 프로그램들을 모아놓은 것이 바로 파이썬 라이브러리이다. ‘라이브러리’는 ‘도서관’이다. 즉, 찾아보는 곳이다. 모든 라이브러리를 공부할 필요는 없다. 그저 어떤 곳에 어떤 라이브러리를 써야 한다는 정도만 알면 된다.

그러기 위해서 어떤 라이브러리들이 존재하고 어떻게 사용하는지에 대해서 알아야 할 필요가 있다. 이 곳에서 파이썬의 모든 라이브러리를 다루지는 않을 것이다. 다만, 자주 쓰이고 꼭 알아야만 한다고 여겨지는 것들에 대해서만 다루도록 하겠다. 그리고 여기서는 주로 실례 위주로 설명할 것이다.

※ 파이썬 라이브러리는 파이썬 설치시 자동으로 컴퓨터에 설치가 된다.

### 명령행에서 인수를 전달(sys.argv)

sys 모듈은 파이썬 인터프리터가 제공하는 변수들과 함수들을 직접 접근하여 제어할 수 있게 해주는 모듈이다.

#### sys.argv

```
C:/User/home> c:\Python34\python test.py abc pey guido
```

위의 예에서와 같이 python test.py 뒤에 또 다른 값들을 함께 넣어주면 sys.argv라는 리스트에 그 값들이 추가되게 된다.

예제따라해 보기)

우선 다음과 같은 파이썬 프로그램을 작성하자. (C:/Python/Mymoduels라는 디렉토리에 저장했다고 가정을 한다.)

```
# argv_test.py
import sys
print(sys.argv)
```

도스창에서 다음과 같이 해보자.

```
C:/Python/Mymodules> python argv_test.py you need python
['argv_test.py', 'you', 'need', 'python']
```

위처럼 python이란 명령어 뒤의 모든 것들이 공백을 기준으로 나뉘어서 sys.argv 리스트의 요소가 됨을 알 수 있다.

### 강제 스크립트 종료법(sys.exit())

```
>>> sys.exit()
```

sys.exit는 Ctrl-Z나 Ctrl-D를 눌러서 대화형 인터프리터를 종료하는 것과 같은 기능을 한다. 또, 프로그램 파일 내에서 쓰이면 프로그램을 중단하게 된다.

### 자신이 만든 모듈 불러서 쓰기(sys.path)

sys.path는 파이썬 모듈들이 저장되어 있는 위치를 나타낸다. 즉, 이 위치에 있는 파이썬 모듈들은 경로에 상관없이 어디에서나 불러올 수가 있다.

다음은 그 실행 결과이다.

```
>>> import sys  
>>> sys.path  
[‘’, ‘C:\\Windows\\SYSTEM32\\python34.zip’, ‘c:\\Python34\\DLLs’, ‘c:\\\\Pyt  
>>>
```

위의 예에서 ”는 현재 디렉토리를 말한다.

```
# path_append.py  
import sys  
sys.path.append("C:/Python/Mymodules")
```

위와 같이 파이썬 프로그램 파일에서 sys.path.append를 이용해 경로명을 추가 시킬 수 있다. 이렇게 하고 난 뒤에는 C:/Python/Mymodules라는 디렉토리에 있는 파이썬 모듈을 불러서 쓸 수가 있다.

### 객체를 그 상태 그대로 파일에 저장하고 싶을 때(pickle)

pickle 모듈은 객체의 형태를 그대로 유지하게 하여 파일에 저장시키고 또 불러 올 수 있게 하는 모듈이다.

아래의 예는 파일을 쓰기 모드로 열어서 딕셔너리 객체인 data를 그대로 pickle.dump로 저장하는 방법이다.

```
>>> import pickle  
>>> f = open("test.txt", 'wb')  
>>> data = {1: 'python', 2: 'you need'}  
>>> pickle.dump(data, f)  
>>> f.close()
```

다음은 pickle.dump에 의해 저장된 파일을 열어 원래 있던 딕셔너리 객체(data)를 그대로 가져오는 예이다.

```
>>> import pickle
>>> f = open("test.txt", 'rb')
>>> data = pickle.load(f)
>>> print(data)
{2:'you need', 1:'python'}
```

위의 예에서는 딕셔너리 객체를 이용하였지만 어떤 자료형이든지 상관없이 저장하고 불러올 수 있다.

### 현재 내 시스템 환경변수값을 알고싶을 때는? (os.environ)

시스템은 제각기 다른 환경 변수 값들을 가지고 있는데 파이썬에는 이러한 환경 변수값 들을 보여주는 os모듈의 environ이다. 다음을 따라해 보자.

```
>>> import os
>>> os.environ
environ({'PROGRAMFILES': 'C:\\\\Program Files', 'APPDATA': 'C:\\\\Users\\\\home\\\\AppData\\\\Local', 'TEMP': 'C:\\\\Temp', 'SYSTEMROOT': 'C:\\\\Windows', 'OS': 'Windows NT', 'COMPUTERNAME': 'LAPTOP-1D9E8A8A', 'LOGONSERVER': '\\\\LAPTOP-1D9E8A8A\\$'}...)
>>>
```

위의 결과값은 필자의 시스템 정보이다. os.environ은 딕셔너리 객체를 돌려준다.

딕셔너리이기 때문에 다음과 같이 호출할 수 있다. 필자의 시스템의 PATH변수이다.

```
>>> os.environ['PATH']
'C:\\\\ProgramData\\\\Oracle\\\\Java\\\\javapath; C:\\\\Windows\\\\system32;C:\\\\Windows
```

## 디렉토리에 대한 것들(os.chdir, os.getcwd)

### os.chdir

아래와 같이 현재 디렉토리의 위치를 변경할 수 있다.

```
>>> os.chdir("C:\\WINDOWS")
```

### os.getcwd

현재 자신의 디렉토리 위치를 돌려준다.

```
>>> os.getcwd()
'C:\\WINDOWS'
```

## 시스템 명령(os.system, os.popen)

### os.system

시스템의 유틸리티나 기타 명령어들을 파일에서 호출할 수 있다. os.system("명령어")처럼 사용해야 한다. 다음은 현재 디렉토리에서 dir을 실행하는 예이다.

```
>>> os.system("dir")
```

---

dir의 결과값이 출력된다.

### os.popen

os.popen은 시스템 명령어를 실행시킨 결과값을 읽기 모드 형태의 파일객체로 돌려준다.

```
>>> f = os.popen("dir")
```

읽은 파일 객체의 내용을 보기 위해서는 다음과 같이 하면 될 것이다.

```
>>> print(f.read())
```

---

### 기타 유용한 os 관련 함수

---

함수	설명
os.mkdir(디렉토리)	디렉토리를 생성한다.
os.rmdir(디렉토리)	디렉토리를 삭제한다. 단, 디렉토리가 비어있어야 삭제가 가능하다.
os.unlink(파일)	파일을 지운다.
os.rename(src, dst)	src라는 이름의 파일을 dst라는 이름으로 바꾼다.

---

## 파일 복사(shutil)

shutil은 파일을 복사해 주는 파이썬 모듈이다.

### shutil.copy(src, dst)

src라는 이름의 파일을 dst로 복사한다. 만약 dst가 디렉토리 이름이라면 src라는 파일이름으로 dst이라는 디렉토리에 복사하고 그 이름이 존재하면 덮어쓰게 된다.

```
>>> import shutil  
>>> shutil.copy("src.txt", "dst.txt")
```

## 디렉토리에 있는 파일들을 리스트로 만들려면 (glob)

가끔 파일을 읽고 쓰는 기능이 있는 프로그램들을 만들다 보면 해당 디렉토리의 파일들의 이름 모두를 알아야 할 때가 있는데 이럴 때 쓰이는 모듈이 바로 glob이다.

### glob(pathname)

glob모듈은 해당 디렉토리내의 파일들을 읽어서 리턴한다. \*, ?등의 메타문자를 써서 원하는 파일만을 읽어들일 수도 있다.

다음은 C:/Python이란 디렉토리에 있는 파일중 이름이 Q문자로 시작하는 파일들을 모두 찾아서 읽는 예이다.

```
>>> import glob
```

---

```
>>> glob.glob("C:/Python/Q*")
['C:\Python\quiz.py', 'C:\Python\quiz.py.bak']
>>>
```

## 임시파일 (tempfile)

임시적으로 파일을 만들어서 쓸 때 유용하게 쓰이는 모듈이 바로 tempfile이다. tempfile.mktemp()는 중복되지 않는 임시파일의 이름을 만들어서 돌려준다.

```
>>> import tempfile
>>> filename = tempfile.mktemp()
>>> filename
'C:\WINDOWS\TEMP\~275151-0'
```

tempfile.TemporaryFile()은 임시적인 저장공간으로 사용될 파일 객체를 돌려준다. 기본적으로 w+b 의 모드를 갖는다. 이 파일객체는 f.close()가 호출될 때 자동으로 사라지게 된다.

```
>>> import tempfile
>>> f = tempfile.TemporaryFile()
>>> f.close() # 생성한 임시파일이 자동으로 삭제됨
```

## 시간에 관한 것들(time)

time 모듈에 관련된 유용한 함수는 굉장히 많다. 그 중에서 가장 유용한 몇 가지만 알아보도록 하자.

### time.time

time.time()은 UTC(Universal Time Coordinated 협정 세계 표준시)를 이용하여 현재의 시간을 실수형태로 반환하여 주는 함수이다. 1970년 1월 1일 0시 0분 0초를 기준으로 지난 시간을 초단위로 돌려준다.

```
>>> import time  
>>> time.time()  
988458015.73417199
```

### time.localtime

time.localtime은 time.time()에 의해서 반환된 실수값을 이용해서 년도, 달, 월, 시, 분, 초,.. 의 형태로 바꾸어 주는 함수이다.

```
>>> time.localtime(time.time())  
time.struct_time(tm_year=2013, tm_mon=5, tm_mday=21, tm_hour=16,  
tm_min=48, tm_sec=42, tm_wday=1, tm_yday=141, tm_isdst=0)
```

### time.asctime

위의 time.localtime에 의해서 반환된 터플 형태의 값을 인수로 받아서 알아보기 쉬운 날짜와 시간 형태의 값을 반환하여 주는 함수이다.

```
>>> time.asctime(time.localtime(time.time()))  
'Sat Apr 28 20:50:20 2001'
```

### time.ctime

위의 time.asctime은 간단하게 다음처럼 표현된다.

```
>>> time.ctime()
'Sat Apr 28 20:56:31 2001'
```

### time.strftime

```
time.strftime('출력할 형식포맷코드', time.localtime(time.time()))
```

strftime 함수는 시간에 관계된 것을 세밀하게 표현할 수 있는 여러 가지 포맷 코드를 제공해 준다.

포맷코드	설명	예
%a	요일 줄임말	Mon
%A	요일	Monday
%b	달 줄임말	Jan
%B	달	January
%c	날짜와 시간을 출력함	06/01/01 17:22:21
%d	날(day)	[00,31]
%H	시간(hour)-24시간 출력 형태	[00,23]
%I	시간(hour)-12시간 출력 형태	[01,12]
%j	1년 중 누적 날짜	[001,366]

---

포맷코드	설명	예
%m	달	[01,12]
%M	분	[01,59]
%p	AM or PM	AM
%S	초	[00,61]
%U	1년 중 누적 주-일요일을 시작으로	[00,53]
%w	숫자로 된 요일	[0(일요일),6]
%W	1년 중 누적 주-월요일을 시작으로	[00,53]
%x	현재 설정된 로케일에 기반한 날짜 출력	06/01/01
%X	현재 설정된 로케일에 기반한 시간 출력	17:22:21
%Y	년도 출력	2001
%Z	시간대 출력	대한민국 표준시
%%	문자	%
%y	세기부분을 제외한 년도 출력	01

---

### strftime 예제

```
>>> import time
>>> time.strftime('%x', time.localtime(time.time()))
'05/01/01'
>>> time.strftime('%c', time.localtime(time.time()))
'05/01/01 17:22:21'
```

## time.sleep

time.sleep 함수는 보통 루프 안에서 많이 쓰이는데 일정한 시간 간격을 주기 위해서 주로 쓰이게 된다. 다음 예제를 보자.

```
#sleep1.py
import time
for i in range(10):
    print(i)
    time.sleep(1)
```

위 예제는 1초 간격으로 0부터 9까지의 숫자를 출력하게 된다. time.sleep 함수의 인수로는 실수 형태가 가능하다. 즉 1이면 1초이고 0.5 이면 0.5초가 되는 것이다.

## 파이썬에서 달력쓰기(calendar)

파이썬에서 달력을 볼 수 있게 해 주는 모듈이다.

2001년의 전체 달력을 볼 수가 있다.

```
>>> import calendar
>>> print(calendar.calendar(2001))
```

위와 똑같은 결과값을 보여준다.

```
>>> calendar.prcal(2001)
```

2001년 4월의 달력만을 보여준다.

```
>>> calendar.prmonth(2001, 4)
April 2001
Mo Tu We Th Fr Sa Su
1
2 3 4 5 6 7 8
9 10 11 12 13 14 15
16 17 18 19 20 21 22
23 24 25 26 27 28 29
30
```

다음의 유용한 calendar 모듈의 함수를 보도록 하자. weekday(년도, 월, 일) 함수는 그 날짜에 해당하는 요일 정보를 돌려준다. 월요일은 0, 화요일은 1, 수요일은 2, 목요일은 3, 금요일은 4, 토요일은 5, 일요일은 6이라는 값을 돌려준다.

```
>>> calendar.weekday(2001, 4, 28)
5
```

위의 예에서 2001년 4월 28일은 토요일이 될 것이다.

monthrange(년도, 월) 함수는 입력받은 달의 1일이 무슨 요일인지와 그 달이 몇 일까지 있는지에 대한 정보를 터플 형태로 돌려준다.

```
>>> calendar.monthrange(2001, 4)
(6, 30)
```

위의 예는 2001년 4월의 1일은 일요일이고 30일까지 있다는 것을 보여준다. 주로 날짜 관련된 프로그래밍을 할 때 위의 두 가지 함수는 매우 유용하게 사용된다.

## 난수 발생시키기 (random)

random은 난수발생 모듈이다. random과 randint에 대해서 알아보자.

다음은 0.0에서 1.0 사이의 실수값 중에서 난수값을 돌려주는 예를 보여준다.

```
>>> import random  
>>> random.random()  
0.53840103305098674
```

다음 예는 1에서 10사이의 정수사이에서 난수값을 돌려준다.

```
>>> random.randint(1,10)  
6
```

다음 예는 1에서 55 사이의 정수 사이의 난수 값을 돌려준다.

```
>>> random.randint(1,55)  
43
```

이러한 난수를 이용해서 재미있는 함수를 하나 만들어보자.

```
# random_pop.py  
import random  
def random_pop(data):  
    number = random.randint(0, len(data)-1)  
    return data.pop(number)  
  
if __name__ == "__main__":  
    data = [1,2,3,4,5]
```

```
while data: print(random_pop(data))
```

결과값:

```
2  
3  
1  
5  
4
```

위의 random.pop 함수는 리스트의 요소 중에서 무작위로 하나를 선택하여 꺼낸 다음 그 값을 돌려주는 함수이다. 물론 꺼내어진 리스트의 요소는 pop 메서드에 의해 사라진다.

위 random.pop 함수는 random 모듈의 choice 함수를 사용하여 다음과 같이 보다 직관적으로 만들 수도 있다.

```
def random_pop(data):  
    number = random.choice(data)  
    data.remove(number)  
    return number
```

random.choice 함수는 입력으로 받은 리스트에서 무작위로 하나를 선택하여 리턴해 주는 함수이다.

만약 리스트의 항목을 무작위로 섞고 싶은 경우에는 random.shuffle 함수를 이용하면 된다.

```
>>> import random  
>>> data = [1,2,3,4,5]  
>>> random.shuffle(data)  
>>> data
```

```
[5, 1, 3, 4, 2]
>>>
```

[1,2,3,4,5]라는 리스트가 shuffle함수에 의해 섞여서 [5,1,3,4,2]로 변한것을 확인할 수 있다.

## 파이썬에서의 쓰레드 (threading)

동작하고 있는 프로그램을 프로세스(Process)라고 한다. 보통 한 개의 프로세스는 한 가지의 일을 하지만, 이 쓰레드를 이용하면 한 프로세스 내에서 두 가지 또는 그 이상의 일을 동시에 할 수 있게 된다. 간단한 예제로 설명을 대신 하겠다.

```
import threading
import time

def say(msg):
    while True:
        time.sleep(1)
        print(msg)

for msg in ['you', 'need', 'python']:
    t = threading.Thread(target=say, args=(msg,))
    t.daemon = True
    t.start()

for i in range(100):
    time.sleep(0.1)
```

```
print(i)
```

첫 번째 for문에서 ['you', 'need', 'python']이라는 리스트의 요소 개수만큼 쓰레드가 생성되고 생성된 쓰레드는 say메서드를 수행하게 되어 1초에 한번씩 입력으로 받은 msg변수값을 출력하게 된다.

두 번째 for문은 매 0.1초마다 0부터 99까지 숫자를 출력하는데 바로 이 부분이 메인 프로그램이 되며 이 메인 프로그램이 종료되는 순간 생성된 쓰레드들도 함께 종료가 된다. `t.daemon = True`와 같이 daemon 플래그를 설정하면 주 프로그램이 종료되는 순간 데몬 쓰레드가 함께 종료된다.

위 예제의 실행 결과값은 다음과 비슷할 것이다.

```
0
you
need
python
1
2
3
4
5
6
7
8
9
10
you
need
python
```

---

11

12

...

예제 결과에서 볼 수 있듯이 쓰레드는 메인 프로그램과는 별도로 실행되는 것을 확인 할 수 있다.

이러한 쓰레드 프로그래밍을 가능하게 해주는 것이 바로 `threading.Thread` 클래스로 첫 번째 인수로는 함수명을, 두 번째 인수로는 그 함수의 입력 변수로 들어갈 튜플 형태의 입력 인수를 받는다.

다음과 같이 쓰레드를 클래스로 정의해도 동일한 결과를 얻을 수 있다.

```
import threading
import time

class MyThread(threading.Thread):
    def __init__(self, msg):
        threading.Thread.__init__(self)
        self.msg = msg
        self.daemon = True

    def run(self):
        while True:
            time.sleep(1)
            print(self.msg)

for msg in ['you', 'need', 'python']:
    t = MyThread(msg)
    t.start()
```

```
for i in range(100):
    time.sleep(0.1)
    print(i)
```

쓰레드를 클래스로 정의할 경우에는 `__init__` 메서드에서 `threading.Thread.__init__(self)` 과 같이 부모 클래스의 생성자를 반드시 호출해야 한다. `MyThread`로 생성된 객체의 `start`메서드 실행 시 `MyThread` 클래스의 `run`메서드가 자동으로 수행된다.

## 웹브라우저 실행시키기 (webbrowser)

`webbrowser`는 자신의 시스템에서 사용하는 기본 웹브라우저가 자동으로 실행되게 하는 모듈이다.

아래의 예는 웹브라우저를 자동으로 실행시키고 해당 URL인 `http://google.com`로 가게 해준다.

```
>>> import webbrowser
>>> webbrowser.open("http://google.com")
```

`webbrowser`의 `open` 함수는 웹브라우저가 실행된 상태이면 해당 주소로 이동하고 웹브라우저가 실행되지 않은 상태이면 새로이 웹브라우저가 실행되어 해당 주소로 이동한다.

`open_new` 함수는 이미 웹브라우저가 실행된 상태에서 새로운 창으로 해당 주소가 열리도록 한다.

```
>>> webbrowser.open_new("http://google.com")
```



## 8. 어디서부터 시작할 것인가?

이 곳에서는 아주 짤막한 스크립트나 함수들을 만들어 볼 것이다. 유용할 수도 있고 그렇지 않을 수도 있지만 독자의 프로그래밍 감각을 늘리는데는 더할 나위 없이 좋은 재료들이 될 것이다. 부디 이 책에 있는 것에 대해서만 생각하지 말고 자신이 새로운 것을 직접 만들어 보고 또 연구도 해가면서 파이썬을 공부하도록 하자.

이곳에 소개되는 모든 파이썬 프로그램 예제는 대화형 인터프리터가 아닌 에디터로 작성해야 한다. 스크립트라는 말이 처음 나왔는데 에디터로 작성한 파이썬 프로그램파일을 스크립트라고 부른다. 앞으로는 에디터로 작성한 파이썬 프로그램 파일을 계속 파이썬 스크립트라고 부를 것이니 혼동하지 말도록 하자.

## 1) 내가 프로그램을 만들 수 있을까?

프로그램을 막 시작하려는 사람이 처음 느끼는 벽은 아마도 다음과 같을 것이다.

“문법도 어느 정도 알겠고, 책의 내용도 대부분 다 이해하는데, 이러한 지식을 바탕으로 내가 도대체 어떤 프로그램을 만들 수 있을까?”

이럴 때는 무턱대고 “어떤 프로그램을 짜야지”라는 생각보다는 다른 사람들의 프로그램 파일들을 구경하고 분석하는데서 시작하는 것이 좋다. 그러면서 다른 사람들의 생각도 읽을 수 있고 거기에 더해 뭔가 새로운 아이디어가 떠오르기도 하는 것이다. 하지만 여기서 가장 중요한 것은 자신의 수준에 맞는 소스를 찾는 일이다. 그래서 이 장에서는 아주 쉬운 예제부터 시작해서 차츰 어려워지는 실용적인 예제까지를 다루려고 노력하였다. 이것들을 어떻게 활용하는가는 독자의 몫이다.

이곳에 있는 예제들은 모두 필자가 만든 것인데, 대부분 다른 사람들이 만든 소스를 보고 아이디어를 얻은 것이 많다.

필자는 이제 막 프로그래밍을 해보려는 사람에게 구구단 프로그램을 짜보라고 요구했던 적이 있었다. 생각보다 쉬운 퀴즈였는데 의외로 파이썬 문법도 다 공부한 사람이 프로그램을 어떻게 만들어야 할 지 갈피를 못잡고 있다는 사실은 놀라운 일이었다. 그래서 필자는 다음과 같은 해결책을 알려 주었다.

“입력과 출력”을 생각하라는 것이었다. 우선 구구단 중 먼저 2단을 만들어야 하니까 2를 입력값으로 주었을 때 원하는 출력 값을 생각해 보라는 힌트였다. 그래도 그림이 그려지지 않는 듯 보여 직접 연습장에 그려주면서 설명을 해 주었다. 그것은 다음과 같았다. 독자들도 함께 따라해 보기자를 바란다.

먼저 에디터를 열고 다음과 같이 쓴다. 즉, GuGu라는 함수에 2라는 입력을 주면 result라는 결과 값을 준다.

```
result = GuGu(2)
```

그렇다면 이제 결과값을 어떤 형태로 받을 것인지를 고민한다. 2단이니까 2,4,6,... 18 까지 같 것이다. 아무래도 위와 같은 데이터는 리스트가 좋을 것 같다. 따라서 `result = [2, 4, 6, 8, 10, 12, 14, 16, 18]` 이런 결과를 얻는 것이 좋겠다는 생각을 먼저 하고 나서 프로그래밍을 시작하는 것이다. 그렇다면 의외로 생각이 가볍게 풀려 지는 것을 느낄 수 있을 것이다. 일단 함수를 다음과 같이 만들어 보자.

```
def GuGu(n):
    print(n)
```

위와 같은 함수를 만들고 `GuGu(2)`처럼 하면 2라는 값을 출력하게 된다. 즉 입력으로 2를 받는 것을 확인을 하는 것이다.

다음에는 결과값을 담을 리스트를 하나 생성하자.

```
def GuGu(n):
    result = []
```

다음에는 `result`에 2, 4, 6,... 18을 어떻게 넣어 주어야 할지 생각해 보자. 필자는 다음과 같이 하였다.

```
def GuGu(n):
    result = []
    result.append(n*1)
    result.append(n*2)
    result.append(n*3)
    result.append(n*4)
    result.append(n*5)
```

```
result.append(n*6)
result.append(n*7)
result.append(n*8)
result.append(n*9)
return result
```

정말 무식한 방법이지만 입력값 2를 주었을 때 원하는 결과값을 얻을 수 있었다. 하지만 위의 함수는 반복이 너무 많다. 가만히 보니 1부터 9까지의 숫자만이 틀린 것을 볼 수 있지 않은가? 그렇다면 이번에는 1부터 9까지를 출력해주는 것을 먼저 생각하자.

대화형 인터프리터를 열고 필자는 다음과 같이 테스트 해 보았다.

```
>>> i = 1
>>> while i < 10:
...     print(i)
...     i = i + 1
```

결과값:

```
1
2
3
4
5
6
7
8
9
```

결과는 아주 만족이다. 따라서 위와 같은 것을 GuGu함수에 적용시키기로 결정했다.

이러한 생각을 바탕으로 만든 함수는 다음과 같다.

```
def GuGu(n):
    result = []
    i = 1
    while i < 10:
        result.append(n * i)
        i = i + 1
    return result
```

다음과 같이 테스트를 해 보았다.

```
print(GuGu(2))
```

결과값:

```
[2, 4, 6, 8, 10, 12, 14, 16, 18]
```

결과는 대 만족이다. 사실 위의 함수는 위와같은 과정을 거치지 않고서도 금방 생각할 수 있는 독자들이 많겠지만 위처럼 간단하지 않다면 필자가 했던 방식이 매우 도움이 된다는 것을 금방 알 수 있을 것이다.

즉, 필자가 강조하고 싶은 것은 프로그래밍이란 것은 위에서 보았듯이 매우 구체적으로 접근해야 머리가 덜 아프다는 얘기이다.

자, 이제 앞으로 소개될 예제들을 보면 독자들 나름대로 멋진 생각들을 해보기

바란다.

## 2) 간단한 메모장

파일에 원하는 메모를 저장하고 수정 및 조회가 가능한 간단한 메모장을 만들어 보도록 하자.

```
# memo.py
import sys
import time

def usage():
    print("""
Usage
=====
python %s -v : View memo
python %s -a : Add memo
"""\% (sys.argv[0], sys.argv[0]))

if not sys.argv[1:] or sys.argv[1] not in ['-v', '-a']:
    usage()
elif sys.argv[1] == '-v':
    try: print(open("memo.txt").read())
    except IOError: print("memo does not exist!")
elif sys.argv[1] == '-a':
    word = input("Enter memo: ")
    f = open("memo.txt", 'a')
    f.write(time.ctime() + ': ' + word+'\n')
    f.close()
    print("Added")
```

위 프로그램은 사용자가 도스창에서 ‘python memo.py -v’라고 입력하면 지금 까지의 메모를 출력해 주고 만약 하나도 추가된 것이 없을 때는 “memo does not exist!”라는 문장을 출력해 준다. ‘python memo.py -a’라고 입력하면 메모를 입력받아서 memo.txt라는 파일에 입력된 내용과 현재의 시간을 함께 파일에 적는다.

사용자가 도스창에서 ‘python memo.py -v’나 ‘python memo.py -a’라고 입력하지 않을 경우에는 usage() 함수를 호출한다.

sys.argv를 어떻게 활용하는지 그리고 try.. except 구문을 어떻게 활용했는지를 주목해서 보도록 하자.

### 3) tab을 4개의 space로 바꾸기

이 스크립트는 문서파일을 읽어서 그 문서파일 내에 있는 텨(Tab)을 공백 네 개(4 Space)로 바꾸어 주는 스크립트이다. 필자는 대부분의 파이썬 스크립트를 리눅스의 VI 에디터를 이용하여 작성하는데 들여쓰기를 항상 텨으로 한다. 그런데 이 소스파일을 문서화시킬 때 텨 사이즈가 8이어서 읽기에 좀 불편했었다. 그런 이유로 이런 스크립트를 만들어 보았다.

```
#tabto4.py

import re
import sys

def usage():
    print("Usage: python %s filename" % sys.argv[0])

try: f = open(sys.argv[1])
except: usage(); sys.exit(2)

msg = f.read()
f.close()
p = re.compile(r'\t')
changed = p.sub(" "*4, msg)

f = open(sys.argv[1], 'w')
f.write(changed)
f.close()
```

위 스크립트에서 주목해서 볼 점은 re 모듈로 어떻게 템을 네 개의 공백으로 바꾸었는가이다. 우선 `p = re.compile(r'\t')`로 패턴을 만들었다. 여기서 `r'\t'`의 의미는 \t탭문자 그대로를 뜻한다. 여기서 r은 raw를 말한다.

그리고 만들어진 패턴 p의 함수인 `sub`를 이용해 패턴과 매칭되는 부분을 ”\*4 즉, 공백 네 개로 바꾸었다. `sub`은 패턴과 매치되는 부분을 원하는 문자열로 바꾸어 주는 기능을 가지고 있다.

```
p.sub(" "*4, msg)
```

※ 정규표현식에 대해서는 9장에서 보다 자세한 내용을 다루고 있다.

#### 4) 게시판 페이징

A씨는 게시판 프로그램을 작성하고 있다.

A씨는 게시물의 총 건수와 한 페이지에 보여줄 게시물수를 입력으로 주었을 때 총 페이지수를 리턴하는 프로그램이 필요하다고 한다.

입력 : 총건수(m), 한페이지에 보여줄 게시물수(n) (단 n은 1보다 크거나 같다.  $n \geq 1$ )

출력 : 총페이지수

A씨가 필요한 프로그램을 만들어 보자.

---

다음은 입력값(m, n)에 대한 정상적인 결과 값이다. 본인이 작성한 프로그램의 결과와 일치하는지 확인해 보자.

m	n	출력
0	1	0
1	1	1
2	1	2
1	10	1
10	10	1
11	10	2

---

다음은 필자의 풀이 방법이다.

```
def getPage(m, n):
    page, remainder = divmod(m, n)
    if remainder > 0: page += 1
    return page
```

위 스크립트는 몫과 나머지를 알 수 있는 divmod를 이용하였다. 나머지가 0인 경우에는 페이지를 증가시키지 않고 나머지가 0 이상인 경우에만 페이지를 증가하도록 했다.

## 5) 하위디렉토리 검색

자신의 PC에서 특정 파일만을 찾아내어 특정 문장이 포함되어 있는 부분을 다른 문구로 수정하여 저장해야 한다고 생각해 보자.(이와 비슷한 상황은 실제 업무에서 매우 빈번하게 발생한다.)

파이썬 프로그래머라면 일일이 파일을 찾은 후에 파일을 열어서 수정한 후에 다시 저장하는 행위를 반복하는 어리석은 짓은 하지 않을 것이다.

다음의 소스를 보자.

```
import os

def search(dirname):
    plist = os.listdir(dirname)
    for f in plist:
        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:/")
```

위 소스는 재귀호출을 이용하여 특정 디렉토리부터 시작하여 그 하위의 디렉토리 파일을 순차적으로 검색하는 프로그램이다. 만약 디렉토리일 경우에는 다시 search함수를 다시 호출하고 파일일 경우에는 doFileWork라는 함수를 호출한다.

위의 예는 d드라이브 밑에 있는 파일 중 확장자가 .py인 파일을 모두 출력하는 예제이다. 만약 .py라는 확장자를 가진 모든파일에서 “ABC”를 “DEF”로 바꾸려면 doFileWork를 아래처럼 구현하면 될 것이다.

```
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()
```

## os.walk

첫번째 예제를 os.walk로 재 구현하면 다음과 같다:

```
for (path, dir, files) in os.walk("d:/"):
    for filename in files:
        ext = os.path.splitext(filename)[-1]
        if ext == '.py':
            print("%s/%s" % (path, filename))
```

os.walk를 사용하면 보다 간편하게 코드를 작성할 수 있으며 속도의 향상을 볼 수 있다.

이왕이면 os.walk를 사용하도록 하자!

## 6) 3과 5의 배수 합하기

이번에는 코딩연습을 할 수 있는 프로젝트 오일러라는 사이트를 소개해 볼까 한다. 1번부터 차례대로 풀이 해 나갈 수 있으며 본인이 작성한 답이 맞는지 즉시 확인 할 수 있는 재미있는 사이트이다.

<http://projecteuler.net>

여기서는 프로젝트 오일러의 첫 번째 문제에 한번 도전 해 보자.

<http://projecteuler.net/problem=1>

문제는 다음과 같다:

10미만의 자연수에서 3과 5의 배수를 구하면 3,5,6,9이다. 이들의 총합은 23이다.

1000미만의 자연수에서 3,5의 배수의 총합을 구하라.

다음은 필자의 풀이이다.

```
result = 0
for i in range(1, 1000):
    if i % 3 == 0 or i % 5 == 0:
        result += i
print(result)
```

1부터 1000 미만의 자연수를 나타내기 위해서 `range(1, 1000)`을 이용했고 3, 5의 배수를 찾기 위해 `%` 연산자를 사용하였다. 3으로 나누었을 때 나머지가 0이 되는 수가 3의 배수이므로 3의 배수는 `i%3==0`과 같이 찾을 수 있다. 마찬가지로 5의 배수도 `i%5==0`으로 찾을 수 있다.

이 문제에는 한가지 함정이 있는데 3과 5로 나누어지는 15와 같은 수를 이중으로 카운트해서는 안된다는 점이다. 15와 같이 3의 배수도 되고 5의 배수도 되는 값을 해결하기 위해 or 연산자를 사용하였다.

## 7) 코딩도장

다음은 “위대한 프로그래머가 되려면 어떻게 해야할까?”라는 질문에 대한 “워드 커닝햄”의 답변이다.

저는 작지만 유용한 프로그램들을 매일 작성할 것을 추천합니다. 누군가가 똑같거나 혹은 더 나은 걸 이미 만들었다는 데에 절대 신경쓰지 마세요. 유용성과 복잡성 간의 균형 감각을 얻기 위해서는 당신 자신이 만든 프로그램의 유용성을 직접 느껴봐야만 합니다.

– 워드 커닝햄 (김창준씨와의 인터뷰중에서)

“워드 커닝햄”의 말을 실천할 수 있는 방법 중 하나로 위키독스의 자매사이트인 코딩도장을 소개한다. 코딩도장은 프로그래밍 문제풀이를 통해서 코딩 실력을 수련(Practice)하는 곳이다.

- 코딩도장 : <http://codingdojang.com>

이 곳에서 쉬운문제부터 천천히 풀어보도록 하자.

## 9. 파일 정규표현식과 XML

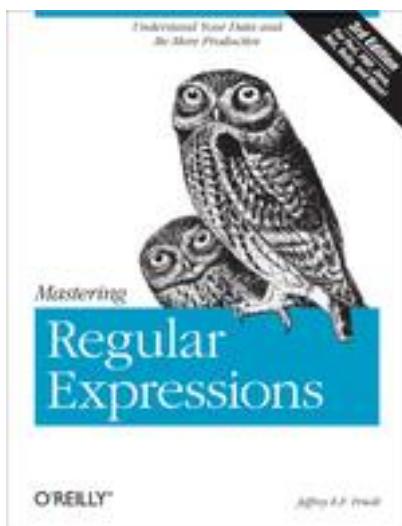
이번 챕터에서는 조금 어려운 걸 해 보려고 한다. 사실 파일 입문자에게는 어울리지 않는 내용이기도 하다. 하지만 그럼에도 불구하고 이곳에 소개하는 이유는 프로그래머의 프로그래밍 인생에 꽤 큰 비중을 차지하는 것들이기 때문이다.

정규표현식과 XML이라는 주제는 꼭 파일에서만 사용되는 기법들이 아니므로 한번 익혀두면 두고두고 써먹을 수 있는 좋은 기술이기도 하다.

이번 챕터는 너무 부담갖지 말고 편하게 한번 읽어 주기 바란다. (프로그래밍 입문자에게 이번 챕터의 내용을 모두 이해하기를 바라는 것은 필자의 욕심일 것이다.)

## 1) 정규표현식

정규표현식(Regular Expressions)은 고수준의 문자열 처리를 위해 사용되는 기법으로 파이썬만의 고유 문법이 아니라 문자열을 처리하는 모든곳에서 사용되는 기술을 말한다. 정규표현식을 배우는 것은 파이썬을 배우는 것과는 또 다른 영역이다. 사실 정규표현식을 제대로 배우려면 다음과 같이 두꺼운 책을 마스터하기를 추천하는 사람들이 많다.



필자는 이 “정규표현식” 챕터를 “점프 투 파이썬”에 포함시켜야 하는지 오랜시간 고민 해 왔다. 출간 당시에는 “정규표현식” 내용이 간단하게라도 실려있었는데 현재 온라인 버전에서는 빠져 있는 상태이다. 왜냐하면 정규표현식은 꽤 오랜기간 코드를 작성해 온 프로그래머라도 잘 모를 수 있는 고급주제의 영역이기 때문에 초보자를 대상으로 하는 “점프 투 파이썬”에는 어울리지 않기 때문이다.

그렇다 하더라도 정규표현식을 잘 알게 되었을 때의 그 달콤한 열매는 무시할 수 없는 것이므로 여기서는 “[파이썬 하우투](#)”를 참고하여 그 정도 수준의 내용을

독자가 이해하고 사용할 수 있도록 노력해 보고자 한다.

여러분이 정규표현식을 잘 다루게 되면 파이썬외에 또 하나의 강력한 무기를 얻게 되는 것이다.

※ 정규표현식은 줄여서 간단히 “정규식”이라고도 말한다.

## 정규표현식은 왜 필요한가?

여러분에게 다음과 같은 문제가 주어졌다고 가정 해 보자.

주민번호를 포함하고 있는 텍스트가 있다. 이 텍스트에 포함된 모든 주민번호의 뒷자리를 \* 문자로 변경하시오.

정규식을 사용하지 않는다면 아마도 다음과 같은 순서로 프로그램을 작성해야 할 것이다.

1. 텍스트를 whitespace로 split 한다.
2. split 된 단어들이 주민번호 형식인지 조사한다.
3. 단어가 주민번호라면 주민번호 뒷부분을 “\*”로 변환한다.
4. split 된 단어들을 재 조립한다.

구현 코드는 아마도 다음과 같을 것이다.

```
data = """
park 800905-1049118
kim 700905-1059119
"""

result = []
for line in data.split("\n"):
    word_result = []
```

---

```

for word in line.split(" "):
    if len(word) == 14 and word[:6].isdigit() and word[7:].isdigit():
        word = word[:6] + "-" + "*****"
    word_result.append(word)
result.append(" ".join(word_result))
print("\n".join(result))

```

결과값:

```

park 800905-*****
kim 700905-*****

```

만약 정규식을 사용한다면 다음과 같이 보다 간편하고 직관적인 코드를 작성할 수 있게된다.

```

import re

data = """
park 800905-1049118
kim 700905-1059119
"""

pat = re.compile("(\\d{6})[-]\\d{7}")
print(pat.sub("\g<1>-*****", data))

```

결과값:

```

park 800905-*****
kim 700905-*****

```

이렇게 간단한 경우에도 정규표현식을 사용할 경우 코드가 상당히 간결해 지는 것을 볼 수 있을 것이다. 만약 찾고자하는 문자열 또는 바꾸어야 할 문자열의 규칙이 매우 복잡하다면 정규식의 효용은 더욱더 증가하게 될 것이다.

이제부터 정규표현식에 대하여 기초부터 차근차근 자세히 알아보도록 하자.

## 정규표현식의 기초

정규표현식에서 사용하는 메타문자(metacharacters)에는 다음과 같은 것들이 있다.

. ^ \$ \* + ? { } [ ] | ( )

정규표현식에 위 메타문자들이 사용되면 이것들은 특별한 의미를 갖게 된다. 이것들에 대해서 모두 알아보는 것이 바로 우리의 목표가 될 것이다.

자, 이제 가장 간단한 정규표현식부터 시작해 보기로 하자.

### 문자클래스(character class, [])

우리가 가장 먼저 살펴 볼 메타문자는 바로 문자클래스(character class)인 []이다.

문자클래스를 만드는 메타문자인 [와 ] 사이에는 어떤 문자도 들어갈 수 있다. 문자클래스로 만들어진 정규식은 “[과 ]사이의 문자들과 매치”라는 의미를 갖는다.

즉, 정규 표현식이 [abc]와 같다면 이 표현식의 의미는 “a,b,c 중 한개의 문자와 매치”를 뜻한다.

이해를 돋기 위해 문자열 “a”, “before”, “dude”가 정규식 [abc]와 어떻게 매치되는지 살펴보자.

- 문자열 “a”는 정규식과 일치하는 문자인 “a”가 있으므로 매치된다.
- 문자열 “before”는 정규식과 일치하는 문자인 “b”가 있으므로 매치된다.
- 문자열 “dude”는 정규식과 일치하는 문자인 a,b,c 중 어느 하나도 포함하고 있지 않으므로 매치되지 않는다.

[] 안의 두문자 사이에 하이픈(-)을 사용하게 되면 두 문자 사이의 범위(From - To)를 의미한다. 예를들어 [a-c]라는 정규표현식은 [abc]와 동일하고 [0-5]는 [012345]와 동일하다.

다음은 하이픈(-)을 이용한 문자클래스의 사용 예이다.

- [a-zA-Z] - 알파벳 모두
- [0-9] - 숫자

문자클래스 내에는 어떤 문자나 메타문자도 사용가능하지만 주의해야 할 메타문자가 한 가지 있다. 그것은 바로 ^ 인데 문자클래스 내에 ^ 메타문자가 사용될 경우에는 반대(not)의 의미를 갖게된다. 즉 [^0-9]라는 정규표현식은 숫자가 아닌 문자만 매치가 되게 되는 것이다.

[0-9] 또는 [a-zA-Z]등은 무척 자주 사용하는 정규표현식이다. 그래서 이렇게 자주사용되는 것들은 별도의 표기법을 따로 만들어 두었다. 다음을 기억 해 두도록 하자.

- \d - 숫자와 매치, [0-9]와 동일한 표현식이다.
- \D - 숫자가 아닌것과 매치, [^0-9]와 동일한 표현식이다.
- \s - whitespace 문자와 매치, [\t\n\r\f\v]와 동일한 표현식이다. 맨 앞의 빈칸은 공백문자(space)를 의미한다.
- \S - whitespace 문자가 아닌것과 매치, [^ \t\n\r\f\v]와 동일한 표현식이다.

- \w - 문자+숫자(alphanumeric)와 매치, [a-zA-Z0-9]와 동일한 표현식이다.
- \W - alphanumeric이 아닌 문자와 매치, [^a-zA-Z0-9]와 동일한 표현식이다.

대문자로 사용된것은 소문자의 반대임을 추측할 수 있을 것이다.

### DOT(.)

정규표현식의 dot(.) 메타문자는 줄바꿈 문자인 \n를 제외한 모든 문자와 매치됨을 의미한다.

다음의 정규식을 보자.

a.b

위 정규식의 의미는 다음과 같다.

“a + 모든문자 + b”

즉 a와 b라는 문자 사이에 어떤 문자가 들어가도 모두 매치된다는 의미이다.

이해를 돋기 위해 문자열 “aab”, “a0b”, “abc”가 정규식 a.b와 어떻게 매치되는지 살펴보자.

- 문자열 “aab”는 가운데 문자 “a”가 모든 문자를 의미하는 .과 일치하므로 정규식과 매치된다.
- 문자열 “a0b”는 가운데 문자 “0”가 모든 문자를 의미하는 .과 일치하므로 정규식과 매치된다.
- 문자열 “abc”는 “a” 문자와 “b” 문자 사이에 어떤 문자라도 하나는 있어야 하는 이 정규식과 일치하지 않으므로 매치되지 않는다.

※ 한가지 주의할 점은 위에서 살펴본 문자클래스(character class) 내에서 사용된 `dot(.)` 메타문자는 모든문자의 의미가 아닌 문자 그대로의 `.`를 의미한다는 점이다.

다음의 정규식을 보자.

`a[.]b`

이 정규식의 의미는 다음과 같다.

“`a` + `Dot(.)`문자 + `b`”

따라서 정규식 `a[.]b` 는 “`a.b`”라는 문자열과는 매치되고 “`a0b`”라는 문자와는 매치되지 않을 것이다.

※ `.`는 `\n`을 제외한 모든 문자와 매치되는데 심지어 `\n`문자와도 매치되게 할 수도 있다. 나중에 알아보겠지만 정규식 작성시 옵션으로 `re.DOTALL`이라는 옵션을 주면 `\n`문자와도 매치되게 할 수 있다.

### 반복 (\*)

다음과 같은 정규식을 보자.

`ca*t`

이 정규식에는 반복을 의미하는 `*` 메타문자가 사용되었다. 여기서 사용된 `*`의 의미는 `*` 바로 앞에 있는 문자 `a`가 0부터 무한개 까지 반복될 수 있다는 의미이다. 즉, 다음과 같은 문자열들이 모두 매치된다.

위 정규식에 대한 매치표는 다음과 같다.

---

정규식	문자열	Match 여부	설명
<code>ca*t</code>	<code>ct</code>	Yes	“ <code>a</code> ”가 0번 반복되어 매치

---

<code>ca*t</code>	<code>ct</code>	Yes	“ <code>a</code> ”가 0번 반복되어 매치
-------------------	-----------------	-----	--------------------------------

---

정규식	문자열	Match 여부	설명
ca*t	cat	Yes	“a”가 0번 이상 반복되어 매치 (1번 반복)
ca*t	caaat	Yes	“a”가 0번 이상 반복되어 매치 (3번 반복)

---

※ 여기서 \* 메타문자의 반복갯수가 무한개까지라고 표현했는데 사실 메모리 제한으로 2억개 정도만 가능하다고 한다.

### 반복 (+)

또하나 반복을 나타내는 메타문자로 + 가 있다. +는 최소 1개 이상의 반복을 필요로 하는 메타문자이다. \*가 반복횟수 0부터의 의미라면 +는 반복횟수 1부터의 의미이다.

다음의 정규식을 보자.

ca+t

위 정규식의 의미는 다음과 같다.

“c + a(1번 이상 반복) + t”

위 정규식에 대한 매치표는 다음과 같다.

---

정규식	문자열	Match 여부	설명
ca+t	ct	No	“a”가 0번 반복되어 매치되지 않음
ca+t	cat	Yes	“a”가 1번 이상 반복되어 매치 (1번 반복)
ca+t	caaat	Yes	“a”가 1번 이상 반복되어 매치 (3번 반복)

---

### 반복 ( $\{m,n\}$ , ?)

그런데 여기서 잠깐 생각해 보자. 반복횟수를 3회만 또는 1회부터 3회까지만 반복시키고 싶을 수도 있지 않을까?

정규식은 역시 이러한 것에 대해서도 준비가 되어 있다. {} 메타문자를 이용하면 반복횟수를 고정시킬 수 있다.  $\{m, n\}$  정규식을 사용하면 반복횟수가 m부터 n인것을 매치시킬 수 있다. m 또는 n을 생략하여 사용도 가능하다. 만약 {3,}처럼 사용하면 반복횟수가 3이상인 경우이며 {,3}처럼 사용하면 반복횟수가 3이하인 것을 의미한다. 생략된 m은 0과 동일한 의미이며 생략된 n은 무한대(2억개 미만)의 의미이다.

※ {1,}은 +와 동일하며 {0,}은 \*와 동일하다.

{ }을 이용한 몇가지 정규식을 살펴보자.

$ca\{2\}t$

위 정규식의 의미는 다음과 같다.

“c + a(반드시 2번 반복) + t”

위 정규식에 대한 매치표는 다음과 같다.

정규식	문자열	Match 여부	설명
$ca\{2\}t$	cat	No	“a”가 1번만 반복되어 매치되지 않음
$ca\{2\}t$	caat	Yes	“a”가 2번 반복되어 매치

또 다음 정규식을 보자.

$ca\{2,5\}t$

위 정규식의 의미는 다음과 같다:

“c + a(2~5회 반복) + t”

위 정규식에 대한 매치표는 다음과 같다.

정규식	문자열	Match 여부	설명
ca{2,5}t	cat	No	“a”가 1번만 반복되어 매치되지 않음
ca{2,5}t	caat	Yes	“a”가 2번 반복되어 매치
ca{2,5}t	caaaaat	Yes	“a”가 5번 반복되어 매치

반복은 아니지만 이와 비슷한 개념으로 ? 이 있다. ? 메타문자가 의미하는 것은 {0, 1} 이다.

다음 정규식을 보자.

ab?c

위 정규식의 의미는 다음과 같다:

“a + b(있어도 되고 없어도 된다) + c”

위 정규식에 대한 매치표는 다음과 같다.

정규식	문자열	Match 여부	설명
ab?c	abc	Yes	“b”가 1번 사용되어 매치
ab?c	ac	Yes	“b”가 0번 사용되어 매치

즉, b문자가 있거나 없거나 둘 다 매치되는 경우이다.

\* , + , ? 메타문자는 모두 {m, n} 형태로 고쳐쓰는 것이 가능하지만 가급적 이해하기 쉽고 표현도 간결한 \* , + , ? 메타문자를 사용하는 것이 좋다.

## 정규표현식 시작하기

지금까지 아주 기초적인 정규표현식에 대해서 알아보았다. 정규식에 대해서 알아야 할 것들이 아직 많이 남아 있지만 그에 앞서 파이썬으로 이러한 정규표현식을 어떻게 사용할 수 있는지에 대해서 먼저 알아보기로 하자.

### 정규식 컴파일하기

파이썬은 정규표현식을 지원하기 위해 re 모듈을 제공한다. re 모듈은 파이썬이 설치될 때 자동으로 설치되는 기본 라이브러리이다.

사용방법은 다음과 같다.

```
>>> import re
>>> p = re.compile('ab*')
```

re.compile 을 이용하여 정규표현식(위 예에서는 ab\*)을 컴파일하고 컴파일된 패턴객체(re.compile의 결과로 리턴되는 객체 p)를 이용하여 그 이후 작업을 수행한다.

정규식 컴파일시 옵션을 주는 것도 가능하다. 예를 들어 다음과 같이 옵션을 주어 컴파일을 할 수 있다.

```
>>> p = re.compile('ab*', re.IGNORECASE)
```

re.IGNORECASE 옵션이 의미하는 것은 대소문자를 구분하지 않는다는 것이다. 따라서 위 패턴은 다음과 같은 문자열들과 모두 매치될 것이다.

- ab
- Ab
- AB
- aB

### 백슬래시(\) 문제

정규표현식을 파이썬에서 사용하려 할 때 혼란을 주게 되는 요소가 한가지 있는데 그것은 바로 백슬래시(\)이다.

예를 들어 LaTex 파일 내에 있는 “\section”이라는 문자열을 찾기 위한 정규식을 만든다고 가정해 보자.

다음과 같은 정규식을 생각해 보자.

```
\section
```

이 정규식은 \s 문자가 whitespace로 해석되어 의도한 대로 매치가 이루어지지 않는다. 위 표현은 다음과 동일한 의미가 된다.

```
[ \t\n\r\f\v]ection
```

따라서 위 정규표현식은 다음과 같이 변경되어야 할 것이다.

```
\\\section
```

즉, 위 정규식에서 사용한 \ 문자가 문자열 그 자체임을 알려주기 위해 백슬래시 두개를 사용하여 Escape 처리를 해야 하는 것이다. 따라서 위 정규식을 컴파일 하려면 다음과 같이 해야 할 것이다.

---

```
>>> p = re.compile('\\\\section')
```

자, 그런데 여기 또 하나의 문제가 발견된다. 위처럼 정규식을 만들어서 컴파일하면 실제 파이썬 정규식 엔진에는 파이썬 문자열 리터럴 규칙에 의하여 \\이 \로 변경되어 \section 이 전달되기 때문이다.

※ 위 문제는 위와같은 정규식을 파이썬에서 사용할 때만 적용되는 문제이다. (파이썬의 리터럴 규칙) 유닉스의 grep, vi등에서는 이러한 문제가 없다.

결국 정규식 엔진에 \\ 문자를 전달하려면 파이썬은 \\\\ 처럼 백슬래시를 4개나 사용해야 한다.

```
>>> p = re.compile('\\\\\\section')
```

이렇게 해야만 원하는 결과를 얻을 수 있을 것이다.

하지만 너무 복잡하지 않은가?

만약 위와 같이 \를 이용한 표현이 반복되서 사용되는 정규식이라면 너무 복잡하여 이해하기 쉽지 않을 것이다. 이러한 문제로 파이썬 정규식에는 Raw string이라는 것이 생겨나게 되었다. 즉 컴파일 해야 하는 정규식이 Raw String 임을 알려줄 수 있도록 파이썬 문법이 만들어진 것이다. 그 방법은 다음과 같다.

```
>>> p = re.compile(r'\\section')
```

위와 같이 정규식 문자열 앞에 r 문자를 선행하면 이 정규식은 Raw String 규칙에 의하여 백슬래시 두개 대신 한개만 써도 두개를 쓴것과 동일한 의미를 갖게된다.

※ 만약 백슬래시를 사용하지 않는 정규식이라면 r의 유무에 상관없이 동일한 정규식이 될 것이다.

## 정규식 검색

이제 컴파일 된 패턴 객체를 이용하여 검색을 수행 해 보자.

컴파일 된 패턴 객체는 다음과 같은 4가지 메쏘드를 제공한다.

Method	목적
match()	문자열의 처음부터 정규식과 매치되는지 조사한다.
search()	문자열 전체를 검색하여 정규식과 매치되는지 조사한다.
findall()	정규식과 매치되는 모든 문자열(substring)을 리스트로 리턴한다
finditer()	정규식과 매치되는 모든 문자열(substring)을 iterator 객체로 리턴한다

match, search는 정규식과 매치될 때에는 match object를 리턴하고 매치되지 않을 경우에는 None을 리턴한다.

이것들에 대한 간단한 예제를 살펴보도록 하자.

우선 다음과 같은 패턴을 만들어 보자.

```
>>> import re
>>> p = re.compile('[a-zA-Z]+')
```

### match

match를 수행 해 보자.

```
>>> m = p.match("python")
>>> print(m)
<_sre.SRE_Match object at 0x01F3F9F8>
```

“python”이라는 문자열은 `[a-z]+` 정규식에 부합되므로 match object가 리턴된다.

```
>>> m = p.match("3 python")
>>> print(m)
None
```

“3 python”이라는 문자열은 처음에 나오는 3이라는 문자가 정규식 `[a-z]+`에 부합되지 않으므로 None이 리턴된다. (match는 문자열의 처음부터 정규식과 매치되는지 조사한다.)

match의 결과로 match object 또는 None이 리턴되기 때문에 보통 파이썬 정규식 프로그램은 다음과 같은 흐름으로 작성한다.

```
p = re.compile(정규표현식)
m = p.match('string goes here')
if m:
    print('Match found: ', m.group())
else:
    print('No match')
```

즉, match object의 결과가 있을 때만 그 후속 작업을 수행하겠다는 의도이다.

### search

동일한 문자열을 가지고 이번에는 search를 수행 해 보자.

```
>>> m = p.search("python")
>>> print(m)
```

```
<_sre.SRE_Match object at 0x01F3FA68>
```

“python”이라는 문자열은 match와 마찬가지로 search 역시 동일하게 매치가 된다.

```
>>> m = p.search("3 python")
>>> print(m)
<_sre.SRE_Match object at 0x01F3FA30>
```

“3 python”이라는 문자열의 첫번째 문자는 “3”이지만 search는 문자열의 처음부터 검색하는 것이 아니라 문자열 전체를 검색하기 때문에 “3” 이후의 “python”이라는 문자열과 매치하게 된다.

match와 search는 이렇듯 문자열의 처음부터 검색하는지의 여부에 따라 다르게 사용해야 한다.

- match - 문자열의 처음부터 검색
- search - 문자열내에서 일치하는것이 있는지 검색

### findall

이번에는 findall을 수행 해 보자.

```
>>> result = p.findall("life is too short")
>>> print(result)
['life', 'is', 'too', 'short']
```

“life is too short”라는 문자열의 “life”, “is”, “too”, “short”라는 단어들이 각각 [a-z]+ 정규식과 매치되어 리스트로 리턴된다.

### finditer

---

이번에는 finditer를 수행 해 보자.

```
>>> result = p.finditer("life is too short")
>>> print(result)
<callable_iterator object at 0x01F5E390>
>>> for r in result: print(r)

...
<_sre.SRE_Match object at 0x01F3F9F8>
<_sre.SRE_Match object at 0x01F3FAD8>
<_sre.SRE_Match object at 0x01F3FAA0>
<_sre.SRE_Match object at 0x01F3F9F8>
```

finditer는.findall과 동일하지만 그 결과로 iterator object를 리턴한다. iterator가 포함하는 각각의 객체는 match object이다.

### **match object**

자, 이젠 match, search등의 결과로 리턴되었던 match object에 대해서 알아보도록 하자. 우리가 정규식을 수행함으로써 궁금한 것들은 아마도 다음과 같은 것들일 것이다.

- 어떤 문자열이 매치되었는가?
- 매치된 문자열의 인덱스는 어디서부터 어디까지인가?

match object의 다음과 같은 메소드들을 이용하면 위와 같은 값들에 대해서 알 수 있다. 다음의 표를 보도록 하자.

---

method	목적
group()	매치된 문자열을 리턴한다.

---

method	목적
group()	매치된 문자열을 리턴한다.

---

**method      목적**

---

- |         |                                  |
|---------|----------------------------------|
| start() | 매치된 문자열의 시작 위치를 리턴한다.            |
| end()   | 매치된 문자열의 끝 위치를 리턴한다.             |
| span()  | 매치된 문자열의 (시작, 끝)에 해당되는 튜플을 리턴한다. |
- 

예제로 확인 해 보자.

```
>>> m = p.match("python")
>>> m.group()
'python'
>>> m.start()
0
>>> m.end()
6
>>> m.span()
(0, 6)
```

예상한대로의 결과값이 출력되는 것을 확인 할 수 있다. match에 의해 리턴된 match object의 start()의 결과값은 항상 0일수 밖에 없을 것이다. 왜냐하면 match는 항상 문자열의 시작부터 일치해야 하기 때문이다. 하지만 match가 아닌 search를 사용했다면 start()의 값은 다음과 같이 다르게 나온다.

```
>>> m = p.search("3 python")
>>> m.group()
'python'
>>> m.start()
2
```

---

```
>>> m.end()
8
>>> m.span()
(2, 8)
```

## 모듈 단위로 수행하기

지금까지 우리는 `re.compile`을 이용하여 컴파일된 패턴 객체로 그 이후 작업을 수행했다. `re` 모듈은 이 것을 좀 축약한 형태의 방법을 제공한다. 다음의 예를 보자.

```
>>> m = re.match('[a-z]+', "python")
```

위 예를 보면 컴파일과 `match`가 한번에 진행됨을 알 수 있다. 보통 한번 만든 패턴 객체를 여러번 사용해야 할 경우에는 이 방법보다 이전에 알아보았던 방법이 유리할 것이다.

## 컴파일 옵션

이전에 우리는 대소문자를 구분하지 않고 문자열을 매치하기 위해 `re.IGNORECASE`라는 컴파일 옵션을 사용했었다. 이 옵션은 `re.IGNORECASE` 옵션을 사용하는 대신 `re.I`와 같이 약어를 사용할 수 도 있다.

컴파일 옵션에는 `re.IGNORECASE` 외에도 많은 것들이 있는데 그것들에 대해서 알아보자.

- `DOTALL`, `S` - . 이 줄바꿈 문자를 포함하여 모든 문자와 매치할 수 있도록 한다.

- IGNORECASE, I - 대소문자에 관계없이 매치할 수 있도록 한다.
- MULTILINE, M - 여러줄과 매치할 수 있도록 한다. (^, \$ 메타문자의 사용과 관계가 있는 옵션이다)
- VERBOSE, X - verbose 모드를 사용할 수 있도록 한다. (정규식을 보기 편하게 만들수 있고 주석등을 사용할 수 있게된다.)

※ 위 컴파일 옵션의 콤마로 구분된 첫번째는 전체 옵션명이고 두번째는 약어를 의미한다.

## **DOTALL, S**

. 메타문자는 줄바꿈 문자(\n)를 제외한 모든 문자와 매치되는 규칙이 있다. 하지만 \n 문자도 포함하여 매치하고 싶은 경우에는 re.DOTALL 또는 re.S 옵션으로 정규식을 컴파일하면 된다.

다음의 예를 보자.

```
>>> import re
>>> p = re.compile('a.b')
>>> m = p.match('a\nb')
>>> print(m)
None
```

정규식이 a.b 인 경우 문자열 a\nb는 매치가 되지 않음을 알 수 있다. 왜냐하면 \n은 .과 매치되지 않기 때문이다. 이것이 가능하려면 다음과 같이 re.DOTALL 옵션을 사용해야 한다.

```
>>> p = re.compile('a.b', re.DOTALL)
>>> m = p.match('a\nb')
>>> print(m)
<_sre.SRE_Match object at 0x01FCF3D8>
```

보통 `re.DOTALL`은 여러줄로 이루어진 문자열에서 `\n`에 상관없이 검색하고자 할 경우에 많이 사용된다.

### `IGNORECASE, I`

`re.IGNORECASE` 또는 `re.I` 는 대소문자 구분없이 매치를 수행하고자 할 경우에 사용하는 옵션이다.

다음의 예제를 보자.

```
>>> p = re.compile('[a-z]', re.I)
>>> p.match('python')
<_sre.SRE_Match object at 0x01FCFA30>
>>> p.match('Python')
<_sre.SRE_Match object at 0x01FCFA68>
>>> p.match('PYTHON')
<_sre.SRE_Match object at 0x01FCF9F8>
```

[a-z] 정규식은 소문자만을 의미하지만 `re.I` 옵션에 의해서 대소문자에 관계 없이 매치되게 된 것이다.

### `MULTILINE, M`

`re.MULTILINE` 또는 `re.M` 옵션은 조금 후에 알아볼 메타문자인 ^, \$와 연관되어 있는 옵션이다. ^와 \$에 대해서 미리 잠시 알아보면 그 의미는 다음과 같다.

- ^ - 문자열의 처음
- \$ - 문자열의 마지막

즉 정규식이 ^python 인 경우 문자열의 처음은 항상 “python”으로 시작해야 매치되고 만약 정규식이 python\$라면 문자열의 마지막은 항상 “python”으로

끝나야 매치가 된다는 의미이다.

자, 다음의 예를 보도록 하자.

```
import re
p = re.compile("^python\s\w+")

data = """python one
life is too short
python two
you need python
python three"""

print(p.findall(data))
```

정규식 `^python\s\w+` 은 “python”이라는 문자열로 시작하고 그 후에 whitespace, 그 후에 단어가 와야한다는 의미이다. 검색할 문자열 `data`는 현재 여러 줄로 이루어져 있다.

이 스크립트를 수행하면 다음과 같은 결과가 리턴된다.

```
[‘python one’]
```

~ 메타문자에 의해 “python”이라는 문자열이 사용된 첫번째 라인만 매치가 되게 된 것이다. 하지만 정규식 ~을 문자열 전체의 처음이 아니라 각 라인별 처음으로 인식시키고 싶은 경우도 있을 것이다.

이럴 경우 사용해야 하는 옵션이 바로 `re.MULTILINE` 또는 `re.M`이다. 위 코드를 다음과 같이 수정 해 보자.

```
import re
```

---

```

p = re.compile("^python\s\w+", re.MULTILINE)

data = """python one
life is too short
python two
you need python
python three"""

print(p.findall(data))

```

`re.MULTILINE` 옵션으로 인해 ^메타문자가 문자열 전체가 아닌 라인의 처음이라는 의미를 갖게 되어 다음과 같은 결과가 출력될 것이다.

```
[ 'python one', 'python two', 'python three' ]
```

즉, `re.MULTILINE` 은 ^, \$ 메타문자가 문자열의 각 라인마다 적용될 수 있도록 해 주는 옵션이다.

## VERBOSE, X

지금껏 알아본 정규식은 매우 간단한 케이스이지만 정규식 전문가들이 만든 정규식을 보면 거의 암호수준이다. 이해하려면 하나하나 조심스럽게 뜯어보아야만 한다. 이렇게 이해하기 어려운 정규식에 주석 또는 라인단위로 구분을 할 수 있다면 얼마나 보기 좋고 이해하기 좋을까? 그것이 과연 가능할까? 물론 가능하다. 바로 `re.VERBOSE` 또는 `re.X` 옵션을 이용하면 된다.

다음의 예를 보자.

```
charref = re.compile(r'#[#](0[0-7]+|[0-9]+\|x[0-9a-fA-F]+);')
```

이해하기 쉬운가?

그럼 이제 다음의 예를 보자.

```
charref = re.compile(r"""
&[#]                      # Start of a numeric entity reference
(
    0[0-7]+                # Octal form
    | [0-9]+                # Decimal form
    | x[0-9a-fA-F]+        # Hexadecimal form
)
;
""", re.VERBOSE)
```

첫번째와 두번째의 컴파일된 패턴 객체는 모두 동일한 역할을 한다. 하지만 정규식이 복잡할 경우 두번째처럼 주석을 적고 여러줄로 표현하는 것이 훨씬 가독성이 좋다는 것을 알 수 있을 것이다.

re.VERBOSE 옵션을 사용하면 문자열에 사용된 whitespace는 컴파일 시 제거된다. (단 [] 내에 사용된 whitespace는 제외)

그리고 라인단위로 # 을 이용하여 주석문을 작성하는 것이 가능하게 된다.

## 강력한 정규표현식의 세계로

이제 남아있는 몇몇 메타문자들의 의미에 대해서 알아보고 그룹(Group)을 만드는 법, 전방탐색등의 보다 강력한 정규표현식에 대해서 알아보자.

## 메타문자

아직 알아보지 않은 메타문자들에 대해서 모두 알아보도록 하자. 앞으로 알아볼 메타문자들은 이전에 알아보았던 메타문자들과 그 성격이 조금 다르다. 이전에 알아본 메타문자들은 모두 매치되는 문자열들을 소모시킨다. 이 소모된다는 의미는 정규식 엔진이 바라보는 검색 대상 문자열의 위치가 변경됨을 의미한다.

`+, *, [], {}` 등의 메타문자는 매치가 진행될 때 현재 매치되고 있는 문자열의 위치가 변경된다. (보통 소모된다고 표현한다.) 하지만 이런것과 달리 문자열을 소모시키지 않는 메타문자들도 있다. 이번에는 이런 문자열 소모가 없는(zero-width assertions) 메타문자들에 대해서 알아보기로 하자.

|

| 메타문자는 “or”의 의미와 동일하다. `A|B` 라는 정규식이 있다면 이것은 A 또는 B라는 의미가 된다.

| 사용 시 한가지 주의해야 할 점이 있는데 그것은 다음의 예로 알아보자.

```
>>> p = re.compile('Crow|Servo')
>>> m = p.match('CroServo')
>>> print(m)
None
```

`Crow|Servo` 라는 정규식에서 |은 매우 낮은 우선순위를 갖기 때문에 ‘Crow’ 또는 ‘Servo’와는 매치되지만 위 예에서 보듯이 ‘CroServo’ (`w|S`가 더 높은 우선순위라고 가정할 경우 매치될 것 같은 문자열)는 매치되지 않는것을 확인 할 수 있다.

^

^ 메타문자는 문자열의 맨 처음과 일치함을 의미한다. 이전에 알아보았던 컴파

일 옵션 `re.MULTILINE` 을 사용할 경우에는 여러줄의 문자열에서는 각 라인의 처음과 일치하게 된다.

다음의 예를 보자.

```
>>> print(re.search('^Life', 'Life is too short'))
<_sre.SRE_Match object at 0x01FCF3D8>
>>> print(re.search('^Life', 'My Life'))
None
```

`^Life` 정규식은 “Life”라는 문자열이 처음에 올 경우에는 매치하지만 처음이 아닌 위치에 있을경우에는 매치되지 않음을 알 수 있다.

\$

\$ 메타문자는 ^ 메타문자의 반대의 경우이다. \$는 문자열의 끝과 매치함을 의미한다.

다음의 예를 보자.

```
>>> print(re.search('short$', 'Life is too short'))
<_sre.SRE_Match object at 0x01F6F3D8>
>>> print(re.search('short$', 'Life is too short, you need python'))
None
```

`short$` 정규식은 검색할 문자열의 “short”로 끝날경우 매치되지만 그 이외의 경우에는 매치되지 않음을 알 수 있다.

※ ^ 또는 \$ 문자를 메타문자가 아닌 문자 그 자체로 매치하고 싶은 경우에는 [^], [\$] 처럼 사용하거나 \^, \\$로 사용하면 된다.

\A

\A는 문자열의 처음과 매치됨을 의미한다. ^와 동일한 의미이지만 re.MULTILINE을 사용할 경우 다르게 해석된다. re.MULTILINE을 사용할 경우 ^은 라인별 문자열의 처음과 매치되지만 \A는 라인과 상관없이 전체 문자열의 처음하고만 매치된다.

\Z

\Z는 문자열의 끝과 매치됨을 의미한다. 이것 역시 \A와 동일하게 re.MULTILINE 모드에서도 \$ 메타문자와는 달리 전체 문자열의 끝과 매치된다.

\b

\b 는 단어 구분자(Word boundary)이다. 보통 단어는 whitespace에 의해 구분이 된다. 다음의 예를 보자.

```
>>> p = re.compile(r'\bclass\b')
>>> print(p.search('no class at all'))
<_sre.SRE_Match object at 0x01F6F3D8>
```

\bclass\b 정규식은 “class”라는 단어와 매치됨을 의미한다. 따라서 “no class at all”의 “class”라는 단어와 매치됨을 확인할 수 있다.

```
>>> print(p.search('the declassified algorithm'))
None
```

위 예의 “the declassified algorithm”라는 문자열 안에 “class”라는 문자열이 포함되어 있긴 하지만 whitespace로 구분된 단어가 아니므로 매치되지 않는다.

```
>>> print(p.search('one subclass is'))
None
```

“subclass”라는 문자열 역시 “class” 앞에 “sub”라는 문자열이 더해져 있으므로 매치되지 않음을 알 수 있다.

\b 메타문자를 이용할 경우 주의해야 할 점이 한가지 있다. \b는 파이썬 리터럴 규칙에 의하면 백스페이스(Back Space)를 의미하므로 백스페이스가 아닌 Word Boundary임을 알려주기 위해 r'\bclass\b' 처럼 raw string 임을 알려주는 기호인 r을 반드시 붙여주어야 한다.

\B

\B 메타문자는 \b 메타문자의 반대의 경우이다. 즉, whitespace로 구분된 단어가 아닌 경우에만 매치된다.

```
>>> p = re.compile(r'\Bclass\B')
>>> print(p.search('no class at all'))
None
>>> print(p.search('the declassified algorithm'))
<_sre.SRE_Match object at 0x01F6FA30>
>>> print(p.search('one subclass is'))
None
```

“class”라는 문자열 좌우에 whitespace가 있는 경우는 매치가 안되는 것을 확인할 수 있다.

## Grouping

만약 “ABC”라는 문자열이 계속해서 반복되는지 조사하는 정규식을 작성하고 싶다고 가정 해 보자.

어떻게 해야 할까? 지금까지 공부한 내용으로는 위 정규식을 작성할 수 없다.

이럴 때 필요한 것이 바로 그룹핑(Grouping) 이다.

위의 경우는 다음처럼 그룹핑을 이용하여 작성할 수 있다.

(ABC)+

그룹을 만들어 주는 메타문자는 바로 (과 )이다.

```
>>> p = re.compile('^(ABC)+')
>>> m = p.search('ABCABCABC OK?')
>>> print(m)
<sre.SRE_Match object at 0x01F7B320>
>>> print(m.group())
ABCABCABC
```

다음의 예를 보자.

```
>>> p = re.compile(r"\w+\s+\d+[-]\d+[-]\d+")
>>> m = p.search("park 010-1234-1234")
```

\w+\s+\d+[-]\d+[-]\d+은 이름 + " " + 전화번호 형태의 문자열을 찾는 정규표현식이다. 그런데 이렇게 매치된 문자열 중에서 이름만 뽑아내고 싶다면 어떻게 해야 할까?

반복적인 문자열을 찾는 경우에 그룹을 이용하지만 그룹을 이용하는 보다 큰 이유는 위에서 보듯이 매치된 문자열 중에서 특정 부분의 문자열만 뽑아내고 싶은 경우가 더 많다.

위 예에서 만약 “이름” 부분만을 뽑아내려 한다면 다음과 같이 할 수 있다.

```
>>> p = re.compile(r"(\w+)\s+\d+[-]\d+[-]\d+")
>>> m = p.search("park 010-1234-1234")
```

---

```
>>> print(m.group(1))
park
```

이름에 해당되는 `\w+` 부분을 그룹(`(\w+)`)으로 만들면 match object의 group(index) 메쏘드를 이용하여 그룹핑된 부분의 문자열만 뽑아내는 것이 가능하다. group 메쏘드의 index는 다음과 같은 의미를 갖는다.

- `group(0)` - 매치된 전체 문자열
- `group(1)` - 첫번째 그룹에 해당되는 문자열
- `group(2)` - 두번째 그룹에 해당되는 문자열
- ...

다음의 예제를 계속해서 보자.

```
>>> p = re.compile(r"(\w+)\s+(\d+[-]\d+[-]\d+)")
>>> m = p.search("park 010-1234-1234")
>>> print(m.group(2))
010-1234-1234
```

이번에는 전화번호 부분을 추가로 그룹(`(\d+[-]\d+[-]\d+)`)으로 만들었다. 이렇게 하면 `group(2)`처럼 사용하여 전화번호만을 뽑아낼 수 있다.

또 전화번호 중에서 국번만 뽑아내고 싶으면 어떻게 해야 할까? 다음과 같이 국번 부분을 또 그룹핑하면 된다.

```
>>> p = re.compile(r"(\w+)\s+((\d+)[-]\d+[-]\d+)")
>>> m = p.search("park 010-1234-1234")
>>> print(m.group(3))
010
```

위 예에서 보듯이 `(\w+)\s+((\d+)[-]\d+[-]\d+)` 처럼 그룹을 중첩되게 사용하는 것도 가능하다. 그룹이 중첩되어 사용되는 경우는 바깥쪽부터 시작하여 안쪽으로 들어갈수록 group index가 증가한다.

## Backreferences

그룹의 또 하나 좋은 점은 한번 그룹핑된 문자열을 재 참조할 수 있다는 점이다.

다음의 예를 보자.

```
>>> p = re.compile(r'(\b\w+)\s+\1')
>>> p.search('Paris in the the spring').group()
'the the'
```

정규식 `(\b\w+)\s+\1`은 `(그룹1) + " " + "그룹1과 동일한 단어"` 와 매치됨을 의미한다. 이렇게 정규식을 만들게 되면 두개의 동일한 단어가 연속적으로 사용되어야만 매치되게 된다. 이것을 가능하게 해 주는 것이 바로 재 참조 메타문자인 `\1`이다. `\1`은 정규식의 그룹중 첫번째 그룹을 지칭한다.

※ 두번째 그룹을 참조하려면 `\2`를 사용하면 된다.

## Named Groups

그룹은 매우 유용하지만 정규식 내에 그룹이 무척 많아 진다고 가정 해 보자. 예를 들어 정규식 내에 그룹이 10개 이상만 되어도 매우 혼란스러울 것이다. 거기에 더해 정규식이 수정되면서 그룹이 추가 삭제될 경우 그 그룹을 index로 참조했던 프로그램들도 모두 변경해 주어야 하는 위험도 갖게 된다.

만약 그룹을 index가 아닌 이름으로 참조할 수 있다면 어떨까? 그렇다면 이런 문제들에서 해방되지 않을까?

이러한 이유로 정규식은 그룹을 만들 때 그룹명을 지정할 수 있게 했다. 그 방법은 다음과 같다.

---

(?P<name>\w+)\s+((\d+)[-]\d+[-]\d+)

위 정규식은 이전에 보았던 이름과 전화번호를 추출하는 정규식이다. 기존과 달라진 부분은 다음과 같다.

- (\w+) -> (?P<name>\w+)

뭔가 대단히 복잡해 진것처럼 보이지만 (\w+)라는 그룹에 “name”이라는 이름을 지어 준 것에 불과하다. 여기서 사용한 (?)... 표현식은 정규표현식의 확장구문이다. (여기서 ...은 다양하게 변한다는 anything의 의미이다.) 이 확장구문을 이용하기 시작하면 가독성이 무척 떨어지지만 그에 더해 강력함을 갖게 되는 것이다. 앞으로 알아볼 “전방탐색” 역시 이 확장구문을 이용해야만 한다.

그룹에 이름을 지어주기 위해서는 다음과 같은 확장구문을 사용해야 한다.

- (?P<그룹명>...)

참고로 전방탐색의 확장구문은 다음과 같다. (조금 후에 자세히 알아볼 것이다.)

- (?=...) - 긍정형 전방탐색
- (?!...) - 부정형 전방탐색

그룹에 이름을 지정하고 참조하는 다음의 예를 보자.

```
>>> p = re.compile(r"(?P<name>\w+)\s+((\d+)[-]\d+[-]\d+)")
>>> m = p.search("park 010-1234-1234")
>>> print(m.group("name"))
park
```

위 예에서 보듯이 “name”이라는 그룹명으로 참조가 가능함을 확인 할 수 있다.

그룹명을 이용하면 정규식 내에서 재 참조역시 가능하다.

```
>>> p = re.compile(r'(?P<word>\b\w+)\s+(?P=word)')
```

---

```
>>> p.search('Paris in the the spring').group()
'the the'
```

위 예에서 보듯이 재 참조시에는 (?P=그룹명)이라는 확장구문을 이용해야 한다.

## Lookahead Assertions

정규식에 막 입문한 사람들이 가장 어려워 하는 것이 바로 전방탐색 확장구문이다. 정규식 안에 이 확장구문이 사용되면 금방 암호문으로 바뀌어 버리기 때문이다. 하지만 이 전방탐색이 꼭 필요한 경우가 있고 매우 유용하게 사용이 되는 경우가 많으니 꼭 알아두도록 하자.

다음의 예제를 보자.

```
>>> p = re.compile(".+:")
>>> m = p.search("http://google.com")
>>> print(m.group())
http:
```

정규식 .+: 과 일치하는 문자열로 “http:”가 리턴되었다. 하지만 “http:”라는 검색 결과에서 “:”을 제거하고 싶다면 어떻게 해야 할까? 그리고 위 예는 간단하지만 훨씬 복잡한 정규식이어서 그룹핑은 추가로 할 수 없다는 조건이 더해진다면 어떻게 해야 할까?

이럴 때 사용할 수 있는 것이 바로 “전방탐색”이다. “전방탐색”에는 긍정(Positive)과 부정(Negative)의 두 가지 종류가 있고 다음과 같이 표현된다.

- (=...) - 긍정형 전방탐색, ...에 해당되는 정규식과 매치되어야 하며 조건이 통과되어도 문자열이 소모되지 않는다.

- (?!...) - 부정형 전방탐색, ...에 해당되는 정규식과 매치되지 않아야 하며 조건이 통과되어도 문자열이 소모되지 않는다.

긍정형 전방탐색을 이용하여 우리는 “http:”의 결과를 “http”로 바꿀 수 있다. 다음의 예를 보자.

```
>>> p = re.compile(".+(?=:)")
>>> m = p.search("http://google.com")
>>> print(m.group())
http
```

정규식 중 :에 해당하는 부분이 긍정형 전방탐색 기법이 적용되어 (?=:) 으로 변경되었다. 이렇게 되면 기존 정규식과 검색에서는 동일한 효과를 발휘하지만 “:”에 해당되는 문자열이 정규식 엔진에 의해 소모되지 않아 검색결과에서는 “:” 이 제거되어 리턴되는 효과가 있다.

자, 이번에는 다음의 정규식을 보자.

.\*[.] .\*\$

이 정규식은 파일명 + '.' + 확장자 를 나타내는 정규식이다. 이 정규식은 foo.bar, autoexec.bat, sendmail.cf등과 매치할 것이다.

자, 이제 이 정규식에 다음과 같은 조건을 추가 해 보자.

“bat 파일은 제외해야 한다.”

가장 먼저 생각할 수 있는 정규식은 다음과 같을 것이다.

.\*[.] [^b] .\*\$

위 정규식은 확장자가 “b” 라는 문자로 시작하면 안된다는 의미이다. 하지만 이 정규식은 foo.bar라는 파일마저 걸러내 버린다. 다시 정규식을 다음과 같이 수

---

정 해 보자.

`.*[.]([`b]...|.`^a]..|[`t])$`

위 정규식은 확장자의 첫번째 문자가 “b” 또는 두번째 문자가 “a” 또는 세번째 문자가 “t”가 아닌경우를 의미한다. 이 정규식에 의하여 foo.bar는 제외되지 않고 autoexec.bat은 제외되어 만족스러운 결과를 보여준다. 하지만 이 정규식은 어렵게도 sendmail.cf 처럼 확장자의 길이가 2인 케이스를 걸러내는 오동작을 하기 시작한다.

자, 이제 다음과 같이 바꾸어야 한다.

`.*[.]([`b].??.|.`^a]??.|..?[^t]?)$`

확장자의 갯수가 2개여도 통과되는 정규식을 만든것이다. 하지만 정규식은 점점 더 복잡해져가고 이해하기 어려워진다. 자, 그런데 bat 말고 exe파일도 제외하라는 조건이 추가로 생긴다면 어떻게겠는가? 이걸 구현하기 위해서 패턴은 더욱더 복잡해져야만 할 것이다.

이러한 상황의 구원투수는 바로 “부정형 전방탐색”이다.

위 케이스는 “부정형 전방탐색”을 사용하면 다음과 같이 간단하게 처리된다.

`.*[.](?!bat$).*$`

확장자가 “bat”가 아닌 경우에만 통과된다는 의미이다. “bat”라는 문자열이 있는지 조사하는 과정에서 문자열이 소모되지 않으므로 “bat”가 아니라고 판단되면 그 이후 정규식 매칭이 진행된다. “exe” 역시 제외하라는 조건이 추가되더라도 다음과 같이 간단하게 표현이 된다.

`.*[.](?!bat$|exe$).*$`

## 문자열 바꾸기

`sub`를 이용하면 정규식과 매치되는 부분을 다른 문자로 쉽게 바꿀 수 있다.

다음의 예를 보자.

```
>>> p = re.compile('blue|white|red')
>>> p.sub('colour', 'blue socks and red shoes')
'colour socks and colour shoes'
```

`sub`메쏘드의 첫번째 파라미터는 “바꿀 문자열(replacement)”이 되고 두번째 입력은 “대상 문자열”이 된다. 위 예에서 보듯이 `blue` 또는 `white` 또는 `red`라는 문자열이 `colour`라는 문자열로 바뀌는 것을 확인 할 수 있다.

자, 그런데 바꾸기는 바꾸는데 딱 한번만 바꾸고 싶은 경우도 있을 것이다. 이렇게 바꾸기 횟수를 제어하려면 다음과 같이 세번째 파라미터인 `count` 값을 넘기면 된다.

```
>>> p.sub('colour', 'blue socks and red shoes', count=1)
'colour socks and red shoes'
```

처음 일치하는 “`blue`”만 “`colour`”라는 문자열로 1회만 바꾸기가 발생함을 알 수 있다.

`subn` 역시 `sub`와 동일하지만 리턴되는 결과가 조금 다르다. `subn`은 리턴결과를 튜플로 첫번째 요소는 변경된 문자열을 두번째 요소로 바꾸기가 발생한 횟수를 담아서 리턴한다.

```
>>> p = re.compile('blue|white|red')
>>> p.subn('colour', 'blue socks and red shoes')
('colour socks and colour shoes', 2)
```

sub 사용 시에 참조구문을 사용할 수 있다. 다음의 예를 보자.

```
>>> p = re.compile(r"(?P<name>\w+)\s+(?P<phone>(\d+)[-]\d+[-]\d+)")
>>> print(p.sub("\g<phone> \g<name>", "park 010-1234-1234"))
010-1234-1234 park
```

위 예는 이름 + 전화번호의 문자열을 전화번호 + 이름으로 바꾸는 예이다. sub의 바꿀 문자열 부분에 \g<그룹명>을 이용하면 정규식의 그룹명을 참조할 수 있게된다.

다음과 같이 그룹명 대신 참조번호를 이용해도 마찬가지 결과가 리턴된다.

```
>>> p = re.compile(r"(?P<name>\w+)\s+(?P<phone>(\d+)[-]\d+[-]\d+)")
>>> print(p.sub("\g<2> \g<1>", "park 010-1234-1234"))
010-1234-1234 park
```

sub 의 첫번째 파라미터로 함수 또한 가능하다. 다음의 예를 보자.

```
>>> def hexrepl(match):
...     "Return the hex string for a decimal number"
...     value = int(match.group())
...     return hex(value)
...
>>> p = re.compile(r'\d+')
>>> p.sub(hexrepl, 'Call 65490 for printing, 49152 for user code.')
'Call 0xffffd2 for printing, 0xc000 for user code.'
```

hexrepl 함수는 match object(위 예에서 숫자에 매치되는)를 입력으로 받아 십

육진수로 변환하여 리턴하는 함수이다. `sub`의 첫번째 파라미터로 함수를 사용할 경우 해당 함수의 첫번째 파라미터에는 정규식과 매치된 match object가 입력된다. 그리고 매치되는 문자열은 함수의 리턴값으로 바뀌게 된다.

## Greedy vs Non-Greedy

다음의 예제를 보자.

```
>>> s = '<html><head><title>Title</title>'  
>>> len(s)  
32  
>>> print(re.match('<.*>', s).span())  
(0, 32)  
>>> print(re.match('<.*>', s).group())  
<html><head><title>Title</title>
```

`<.*>` 정규식의 매치결과로 `<html>` 문자열이 리턴되기를 기대했지만 \* 메타문자는 매우 탐욕스럽기 때문에 `<html><head><title>Title</title>` 문자열을 모두 소모시켜 버렸다. 어떻게 하면 이 탐욕스러움을 제한하고 `<html>`이라는 문자열까지만 소모되도록 막을 수 있을까?

다음과 같이 non-greedy 문자인 `?`을 사용하면 \*의 욕심을 억제시킬 수 있다.

```
>>> print(re.match('<.*?>', s).group())  
<html>
```

non-greedy 문자인 `?`은 `*?`, `+?`, `??`, `{m,n}?`과 같이 사용이 가능하며 가능한한 가장 적은 반복을 수행하도록 도와주는 역할을 한다.

이상과 같이 파이썬 정규표현식에 대해서 알아보았다.

## 2) XML 처리

파이썬으로 XML문서를 다루는 방법에 대해서 알아보자.

XML처리를 위한 파이썬 라이브러리들은 아주 많다. XML처리를 위한 대부분의 라이브러리는 아래문서에서 확인 할 수 있다.

<http://wiki.python.org/moin/PythonXml>

### ElementTree

이 곳에서는 가장 많은 사람들의 사랑을 받고 있는 ElementTree를 사용하는 방법에 대해서 다룬다.

ElementTree는 외부라이브러리로 존재하다가 파이썬 2.5부터 통합되었다. ElementTree는 Tkinter로 잘 알려진 프레드릭(Fredrik Lundh)에 의해서 만들어진 XML 제너레이터 & 파서이다.

### XML문서 생성하기

ElementTree를 이용하여 다음과 같은 구조의 XML문서를 생성해 보자.

```
<note date="20120104">
    <to>Tove</to>
    <from>Jani</from>
    <heading>Reminder</heading>
    <body>Don't forget me this weekend!</body>
</note>
```

먼저 다음과 같은 소스를 작성해 보자:

```
from xml.etree.ElementTree import Element, dump

note = Element("note")
to = Element("to")
to.text = "Tove"

note.append(to)
dump(note)
```

위 소스의 실행 결과는 다음과 같다:

```
<note><to>Tove</to></note>
```

위와같이 Element를 이용하면 태그를 만들 수 있고 만들어진 태그에 텍스트값을 추가할 수 있음을 알 수 있다.

SubElement를 이용하면 조금 더 편리하게 태그를 추가할 수 있다:

```
from xml.etree.ElementTree import Element, SubElement, dump

note = Element("note")
to = Element("to")
to.text = "Tove"

note.append(to)
SubElement(note, "from").text = "Jani"

dump(note)
```

결과는 다음과 같다:

```
<note><to>Tove</to><from>Jani</from></note>
```

SubElement는 태그명과 태그의 텍스트값을 한번에 설정할 수 있다.

다음과 같이 태그사이에 태그를 추가하거나 특정 태그를 삭제할 수도 있다:

```
dummy = Element("dummy")
note.insert(1, dummy)
note.remove(dummy)
```

위의 예는 dummy라는 태그를 삽입하고 삭제하는 것을 보여준다.

이번에는 note 태그에 attribute를 추가 해 보도록 하자:

```
from xml.etree.ElementTree import Element, SubElement, dump

note = Element("note")
to = Element("to")
to.text = "Tove"

note.append(to)
SubElement(note, "from").text = "Jani"
note.attrib["date"] = "20120104"

dump(note)
```

note.attrib[“date”]와 같은 방법으로 어트리뷰트값을 추가할 수 있었다.

다음과 같이 Element 생성 시 직접 attribute값을 추가하는 방법을 사용해도 된다:

```
note = Element("note", date="20120104")
```

결과는 다음과 같다:

```
<note date="20120104"><to>Tove</to><from>Jani</from></note>
```

이상과 같이 XML 태그와 어트리뷰트를 추가하는 방법에 대해서 알아보았다.

완성된 버전은 다음과 같다:

```
from xml.etree.ElementTree import Element, SubElement, dump

note = Element("note")
note.attrib["date"] = "20120104"

to = Element("to")
to.text = "Tove"
note.append(to)

SubElement(note, "from").text = "Jani"
SubElement(note, "heading").text = "Reminder"
SubElement(note, "body").text = "Don't forget me this weekend!"
dump(note)
```

결과는 다음과 같다:

```
<note date="20120104"><to>Tove</to><from>Jani</from>...생략...</note>
```

위 결과값은 한줄로 이어져 있어 보기 가 쉽지 않다. 정렬된 xml 값을 보기 쉽게 하기 위해 다음과 같이 indent 함수를 이용 해 보자:

```
from xml.etree.ElementTree import Element, SubElement, dump

note = Element("note")
note.attrib["date"] = "20120104"

to = Element("to")
to.text = "Tove"
note.append(to)

SubElement(note, "from").text = "Jani"
SubElement(note, "heading").text = "Reminder"
SubElement(note, "body").text = "Don't forget me this weekend!"

def indent(elem, level=0):
    i = "\n" + level*"  "
    if len(elem):
        if not elem.text or not elem.text.strip():
            elem.text = i + "  "
        if not elem.tail or not elem.tail.strip():
            elem.tail = i
        for elem in elem:
            indent(elem, level+1)
        if not elem.tail or not elem.tail.strip():
            elem.tail = i
```

```
else:  
    if level and (not elem.tail or not elem.tail.strip()):  
        elem.tail = i  
  
    indent(note)  
    dump(note)
```

indent 함수를 이용하면 다음과 같이 정렬된 형태의 xml을 확인 할 수 있을 것이다:

```
<note date="20120104">  
  <to>Tove</to>  
  <from>Jani</from>  
  <heading>Reminder</heading>  
  <body>Don't forget me this weekend!</body>  
</note>
```

이제 생성한 xml을 다음과 같이 Element의 write라는 메써드를 이용하여 파일에 write 해 보도록 하자:

```
ElementTree(note).write("note.xml")
```

note.xml이 생성되는 것을 확인 할 수 있을 것이다

## XML문서 파싱하기

이번에는 XML 생성과는 반대로 생성된 XML문서를 파싱하고 검색하는 방법에 대해서 알아보자.

XML문서는 다음과 같은 방법으로 파싱한다:

```
from xml.etree.ElementTree import parse
tree = parse("note.xml")
note = tree.getroot()
```

ElementTree의 parse라는 함수를 이용하여 xml을 파싱할 수 있다.

어트리뷰트 값은 다음과 같이 읽을 수 있다:

```
print note.get("date")
print note.get("foo", "default")
print note.keys()
print note.items()
```

결과는 다음과 같다:

```
20120104
default
['date']
[('date', '20120104')]
```

get 메써드는 어트리뷰트의 값을 읽는다. 만약 두번째 파라미터로 디폴트 값을 주었다면 첫번째 인자에 해당되는 어트리뷰트 값이 없을 경우 두번째 값을 리턴한다. keys 는 모든 어트리뷰트의 키값을 리스트로 리턴한다. items 는 key, value 쌍을 리턴한다. 어트리뷰트 값을 가져오는 방법은 덕셔너리와 동일함을 알 수 있다.

XML 태그에 접근하는 방법은 다음과 같다:

---

```
from_tag = note.find("from")
from_tags = note.findall("from")
from_text = note.findtext("from")
```

- note.find("from")은 note태그 하위에 from과 일치하는 첫번째 태그를 찾아서 리턴한다. 없으면 None을 리턴한다.
- note.findall("from")은 note태그 하위에 from과 일치하는 모든 태그를 리스트로 리턴한다.
- note.findtext("from")은 note태그 하위에 from과 일치하는 첫번째 태그의 텍스트값을 리턴한다.

특정 태그의 모든 하위 엘리먼트를 순차적으로 처리할 때는 아래의 메써드를 사용한다:

```
childs = note.getiterator()
childs = note.getchildren()
```

getiterator() 함수는 첫번째 인수로 다음과 같이 태그명을 받을 수도 있다:

```
note.getiterator("from")
```

위와 같은 경우 from태그의 하위 엘리먼트들이 순차적으로 리턴된다.

보통 다음과 같이 많이 사용된다:

```
for parent in tree.getiterator():
    for child in parent:
        ... work on parent/child tuple
```

이상과 같이 파이썬에서 XML을 처리하기 위한 ElementTree의 사용방법에 대해서 간략히 알아보았다. 이보다 더 쉬운 XML 라이브러리가 있을까? 있다면 이곳에 소개시켜 주시기를.. ^^

Fredrik Lundh 가 직접 작성한 튜토리얼을 꼭 읽어보도록 하자.

<http://effbot.org/zone/element.htm>

## A. 부록

부록 챕터에서는 다음과 같은 내용을 다룬다.

- 파이썬 2.7 과 파이썬 3
- 파이썬과 에디터
- 파이썬과 다른 언어들과의 비교
- 유용한 파이썬 문서들

## 1) 파이썬 2.7 vs 파이썬 3

이 책은 파이썬 2.7 버전과 3 버전에 상관없이 책을 읽을 수 있도록 작성되어졌다. 아래의 몇 가지 큰 차이점만 숙지하면 파이썬 버전에 상관없이 이 책의 예제를 수행하는 데 무리가 없을 것이다.

### print

파이썬 3 버전은 출력할 문자열에 괄호를 필요로 한다.

파이썬 3 버전의 예

```
print ("Hello Python")
```

파이썬 2.7 버전의 예

```
print "Hello Python"
```

파이썬 2.7 버전인 경우 파이썬 3 버전처럼 괄호를 사용해도 동일하게 동작한다.  
(단, 2.7 버전 이하의 파이썬 구버전에서는 오류가 발생할 수 있다.)

### 줄바꿈 방지

print 문 실행 시 항상 문자열 마지막에 \n 문자가 출력되어 줄바꿈이 일어나게 된다. 이렇게 마지막 문자인 \n을 생략할 수 있는 방법이 있는데 이것또한 파이썬 3 버전과 파이썬 2.7 버전이 서로 다르다.

파이썬 3 버전의 예

```
print ("No new line", end=" "); print ("ok?")
```

---

파이썬 3 버전의 경우 줄바꿈 문자를 제거하기 위해서 위와 같이 끝 문자를 지정할 수 있는 end 파라미터를 설정하면 된다. 지정하지 않으면 디폴트로 \n 문자가 세팅된다.

#### 파이썬 2.7 버전의 예

```
print "No new line",;print "ok?"
```

파이썬 2.7 버전의 경우 줄바꿈 문자를 제거하기 위해서 문자열의 끝에 콤마(,)를 덧붙이면 된다.

### 자동 형 변환

파이썬 3의 경우 숫자연산 시 자동으로 형 변환이 된다.

#### 파이썬 3 버전의 예

```
>>> 3 / 4  
0.75
```

#### 파이썬 2.7 버전의 예

```
>>> 3 / 4  
0  
>>> 3 / 4.0  
0.75
```

## input

파이썬 3 버전의 `input` 내장함수와 파이썬 2.7버전의 `raw_input` 내장함수는 동일하다. 기존 파이썬 2.7의 `input` 내장함수는 파이썬 3부터는 더이상 지원되지 않는다.

파이썬 3 버전의 예

```
>>> name = input("이름을 입력하세요:")
```

파이썬 2.7 버전의 예

```
>>> name = raw_input("이름을 입력하세요:")
```

## 소스코드 인코딩

파이썬 3 버전부터는 utf-8이 기본 소스코드 인코딩이므로 다음과 같은 문자열을 소스코드 첫줄에서 생략할 수 있다.

```
# -*- coding: utf-8 -*-
```

하지만 utf-8 이 아닌 다른 형태의 소스코드 인코딩을 사용해야 할 경우에는 해당 인코딩을 명시해야 한다. 하지만 파이썬 2.7 버전은 무조건 위와 같은 문자열을 소스코드 첫 줄에 명시해야만 인코딩 오류가 발생하지 않는다.

## 에러처리

try ... except... 에러 처리 시 에러 변수명을 표기하는 방식이 파이썬 버전 3과 버전 2.7이 서로 다르다.

파이썬 3 버전의 예

```
try:  
    4 / 0  
except ZeroDivisionError as e:  
    print(e)
```

파이썬 2.7 버전의 예

```
try:  
    4 / 0  
except ZeroDivisionError, e:  
    print e
```

에러변수 설정 시 파이썬 3 버전은 as를 2.7 버전은 콤마(,)를 사용한다.

## 2) 파이썬과 에디터

파이썬 코드를 작성하기 위해서 어떤 에디터를 선택할 것인가? 아마도 자신에게 익숙한 에디터를 선택하는 것이 정답이겠지만 아직 마땅히 사용할 만한 에디터가 없는 독자에게 필자는 몇 가지 추천하고픈 에디터가 있다. 윈도우즈 사용자라면 에디트 플러스를, 리눅스 사용자라면 당연히 VI에디터를 추천한다. 물론 리눅스에는 이맥스라는 좋은 에디터가 있긴 하지만 초보자에겐 어울리지 않을 듯 하다.

여기서는 에디트 플러스를 설치하고 파이썬 사용에 필요한 몇 가지 설정 사항에 대해서 알아보기로 하자.

### 에디트 플러스(Edit Plus)

이 프로그램은 무료 소프트웨어는 아니기 때문에 평가판을 이용해야 한다. 다운 받은 후부터 한달 간 쓸 수 있다.

<http://www.editplus.com/kr>

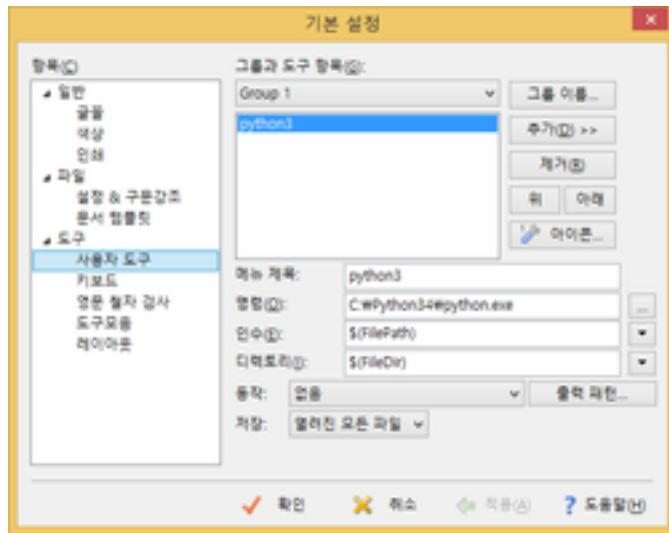
위 URL에서 파일을 다운로드하고 설치를 하자. 이제 파이썬을 에디트 플러스에서 사용하기 위해 어떻게 설정을 해야하는지 보도록 하자. 에디트 플러스를 실행시키자. 평가판은 “동의함” 버튼을 눌러야만 쓸 수 있다. 에디트 플러스에서 파이썬을 편하게 사용하기 위한 몇가지 설정을 먼저 하도록 하자.

### 파이썬 설정

우선 에디트 플러스를 실행시키고 메뉴바에서 [도구 -> 기본설정]을 선택하자.

그리고 [도구 -> 사용자도구]를 선택하자.

다음과 같은 화면을 볼 수 있을 것이다.



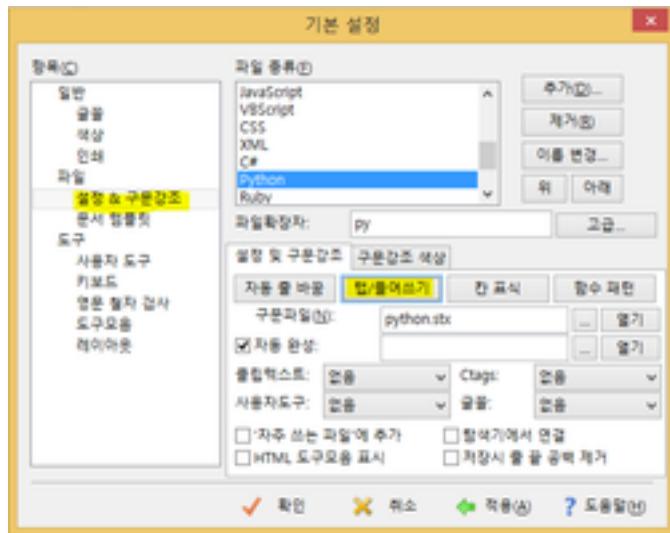
위의 창처럼 내용이 채워지도록 다음과 같이 수정해야 한다.

1. [추가 -> 프로그램]을 선택한다.
2. “메뉴제목”에는 “python3” 라고 입력한다.
3. “명령”에는 파이썬 프로그램(C:\Python34\python.exe)을 선택한다.
4. “인수”에는 우측 선택 중 첫 번째 항목인 “파일경로\$(FilePath)”를 선택한다.
5. “디렉토리”에는 우측 선택 중 첫 번째 항목인 “파일디렉토리\$(FileDir)”를 선택한다.
6. 하단의 “적용” 버튼을 클릭하여 설정을 저장한다.

이제 에디트 편집으로 test.py와 같은 프로그램을 작성하고 저장한 후 Ctrl+1 을 실행하면 test.py 프로그램이 자동으로 실행되는 것을 확인할 수 있을 것이다.

## 들여쓰기 설정

들여쓰기로 “탭”을 사용하더라도 항상 4개의 스페이스로 변환되도록 하기위해 다음과 같이 설정하도록 하자.

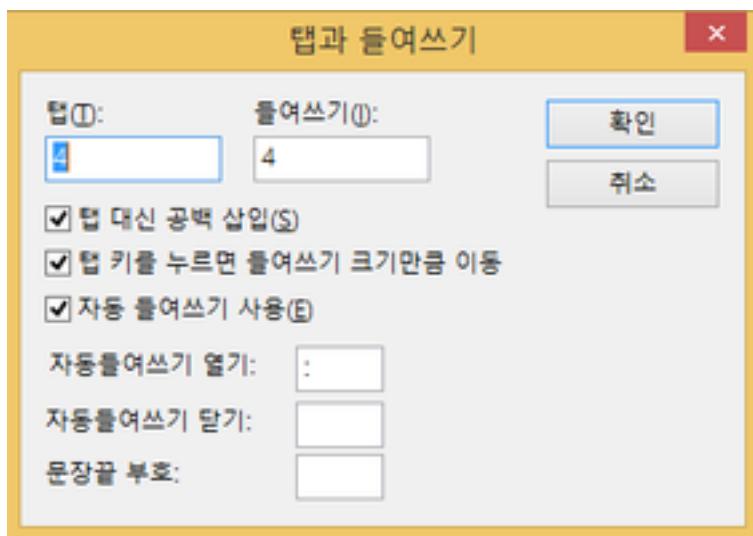


1. [파일 -> 설정과 구문강조 -> python] 선택
2. “탭/들여쓰기” 클릭

“탭/들여쓰기”를 클릭하면 다음과 같은 팝업 창이 나타난다.

1. “탭”과 “들여쓰기” 항목에는 “4”를 입력
2. “탭 대신 공백 삽입” 체크
3. “탭 키를 누르면 들여쓰기 크기만큼 이동” 체크
4. “자동 들여쓰기 사용” 체크

위와 같이 수정한 후 에디터 창에서 탭키를 이용하여 들여쓰기를 하면 위에서 설정한데로 탭 대신 4개의 스페이스로 변환되어 들여쓰기가 되는 것을 확인 할



수 있을 것이다.

### 3) 파이썬과 다른 언어들과의 비교

Guido van Rossum 저. (이강성 역)

Python은 흔히 Java, JavaScript, Perl, Tcl 또는 Samlltalk와 같은 인터프리터 언어와 비교된다. C++, Common Lisp 그리고 Scheme과 같은 언어와도 비교되 부각된다. 이 절에서 난 이들 언어를 간단히 비교할 것이다. 이 비교는 언어적인 측면에서만 다룬다. 실제적으로, 프로그래밍 언어의 선택은 다른 실세계 제약(비용, 유용성, 학습 그리고 선행 투자비용 혹은 감성적인 친근감까지도)에 의해 자주 언급되기도 한다. 이러한 면들은 아주 가변적이고, 이러한 면으로 비교를 한다는 것은 시간 낭비에 가깝다.

#### Java (자바)

일반적으로 Python 프로그램은 Java 프로그램 보다는 느리게 수행된다. 하지만 Python 프로그램은 개발하는 시간이 훨씬 적게 걸린다. Python 프로그램은 Java 프로그램보다도 3-5배 정도 코드가 짧다. 이 차이는 Python의 내장 고수준 데이터 형과 동적인 형결정 기능에서 기인한다고 생각한다. 예를 들면, Python 프로그래머는 인수나 변수의 형을 선언하는데 시간을 허비하지 않고, 구문적인 지원이 언어안에 내장되어 있는 Python의 강력한 다형질의 리스트(polymorphic list)와 사전 형은 거의 모든 Python 프로그램에서 유용하게 활용되고 있다. 실행시간 형결정으로 인해서 Python의 실행시간에 Java가 하는 것보다 좀더 많은 일을 한다. 예를 들면,  $a+b$ 와 같은 식을 계산할 때, 먼저 컴파일시에 알려지지 않은  $a$ 와  $b$  객체를 검사하여 그들의 형을 알아내야 한다. 그리고 나서 적절한 덧셈 연산을 호출한다. 그 덧셈 연산은 객체에 따라 사용자에 의해 오버로드(overloaded)된 것일 수 있다. 반면에, Java는 효과적인 정수형, 실수형 덧셈을 한다. 하지만  $a$ 와  $b$ 의 변수선언을 요구하고, + 연산자에 대한 사용자 정의 연산자 오버로딩을 허용하지 않는다. 이러한 이유들로, Python은 ‘겹착’ 언어로서 아주 적당한 반면, Java는 저수준 구현 언어로 특성화 지

을 수 있다. 사실 이 두 언어는 아주 훌륭한 조합을 이룬다. Java에서 개발된 요소(components)들이 Python에서 활용된다; Python 역시 Java로 구현되기 전에 그 프로토타입을 정하는데 활용된다. 이러한 형의 개발을 지원하기 위해, Java로 쓰여진 Python 구현(implementation)이 개발중이다. 이것은 Java에서 Python을 호출하고 그 반대도 가능하게 해준다. 이 구현으로, Python 소스코드는 Java 바이트코드로 (Python의 동적 의미를 지원하기 위한 실행시간 라이브러리의 도움으로)번역된다.

## Javascript (자바스크립트)

Python의 ‘객체기반’ 부분 집합이 대략 JavaScript와 동일하다. JavaScript와 같이 (그러나 Java와는 다르게), Python은 클래스안에 정의하지 않아도 되는, 단순한 함수와 변수를 사용하는 프로그래밍 스타일을 지원한다. 하지만 JavaScript는 이것이 지원하는 전부이다. Python은, 반면에, 훨씬 큰 프로그램을 클래스와 상속이 중요한 역할을 하는 진정한 객체 지향 프로그래밍 스타일을 통하여 더 좋은 코드 재사용을 하도록 지원한다.

## Perl (펄)

Python과 Perl은 비슷한 배경에서 개발되었다(유닉스 스크립트언어에서 성장했다). 그리고 많은 비슷한 기능을 지원한다. 그러나 철학은 다르다. Perl은 보편적인 응용지향 태스크를 지원하는데 중심을 두었지만 (예:내장 정규식 표현, 파일 스캐닝과 보고서 생성 기능들), Python은 보편적인 프로그래밍 방법론 (자료구조 설계 및 객체지향 프로그래밍)을 지원한다. 그리고 프로그래머가 우아하고(elegant) 암호같지 않은 코드를 통해 읽기 쉽고 관리하기 쉽도록 한다. 결과적으로, Python이 Perl과 가깝지만 그 원래 응용 영역을 침범하는 일은 많지 않다. 하지만 Python은 Perl의 적합한 응용분야 외에 많은 부분에서 적용성을 갖는다.

## Tcl (티클)

Python과 같이 Tcl은 독립적인 프로그래밍 언어 뿐 아니라, 응용 확장언어(extension language)로도 사용된다. 하지만, 전통적으로 모든 데이터를 문자열로 처리하는 Tcl은 자료구조에 약하고 Python 보다 실행에 시간이 많이 걸린다. Tcl은 또한 모듈의 이름영역(name space)과 같은, 큰 프로그램을 쓰기에 적합한 특징들을 가지고 있지 않다. 따라서 Tcl을 사용하는 전형적인 큰 응용 프로그램은 특별히 그 응용에 필요한 C나 C++로 확장된 부분을 갖는다. 이에 반해서 Python 응용 프로그램은 '순수한 Python'으로만 흔히 기술된다. 물론 순수한 Python을 이용한 개발은 C나 C++부분을 쓰고 디버깅하는 것보다도 훨씬 빠르다. Tcl의 결점은 매우는 부분이 Tk 툴킷이다. Python은 Tk을 표준 GUI 라이브러리로 쓰도록 적용했다. Tcl 8.0은 바이트코드를 도입하여 빠른 처리를 했고, 제한된 데이터 형 지원과 이름영역을 지원한다고 하지만 여전히 거추장스러운 프로그래밍 언어이다.

## Smalltalk (스몰토크)

아마도 Python과 Smalltalk의 가장큰 차이는 Python이 보다 더 '주류(mainstream)' 구문을 가진다는 것이다. Python은 Smalltalk과 같이 동적인 형결정과 결합(binding)을 한다. Python의 모든 것은 객체이다. 하지만, Python은 내장 객체 형과 사용자 정의 클래스를 구별하고, 내장 형으로부터의 상속은 현재로서 허용하지 않는다. Smalltalk의 데이터 타입의 표준 라이브러리 모음은 훨씬 섬세한 반면, Python의 라이브러리는 인터넷과 WWW 세계 (email, HTML, FTP등)에 적응하기 좋은 많은 기능을 제공한다. Python은 개발환경과 코드 배포에 있어서 다른 철학을 갖는다. Smalltalk이 환경과 사용자 프로그램을 포함하는 통일된 '시스템 이미지'를 갖는데 반해, Python은 표준 모듈과 사용자 모듈을 다른 파일에 저장하여 쉽게 재배열되고 시스템 밖으로 배포될 수 있다. 한 결과를 예를 들면, GUI가 시스템 안에서 설계된 것이 아니므로, GUI를 붙이기 위한 한가지 이상의 선택이 Python 프로그램에 있다.

## C++

Java에 대해서 이야기 한 대부분이 C++에도 적용된다. Python코드가 Java 코드보다 3-5배 짧으며, C++코드에 비해 5-10배 짧다!! 일 예로 한명의 Python 프로그래머는 C++프로그래머 두 명이 1년에 끝낼 수 없는 일을 두달만에 끝낼 수 있다. Python은 C++로 쓰여진 코드를 사용하는 접착 언어로 빛을 발한다.

## Common Lisp and Scheme

이들 언어는 동적인 의미해석에서 Python에 가깝다. 그러나 구문해석 접근은 너무 달라서 매우 심한 논쟁거리가 될 만한 비교가 된다: Lisp의 구문적인 부족 함이 장점일까 단점일까? Python은 Lisp과 같은 내성적인 능력(capabilities)이 있고, Python 프로그램은 아주 쉽게 프로그램 부분을 구성해서 실행할 수 있다는 것을 밝혀야겠다. 일반적으로, 실세계 실체가 결정적이다: Common Lisp은 크다(어떠한 관점에서도 그렇다). Scheme 세계는 많은 어울리지 않는 버전들로 나누어져있다. 그에 반해서 Python은 하나이고, 무료이고, 작게 구현되었다. 더 자세한 Scheme와의 비교에 관해선 Moshe Zadka가 쓴 Python vs. Scheme을 보라.

## 4) 유용한 파이썬 문서들

파이썬을 제대로 알고 잘 활용하기 위해서는 직접 많은 프로그램들을 만들어 보는 것도 중요하지만 관련 서적과 참고 문헌들을 언제나 가까이 하고 읽는 것 또한 매우 중요한 일일 것이다. 이 책을 읽고 난 후에 다음의 문서들을 읽어본다면 독자는 한층 더 깊이있는 파이썬 프로그래머가 될 것이다.

### 파이썬 튜토리얼

파이썬을 만든 귀도가 직접 만든 문서로서 파이썬 배포본에 함께 들어있는 문서이다. 튜토리얼인만큼 쉽게 다가설 수 있지만 내용이 매우 학축적이어서 초보자가 얼른 이해하기 힘든 부분이 많이 있다. 그리고 무엇보다도 영문이기 때문에 어려움이 많다. 한국 파이썬 사용자 모임인 파이썬 정보광장에서 이만용씨가 튜토리얼 문서를 번역한 것을 찾아 볼 수도 있다. 꼭 한번은 읽어야만 될 문서이다.

### 파이썬 라이브러리 레퍼런스

이 문서의 중요성을 새삼 거론할 필요는 없을 것 같다. 이 문서 없이는 제대로 된 파이썬 프로그램을 작성할 수가 없다. 항상 가까이에 두고 늘 찾아보아야 하는 문서이다. 이 문서를 처음부터 끝까지 대충이라도 꼭 한번 읽어보도록 하자. 어떤 모듈들이 있는지 조차 모른다면 프로그래밍을 할 때 많은 어려움을 겪게 될 것이다. 또한 프로그래밍을 할 때 어떤 모듈을 사용할 것인지에 대한 감각을 익히기 바란다.

## 파이썬 하우투들

<http://www.python.org/doc/howto> 에 가면 파이썬 하우투 문서들을 찾아 볼 수가 있다. 특정한 주제에 대해서 라이브러리 레퍼런스의 내용보다 자세한 설명을 해 준다. 다음과 같은 문서들이 있다.

### Advocacy

파이썬을 사용하는 이유에 대한 토론, 파이썬으로 하기에 적당한 일과 그렇지 않은 것들에 대한 이야기등.

### Curses Programming with Python

파이썬과 Curses를 이용한 텍스트 프로그래밍 소개

### Editor Configuration for Python

파이썬 프로그래밍에 사용되는 에디터 설정법.

### Regular Expression HOWTO

정규표현식 하우투 문서.

### Socket Programming HOWTO

소켓 프로그래밍 하우투 문서.

## Sorting Mini-HOWTO

내장 함수인 sort()메소드를 이용한 많은 소팅 기법소개.

## XML HOWTO

파이썬으로 하는 XML 처리

## 유용한 온라인 문서들

온라인 상에 있는 파이썬에 관련한 유용한 문서들을 소개한다.

(아래 문서의 URL들은 2000년 초반에 작성된 기사들이라 변경 또는 삭제되었을 수 있습니다.)

### Why Python? Eric. S. Raymond

<http://www2.linuxjournal.com/lj-issues/issue73/3882.html>

에릭 레이먼드의 글로 자신이 파이썬을 사용하게 된 계기와 파이썬을 사용하는 이유, 그리고 파이썬을 사용함으로써 얻을 수 있었던 것들에 대한 자신의 경험을 위주로 한 글이다.

### Learning to Program - by Alan Gauld

<http://www.crosswinds.net/~agauld/>

컴퓨터 프로그래밍에 대한 기본 원리를 설명하고 프로그래밍이 무엇인지에 대한 내용과 문제 해결을 위한 기초 테크닉들을 설명한다. 대부분 모든 설명이 파이썬을 기반으로 이루어져 있다.

**Non-Programmers Tutorial - Josh Cogliati**

<http://www.honors.montana.edu/~jjc/easytut/easytut/>

프로그래밍에 대한 경험이 없는 사람을 대상으로 하는 파이썬 프로그래밍에 대한 문서이다.

**Instant Python - by Magnus Lie Hetland**

<http://www.hetland.org/python/instant-python.php>

파이썬을 빠르게 배울수 있는 간단한 튜토리얼 문서이다.

**Python for experienced programmers - Mark Pilgrim**

<http://diveintopython.org/toc.html>

객체지향 프로그래밍과 파이썬 프로그래밍에 대한 경험이 있는 사람들을 대상으로 한 보다 깊이 있는 파이썬 프로그래밍 설명 문서이다.

**How to think like a computer scientist Python Version by Allen B.Downey and Jeffrey Elkner**

<http://www.ibiblio.org/obp/thinkCSp/>

제목그대로 프로그래밍을 할 때 컴퓨터 과학자처럼 생각하는 방법에 대한 설명 문서이다. 객체지향 프로그래밍에 대한 설명도 포함한다.

**comp.lang.python**

이곳에는 파이썬에 대한 질문과 답변들, 새로운 소식들이 하루에도 100건 이상 씩 올라온다. 이 뉴스그룹을 잘만 활용한다면 많은 도움이 될 것이다. 이 뉴

스그룹에 글을 올리는 사람들 중에는 잘 알려진 유명한 사람들이 많다. 그러한 사람들과 서로 뉴스그룹 상에서 대화할 수 있는 기회를 가져보도록 하자. comp.lang.python 을 활용하는 방법으로 데자 뉴스를 이용하는 것도 좋은 방법이다. 다음의 URL에 접속하면 comp.lang.python의 기사를 더욱 쉽게 읽을 수 있다. 데자 뉴스는 질문과 답변이 잘 정리되어 있다.

[http://groups.google.com/groups?oi=djq&as\\_ugroup=comp.lang.python](http://groups.google.com/groups?oi=djq&as_ugroup=comp.lang.python)

### **www.python.org**

파이썬 공식 홈페이지로 파이썬에 대한 대부분의 모든 정보를 이곳에서 구할 수 있다. 파이썬 공식 패키지를 다운받을 수 있고, 파이썬에 관한 무수히 많은 문서들을 볼 수가 있다. Topic Guide에서는 Tkinter, XML, Databases, Web Programming 등 깊직한 주제를 다루고 SIGs(Special Interest Groups)에서는 자신이 관심있는 프로그래밍 부분에 대한 자료를 충분히 구할 수 있고 또한 참여 할 수 있으며, 메일링 리스트를 이용하여 정보를 교환할 수 있을 것이다.

### **파이썬 마을(python.kr)**

한국 파이썬 사용자들이 대부분 있는 곳이다. 파이썬을 공부해가며 모르는 부분에 대해서 질문을 할 수 있는 게시판등이 있고 각종 번역물이나 파이썬 관련 정보를 손쉽게 구할 수 있다. 수많은 글들이 올라오고 많은 문서들이 생겨나는 매우 바쁜 곳이다. 아마도 필연적으로 독자는 이 사이트에서 많은 시간을 보내게 될 것이다.