

# 파이썬 프로그래밍

## 포트폴리오

컴퓨터정보공학과

20122460 박성준

# 목차

Chapter 1. 파이썬 언어의 개요와 첫 프로그래밍 -----	3
Chapter 2. 파이썬 프로그래밍을 위한 기초 다지기 -----	9
Chapter 3. 일상에서 활용되는 문자열과 논리 연산 -----	15
Chapter 4. 일상생활과 비유되는 조건과 반복 -----	21
Chapter 5. 항목의 나열인 리스트와 튜플 -----	24
Chapter 6. 일상에서 활용되는 문자열과 논리 연산 -----	28
Chapter 7. 특정 기능을 수행하는 사용자 정의 함수와 내장 함수 -----	34

# 참고 자료

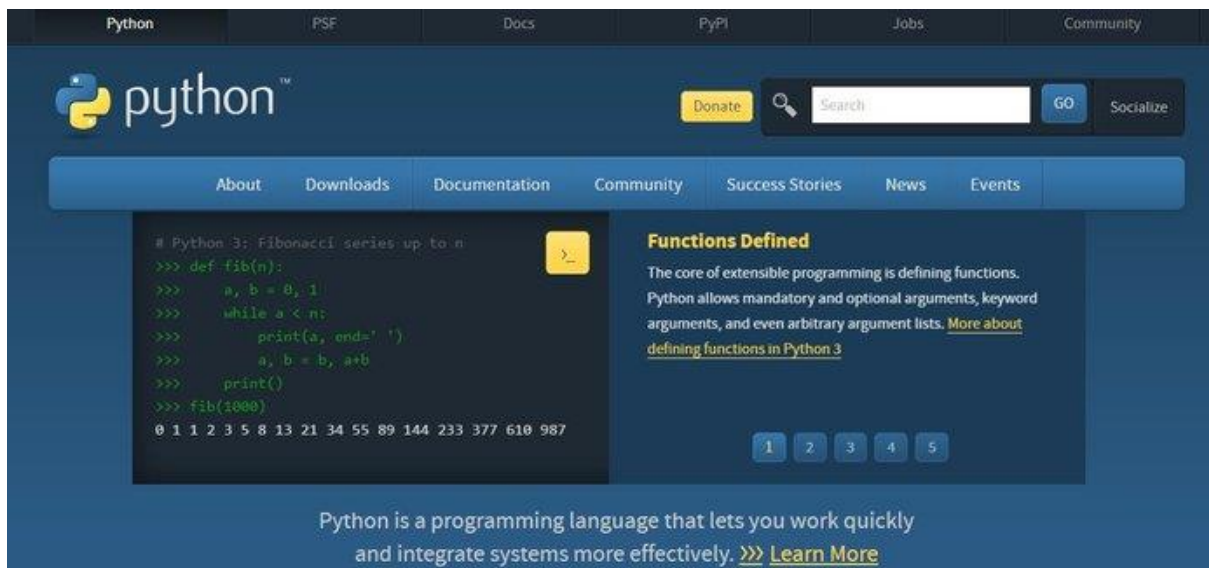
- 파이썬으로 배우는 누구나 코딩 (강환수, 신용현 지음)
- 점프 투 파이썬 (박응용 지음)

# Chapter 01. 파이썬 언어의 개요와 첫 프로그래밍

## 1.파이썬 언어와 컴퓨팅 사고력

### 1.1 파이썬 언어란?

1) 파이썬(python)은 배우기 쉽고 누구나 무료로 사용할 수 있는 오픈 소스(open source) 프로그래밍 언어(programming language) 다. 파이썬은 1991년 네덜란드의 귀도 반 로섬이 개발했으며, 현재는 비영리 단체인 파이썬 소프트웨어 재단이 관리하고 있다.



2) 파이썬은 현재 미국과 우리나라의 대학 등 전 세계적으로 가장 많이 가르치는 프로그래밍 언어 중 하나다. 특히 비전공자의 컴퓨팅 사고력을 키우기 위한 프로그래밍 언어로도 많이 활용되고 있다. 파이썬은 배우기 쉽고 간결하며, 개발 속도가 빠르고 강력하기 때문이다.

또한 파이썬은 라이브러리(library)가 풍부하고 다양한 개발 환경을 제공하고 있어 개발자가 쉽고 빠르게 소프트웨어를 개발하는 데 도움을 준다. 파이썬은 프로그래밍 교육 분야에서뿐 아니라 실무에서도 사용이 급증하고 있으며, 이를 증명이라도 하듯 프로그래밍에 대해 질문하고 답하는 사이트로 유명한 스택오버플로에서 '가장 빠르게 성장하는 프로그래밍 언어'로 선택되기도 했다.

### 1.2 컴퓨팅 사고력과 파이썬

- 지금은 지능, 정보화 혁명 시대인 제4차 산업혁명 시대

우리는 정보와 통신 기술의 발달로 디지털 및 정보화 혁명이 일어난 제3차 산업혁명 시대를 거쳐 스마트폰의 개발과 인공 지능(AI), 사물 인터넷(IoT), 빅데이터 등의 발전으로 제4차 산업혁명 시대에 살고 있다. 즉, 제4차 산업혁명이란 '모든 사물이 연결된 초연결 사회에서 생산되는 빅데이터

를 기존 산업과 융합해 인공지능, 클라우드 등의 첨단 기술로 처리하는 정보-지능화 혁명 시대'라고 할 수 있다.

- 제4차 산업혁명 시대 인재의 핵심역량은 문제 해결 능력과 창의-융합 사고 능력

제4차 산업혁명을 이끌 인재가 갖춰야 할 덕목으로는 문제 해결 능력, 창의-융합 사고 능력, 의사소통 능력과 협업 능력, 자기 주도 학습 능력을 들 수 있다.

- 문제 해결 능력과 창의-융합 사고 능력에 필요한 컴퓨팅 사고력

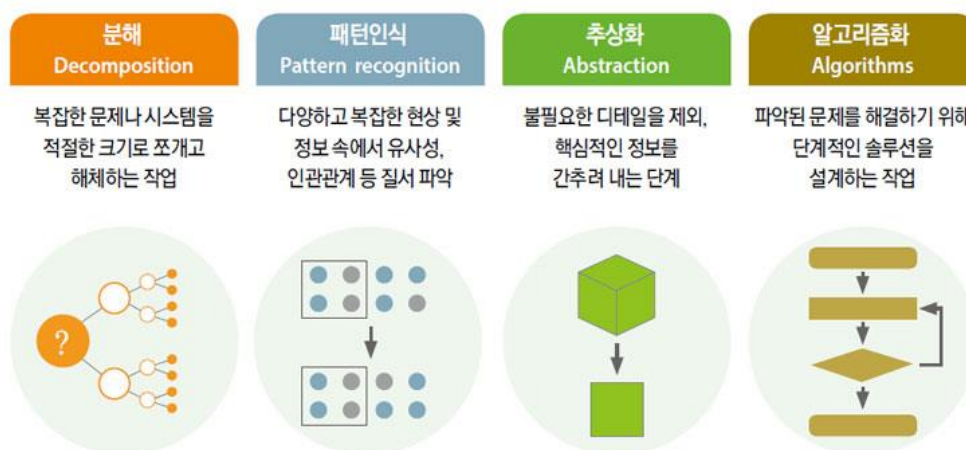
### 컴퓨팅 사고력

컴퓨터 과학의 기본 개념과 원리 및 컴퓨팅 시스템을 활용하여  
실생활 및 다양한 학문 분야의 문제를 이해하고  
창의적 해법을 구현하여 적용할 수 있는 능력



- 컴퓨팅 사고력 구성 요소 : 분해, 패턴 인식, 추상화, 알고리즘

표2 컴퓨팅적 사고 능력의 하위 요소



- 프로그래밍의 절차: 이해, 설계, 구현, 공유

## 2. 파이썬 설치와 파이썬 쉘 실행

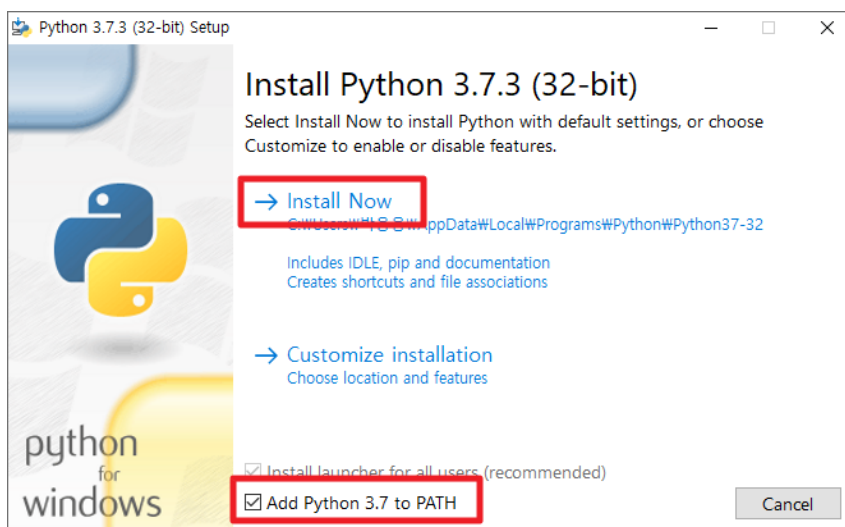
### 2.1 파이썬 개발 도구 설치와 파이썬 쉘의 실행

- 자신의 운영체제에 적합한 파이썬 설치

윈도 사용자라면 파이썬 홈페이지에서 download를 누르면 나타나는 메뉴에서 Python 3.~을 클릭해 설치한다. 파이썬 다운로드 페이지([www.pythong.org/downloads](http://www.pythong.org/downloads))로 이동한 후 Download Python 3.~을 눌러 설치할 수도 있다.



파이썬 설치 첫 화면에서 Install Now를 선택하면 간단하게 설치 할 수 있다.



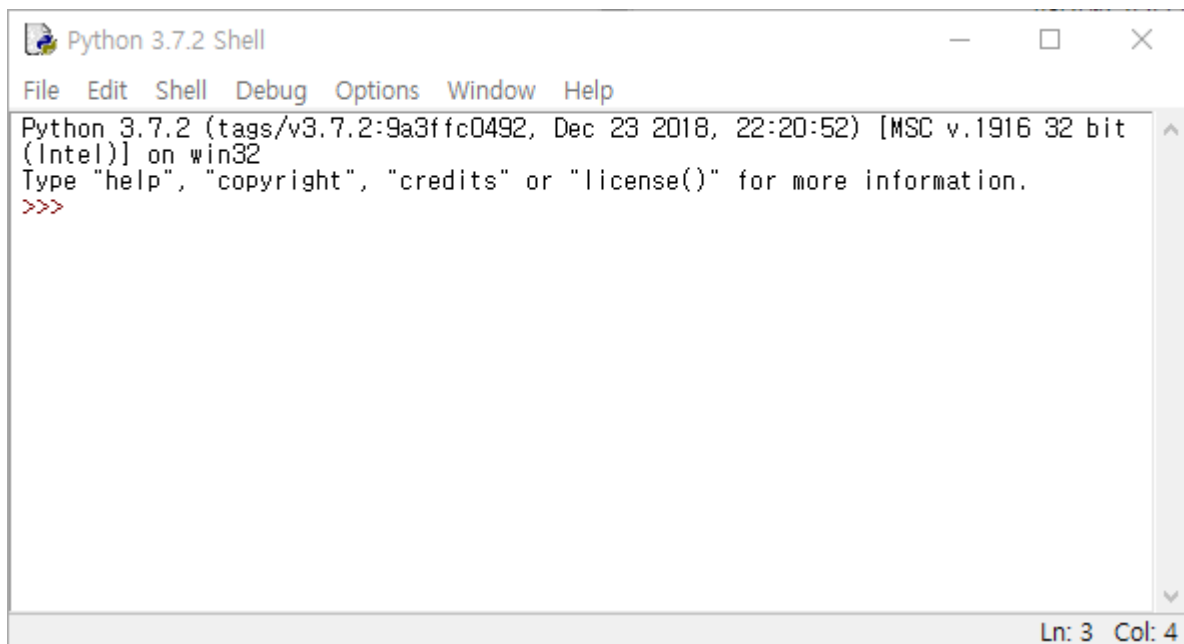
Install Now를 이용해 설치([docs.python.org/3/using/windows.html](https://docs.python.org/3/using/windows.html))할 때는 다음 사항에 유의하자

- 표준 라이브러리와 파이썬 쉘인 IDLE 및 pip, 매뉴얼 문서(documentation) 등이 설치
- 체크박스 Add Python 3.7 to PATH 옵션을 선택하면 파이썬 설치 폴더가 PATH에 설정

#### - 설치한 파이썬 쉘의 실행과 종료

설치된 파이썬 메뉴를 찾아 실행해 보자. 다음 두 가지 방식으로 시스템의 [설치 프로그램 메뉴]를 표시 한 후, 설치된 [Python 3.7] – [IDLE(Python 3.7 32-bit)]를 눌러 '파이썬 쉘 IDLE'을 실행한다.

- 윈도우 화면의 하단에 위치한 [작업 표시줄]의 가장 왼쪽에 있는 [윈도 시작] 버튼 선택
- 키보드의 Windows 로고 키 선택



첫 칸에 프롬프트라 부르는 3개의 부호 기호인 >>>가 표시되며, 키보드 입력을 기다리는 커서가 깜박인다. 프롬프트는 개발자로부터 셸의 명령이나 파이썬 문장을 기다린다는 의미를 가진 기호다. 『파이썬 셸 IDLE』과 같은 모습의 창을 '대화형 창' 또는 'REPL(read-eval-print-loop)창' 이라고도 한다.

## 2.2 파이썬 셸에서 첫 대화형 프로그래밍

- 글자 Hello World!를 출력하는 첫 대화형 코딩

파이썬 셸에 출력하려면 print 다음에 '(Hello World!)'를 입력한다.

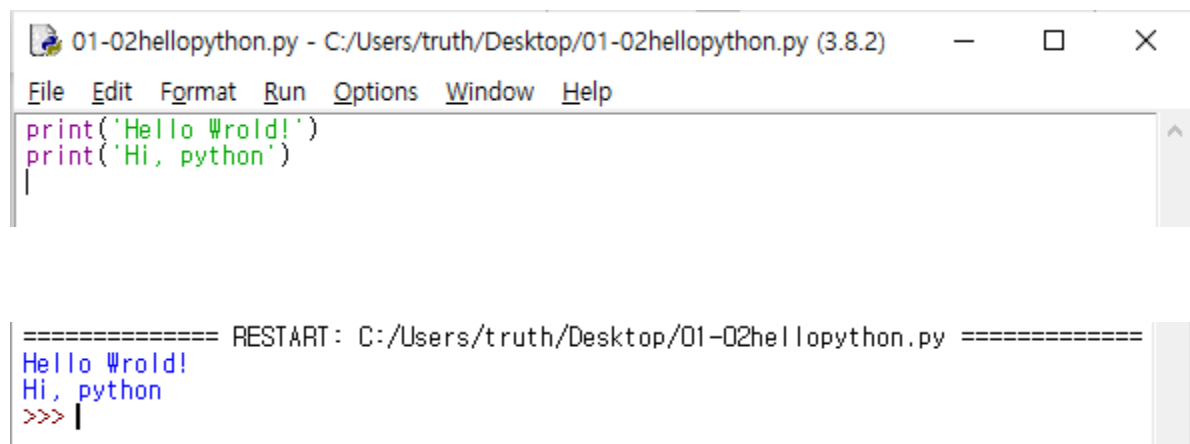
- 'Hello World!'와 같이 출력하고 자하는 글자의 앞뒤에 작은따옴표(또는 홑따옴표)를 붙인다.
- 작은따옴표를 붙인 글자를 문자열 또는 스트링(string)이라 한다.
- Print('Hello World!')를 입력한 후 Enter를 눌러 결과를 알아보자.

```
>>> print('Hello World!')
Hello World!
```

- 파이썬 셸의 명령어는 첫 칸부터 입력

모든 명령어는 첫 칸부터 입력해야 한다. 첫 칸뿐 아니라 그 이상을 공간(space)으로 두고 입력하면 무조건 'SyntaxError: unexpected indent'가 표시된다.

- 두 줄의 출력이 있는 파이썬 두 번째 프로그램 작성



The screenshot shows a Python IDLE window titled '01-02hellopython.py - C:/Users/truth/Desktop/01-02hellopython.py (3.8.2)'. The window has a menu bar with 'File', 'Edit', 'Format', 'Run', 'Options', 'Window', and 'Help'. The main text area contains the following code:

```
print('Hello Wrold!')
print('Hi, python')
```

Below the code editor, the output of the program is displayed, preceded by a separator line: '==== RESTART: C:/Users/truth/Desktop/01-02hellopython.py ====='. The output is:

```
Hello Wrold!
Hi, python
>>>
```

### 3. 제4차 산업혁명 시대, 모두에게 필요한 파이썬

#### 파이썬으로 할 수 있는 일

파이썬으로 할 수 있는 일은 아주 많다. 대부분의 프로그래밍 언어가 하는 일을 파이썬은 쉽고 깔끔하게 처리한다. 파이썬으로 할 수 있는 일들을 나열하자면 끝도 없겠지만 대표적인 몇 가지 예를 들어 보겠다.

#### 시스템 유틸리티 제작

파이썬은 운영체제(윈도우, 리눅스 등)의 시스템 명령어를 사용할 수 있는 각종 도구를 갖추고 있기 때문에 이를 바탕으로 갖가지 시스템 유틸리티를 만드는 데 유리하다. 실제로 여러분은 시스템에서 사용 중인 서로 다른 유틸리티성 프로그램을 하나로 뭉쳐서 큰 힘을 발휘하게 하는 프로그램들을 무수히 만들어낼 수 있다.

※ 유틸리티란 컴퓨터 사용에 도움을 주는 여러 소프트웨어를 말한다.

#### GUI 프로그래밍

GUI(Graphic User Interface) 프로그래밍이란 쉽게 말해 화면에 또 다른 윈도우 창을 만들고 그 창에 프로그램을 동작시킬 수 있는 메뉴나 버튼, 그림 등을 추가하는 것이다. 파이썬은 GUI 프로그래밍을 위한 도구들이 잘 갖추어져 있어 GUI 프로그램을 만들기 쉽다. 대표적인 예로 파이썬 프로그램과 함께 설치되는 Tkinter(티케이인터)가 있다. Tkinter를 사용하면 단 5줄의 소스 코드만으로 윈도우 창을 띄울 수 있다.

#### C/C++와의 결합

파이썬은 접착(glue) 언어라고도 부르는데, 그 이유는 다른 언어와 잘 어울려 결합해서 사용할 수 있기 때문이다. C나 C++로 만든 프로그램을 파이썬에서 사용할 수 있으며, 파이썬으로 만든 프로그램 역시 C나 C++에서 사용할 수 있다.

#### 웹 프로그래밍

일반적으로 익스플로러나 크롬, 파이어폭스 같은 브라우저로 인터넷을 사용하는데, 누구나 한번쯤 웹 서핑을 하면서 게시판이나 방명록에 글을 남겨 본 적이 있을 것이다. 그러한 게시판이나 방명록을 바로 웹 프로그램이라고 한다. 파이썬은 웹 프로그램을 만들기에 매우 적합한 도구이며, 실제로 파이썬으로 제작한 웹 사이트는 셀 수 없을 정도로 많다.



## 수치 연산 프로그래밍

사실 파이썬은 수치 연산 프로그래밍에 적합한 언어는 아니다. 수치가 복잡하고 연산이 많다면 C 같은 언어로 하는 것이 더 빠르기 때문이다. 하지만 파이썬은 NumPy 라는 수치 연산 모듈을 제공한다. 이 모듈은 C 로 작성했기 때문에 파이썬에서도 수치 연산을 빠르게 할 수 있다.

## 데이터베이스 프로그래밍

파이썬은 사이베이스(Sybase), 인포믹스(Infomix), 오라클(Oracle), 마이에스큐엘(MySQL), 포스트그레스큐엘(PostgreSQL) 등의 데이터베이스에 접근하기 위한 도구를 제공한다.

또한 이런 굵직한 데이터베이스를 직접 사용하는 것 외에도 파이썬에는 재미있는 도구가 하나 더 있다. 바로 피클(pickle)이라는 모듈이다. 피클은 파이썬에서 사용하는 자료를 변형 없이 그대로 파일에 저장하고 불러오는 일을 맡아 한다. 이 책에서는 외장 함수에서 피클을 어떻게 사용하고 활용하는지에 대해서 알아본다.

## 데이터 분석, 사물 인터넷

파이썬으로 만든 판다스(Pandas) 모듈을 사용하면 데이터 분석을 더 쉽고 효과적으로 할 수 있다. 데이터 분석을 할 때 아직까지는 데이터 분석에 특화된 'R'이라는 언어를 많이 사용하고 있지만, 판다스가 등장한 이후로 파이썬을 사용하는 경우가 점점 증가하고 있다.

사물 인터넷 분야에서도 파이썬은 활용도가 높다. 한 예로 라즈베리파이(Raspberry Pi)는 리눅스 기반의 아주 작은 컴퓨터이다. 라즈베리파이를 사용하면 홈시어터나 아주 작은 게임기 등 여러 가지 재미있는 것들을 만들 수 있는데, 파이썬은 이 라즈베리파이를 제어하는 도구로 사용된다. 예를 들어 라즈베리파이에 연결된 모터를 작동시키거나 LED 에 불이 들어오게 하는 일을 파이썬으로 할 수 있다.

# Chapter 02. 파이썬 프로그래밍을 위한 기초 다지기

## 1. 다양한 자료: 문자열과 수

### 1.1 자료의 종류와 문자열 표현

- 글자의 모임인 문자열

파이썬에서는 문자 하나 또는 문자가 모인 단어나 문장 또는 단락 등을 문자열(string)이라 한다. 즉, 문자열은 '일련(sequence)의 문자(character) 모임'이라 할 수 있다. 파이썬에서 작은따옴표나 큰따옴표로 앞뒤를 둘러싸 '문자열' 또는 "문자열"처럼 표현한다.

- 파이썬은 문자 하나도 문자열로 취급하며, 따옴표로 둘러싼 숫자(예: '34', '3.14')도 문자열로 취급 한다.
- 즉, 따옴표로 둘러싸면 모두 문자열이다.

다음 출력 함수 print()를 사용하면 대화형 모드와 파일 실행에서 모두 출력된다.

- 함수 print(출력될 자료)는 문자열이나 숫자 등의 자료를 콘솔에 출력하는 일을 수행한다.
- 함수 print()는 출력 이후에 다음 줄로 이동해 출력을 준비한다.

- 문자열의 따옴표는 앞뒤를 동일하게 사용

```
>>> print("Hello string!")
```

SyntaxError: EOL while scanning string literal

## 1.2 문자열 연산자 +, \*와 주석

- 문자열의 연결 연산자 +와 반복 연산자 \*

- 더하기 문자인 +는 문자열에서 문자열을 연결하는 역할을 한다.

```
>>> print("문자열 " '연결')
```

문자열 연결

```
>>> print("python " + "program")
```

Python program

별표(asterisk) \*는 문자열에서 문자열을 지정된 수만큼 반복하는 연산을 수행한다.

```
>>>print('방가 ' * 3)
```

방가 방가 방가

```
>>>print(4 * '쿨록 ')
```

쿨록 쿨록 쿨록 쿨록

-여러 줄의 문자열의 처리에 사용되는 삼중 따옴표

```
>>>print("""비록 그대가 우둔해 그 방법이 처음에는 명확해 보이지 않더라도
```

```
지금 하는게 아예 안 하는 것보다 낫다.""")
```

비록 그대가 우둔해 그 방법이 처음에는 명확해 보이지 않더라도

지금 하는 게 아예 안 하는 것보다 낫다.

-문법과 상관 없는 주석

파이썬 주석은 #으로 시작하고 그 줄의 끝까지 유효하다. 주석 #은 한 줄만 가능하므로 여러 줄에 걸친 주석이 필요하면 줄마다 맨 앞에 #을 넣거나 문자열의 삼중따옴표를 사용한다.

# print()는 콘솔에 자료를 출력하는 함수

```
>>>print('# 이후는 주석') # 한 줄에서 문장 이후에도 주석 사용 가능
```

```
# 이후는 주석
```

```
>>> '''주석으로 사용 가능'''
```

```
'주석으로 사용 가능'
```

### 1.3 정수와 실수의 다양한 연산

- 수의 더하기, 빼기, 곱하기, 나누기 연산자 +, -, \*, /

이러한 연산자를 모두 산술 연산자라 부른다.

-수의 몫, 나머지, 지수승 연산자 //, %, \*\*

정수 나눗셈 연산자 //는 나눈 몫이 결과다. 그러므로 나누기 / 연산에서 소수부 없이 정수만 남는다.

```
>>> 8 / 5
```

```
1.6
```

```
>>> 8 // 5
```

```
1
```

나머지 연산자 %는 나눈 나머지가 결과다.

```
>>> 5 % 3
```

```
2
```

연속한 별표 2개인 연산자 \*\*는 지수승 연산자다.

```
>>> 5 ** 3
```

```
125
```

-최근 결과의 특별한 저장 장소, 언더스코어(\_)

대화형 모드에서 마지막에 실행된 결과값은 특별한 저장 공간인 \_에 대입된다.

```
>>> 60
```

```
60
```

```
>>> _
```

```
60
```

```
>>> 3 * _
```

```
180
```

```
>>> _
```

```
180
```

숫자, 문자열 등 유효한 연산식이면 모두 가능하다.

```
>>> '파'
```

```
'파'
```

```
>>> _
```

```
'파'
```

```
>>> '파이' + '썬'*3
```

```
'파이 썬썬썬'
```

```
>>> _
```

```
'파이 썬썬썬'
```

-표현식 문자열 실행 함수 eval()

함수 eval('expression')은 실행 가능한 연산식 문자열인 expression을 실행한 결과를 반환한다.

```
>>> eval('3 + 15 / 2')
```

```
10.5
```

## 2. 변수와 키워드, 대입 연산자

### 2.1 자료형

-자료형과 type() 함수

- 파이썬에서 자료형을 살펴보면 정수는 int, 실수는 float 그리고 문자열은 str로 사용  
대화형 모드에서 자료형을 직접 알아보려면 type() 함수를 사용해야 한다. 다음 결과에서 알 수 있듯이 결과가 모두 <class 'int'>와 같이 클래스(class)가 앞에 표시 된다.

```
>>> type(3)
```

```
<class 'int'>
```

```
>>> pi = 3.14
```

```
>>> type(pi)
```

```
<class 'float'>
```

```
>>> type('python')
```

```
<class 'str'>
>>> type( 3 + 4j ) # 복소수는 class complex 클래스다.
<class 'complex'>
```

## 2.2 변수와 대입연산자

- 변수의 이해와 대입 연산자 =을 이용한 값의 저장

값을 변수에 저장하기 위해서는 대입 연산자(=)가 필요하다. =의 의미는 왼쪽 화살표라고 생각하자. 즉, 오른쪽 값을 왼쪽 변수에 저장하라는 의미다.

```
>>> unit = 3
```

```
>>> 3 = unit
SyntaxError: can't assign to literal
```

```
>>> 5 = 3 + 2
SyntaxError: can't assign to literal
```

```
>>> 5 == 3 + 2
True
```

## 3. 자료의 표준 입력과 자료 변환 함수

### 3.1 표준 입력과 다양한 변환 함수

프로그램 과정에서 셸이나 콘솔에서 사용자의 입력을 받아 처리하는 방식을 표준 입력이라 한다.

- 함수 input()으로 문자열 표준 입력

```
>>> input()
Java
'java'
>>> pl = input()
python
>>> print(pl)
Python
```

- 문자열과 정수, 실수 간의 자료 변환 함수 str(), int(), float()

함수 str()은 주로 정수와 실수를 문자열로 변환하는 데 사용한다.

```
>>> str(235)
'235'
>>> str(2.71828)
'2.71828'
```

함수 int()는 정수 형태의 문자열인 '6400' 등을 정수, float()는 소수점이 있는 실수 형태의 문자열인 '2.71828' 등을 실수로 변환한다.

```
>>> int('6400')
6400
>>> float('3.141592')
3.141592
```

또한 함수 int()는 실수를 정수, float()는 정수를 실수로 변환한다.

```
>>> int(2.71828)
2
>>> float(3)
3.0
```

- 숫자 형태의 문자열을 정수나 실수로 변환

```
>>> rate = input('예금 만기 이율(%)은? ')
예금 만기 이율(%)은? 4.2
>>> rate / 100 # rate가 문자열이므로 오류 발생
>>> float(rate) / 100
0.042
```

### 3.2 16진수, 10진수, 8진수, 2진수와 활용

- 16진수, 8진수, 2진수 상수 표현

16진수는 맨 앞에 0x, 8진수와 2진수는 각각 0o, 0b로 시작한다. 각각의 진수를 표시하는 알파벳은 대소문자가 가능하다.

- 10진수의 변환 함수 bin(), oct(), hex()

10진수를 바로 16진수, 8진수, 2진수로 표현되는 문자열로 변환하려면 bin(), oct(), hex()를 사용해야 한다.

- int(정수, 진법기수)를 활용한 여러 진수 상수 형태 문자열의 10진수 변환

여러 진수 형태 문자열을 10진수로 바꾸려면 함수 int('strnum', n)을 사용해야 한다. 여기서 n은 변환하려는 문자열 진수의 숫자를 2, 8, 10, 16 등으로 넣는다.

```
>>> int('17'), int('25', 10) # 모두 10진수로 변환
(17, 25)
>>> int('11', 2) # 2진수 문자열을 10진수로 변환
3
>>> int('10', 8) # 8진수 문자열을 10진수로 변환
8
```

```
>>> int('1A', 16) # 16진수 문자열을 10진수로 변환
26
```

## Chapter 3. 일상에서 활용되는 문자열과 논리 연산

### 1. 문자열 다루기

#### 1.1 문자열 str 클래스와 부분 문자열 참조 슬라이싱

- 문자열은 클래스 str의 객체
- 함수 len()으로 문자열의 길이 참조

함수 len(문자열)으로 문자열의 길이를 알 수 있다.

```
>>> a = 'python'
>>> len(a)
6
>>> len('파이썬')
3
```

- 문자열의 문자 참조

문자열을 구성하는 문자는 0부터 시작되는 첨자(index)를 대괄호 안에 기술해 참조(indexing)가 가능하다. -1부터 시작돼 -2, -3으로 작아지는 첨자도 역으로 참조한다. 첨자가 유효 범위를 벗어나면 IndexError가 발생한다.

```
Index from rear:  -6  -5  -4  -3  -2  -1
Index from front:   0   1   2   3   4   5
                  +---+---+---+---+---+---+
                  | a | b | c | d | e | f |
                  +---+---+---+---+---+---+
Slice from front:  :   1   2   3   4   5   :
Slice from rear:   :  -5  -4  -3  -2  -1   :
```

#### 1.2 문자열의 부분 문자열 참조 방식

- 0과 양수를 이용한 문자열 슬라이싱

문자열에서 일부분을 참조하는 방법을 슬라이싱(slicing)이라고 한다.

콜론을 사용한 [start:end]로 부분 문자열을 반환하는데, start 첨자에서 end-1 첨자까지의 문자열을 반환한다.

```
>>> 'python' [1:5]
'ytho'
```

- 음수를 이용한 문자열 슬라이싱

양수와 마찬가지로이다.

```
>>> 'python' [-5:-1]
'ytho'
```

순 방향의(), 양수, 역방향의 음수를 혼합해 슬라이싱에 사용할 수 있다.

```
>>> 'python' [1:-1]
'ytho'
>>> 'python' [0:-1]
'pytho'
```

문자열 슬라이싱에서 start와 end로 구성하는 문자열이 없다면 아무것도 없는 빈 문자열 "" 을 반환한다.

```
>>> 'python' [5:-1]
''
>>> 'python' [3:1]
''
>>> 'python' [-3:-4]
''
```

- start와 end를 비우면 '처음부터'와 '끝까지'를 의미

```
>>> 'python' [:3]
'pyt'
>>> 'python' [:4]
'pyth'
>>> 'python' [1:]
'ython'
```

그러므로 앞뒤를 모두 비우면 문자열 전체를 반환한다.

- str[start:end:step]으로 문자 사이의 간격을 step으로 조정 가능

부분 반환 문자열에서 문자 사이의 간격을 step으로 조정하려면 str[start:end:step]을 사용해야 한다. step을 생략하면 1이다.

```
>>> 'python'[: : 1]
'python'
>>> 'python'[1: 5: 2]
'yh'
>>> 'python'[1: 5: 3]
'yo'
```



간격인 step은 음수도 가능하며, 이 경우 start는 end보다 작아야 한다.

str[::-1]은 문자열 str을 역순으로 구성된 문자열로 반환한다.

```
>>> 'python'[::-1]
'nohtyp'
>>> 'python'[5:0:-1]
'nohty'
>>> 'python'[-1:-7:-1]
'nohtyp'
```

### 1.3 문자 함수와 이스케이프 시퀀스

- 문자 함수 ord()와 chr()

파이썬은 유니코드 문자를 사용, 한글 문자 '가'의 유니코드 번호는 내장 함수 ord('가')로 알 수 있다. 이와 반대로 유니코드 번호 44032로 내장 함수 chr(44032)를 호출하면 문자를 반환한다.

```
>>> ord('가')
44032
>>> chr(44032)
'가'
```

- 이스케이프 시퀀스 문자

#### [이스케이프 코드란?]

문자열 예제에서 여러 줄의 문장을 처리할 때 백슬래시 문자와 소문자 n을 조합한 \n 이스케이프 코드를 사용했다. 이스케이프 코드란 프로그래밍할 때 사용할 수 있도록 미리 정의해 둔 "문자 조합"이다. 주로 출력물을 보기 좋게 정렬하는 용도로 사용한다. 몇 가지 이스케이프 코드를 정리하면 다음과 같다.

코드	설명
\n	문자열 안에서 줄을 바꿀 때 사용
\t	문자열 사이에 탭 간격을 줄 때 사용
\\	문자 \를 그대로 표현할 때 사용
\'	작은따옴표(')를 그대로 표현할 때 사용
\"	큰따옴표(")를 그대로 표현할 때 사용
\r	캐리지 리턴(줄 바꿈 문자, 현재 커서를 가장 앞으로 이동)
\f	폼 피드(줄 바꿈 문자, 현재 커서를 다음 줄로 이동)
\a	벨 소리(출력할 때 PC 스피커에서 '뻑' 소리가 난다)
\b	백 스페이스
\000	널 문자

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

- 문자열의 최대와 최소

내장 함수 min()과 max()는 인자의 최댓값과 최솟값을 반환하는 함수다.

```
>>> min('3259')
'2'
>>> max('3259')
'9'
>>> max(3, 96.4, 13)
96.4
>>> min('ipython', 'java')
'ipython'
>>> max('ipython', 'java')
'java'
```

## 2. 문자열 관련 메소드

### 2.1 문자열 대체와 부분 문자열 출현 횟수, 문자열 삽입

- 문자열 바꿔 반환하는 메소드 replace()

메소드 str.replace(a, b)는 문자열 str에서 a가 나타나는 모든 부분을 b로 모두 바꾼 문자열을 반환한다.

```
>>> str = '자바는 인기 있는 언어 중 하나다.'
>>> str.replace('자바는', '파이썬은')
'파이썬은 인기 있는 언어 중 하나다.'
>>> str.replace(' ', '')
'자바는인기있는언어중하나다.'
```

메소드 replace(old, new, count)는 문자열 old를 new로 대체하는데, 옵션인 count는 대체 횟수를 지정한다.

```
>>> str = '파이썬 파이썬 파이썬'
>>> str.replace('파이썬', 'Python!')
'Python! Python! Python!'
>>> str.replace('파이썬', 'Python!', 1)
'Python! 파이썬 파이썬'
>>> str.replace('파이썬', 'Python!', 2)
'Python! Python! 파이썬'
```

- 부분 문자열 출현 횟수를 반환하는 메소드 count()

```
str = '단순한 것이 복잡한 것보다 낫다.'
str.count('복잡') 1
str.count('것') 2
```

- 문자와 문자 사이에 문자열을 삽입하는 메소드 join()

```
>>> num = '12345'
```

```
>>> '->'.join(num)
```

```
'1->2->3->4->5'
```

## 2.2 문자열 찾기

- 문자열을 찾는 메소드 find()와 index()

메소드 index()는 찾는 문자열이 없을 경우, ValueError를 발생시키지만 find()는 오류를 발생시키지 않고 -1을 반환한다.

```
>>> str = '자바 C 파이썬 코틀린'
```

```
>>> str.find('파이')
```

```
5
```

```
>>> str.index('파이썬')
```

```
5
```

## 2.3 문자열 나누기

- 문자열을 여러 문자열로 나누는 split() 메소드

문자열 메소드 str.split()는 문자열 str에서 공백을 기준으로 문자열을 나눠 준다.

```
>>> '사과 배 복숭아 딸기 포도'.split()
```

```
['사과', '배', '복숭아', '딸기', '포도']
```

```
>>> '데스크톱 1000000, 노트북 1800000, 스마트폰 1200000'.split()
```

```
['데스크톱', '1000000', '노트북', '1800000', '스마트폰', '1200000']
```

만일 str.split(',')처럼 괄호 안에 특정한 문자열 값이 있을 경우에는 이 부분 문자열 값을 구분자를 이용해 문자열을 나눠 준다.

```
>>> '데스크톱 1000000, 노트북 1800000, 스마트폰 1200000'.split(',')
```

```
['데스크톱 1000000', '노트북 1800000', '스마트폰 1200000']
```

## 2.4 출력을 정형화하는 함수 format()

```
>>> str = '{} + {} = {}'.format(3, 4, 3+4)
```

```
>>> print(str)
```

```
3 + 4 =7
```

인자는 상수뿐 아니라 변수도 가능하다. {0}, {1} 처럼 인자의 순서를 나타내는 정수를 0부터 삽입하면 인자의 순서와 출력 순서를 조정할 수 있다.

```
>>> a, b = 10, 4
```

```
>>> print('{} * {} = {}'.format(a, b, a*b))
```

```
10 * 4 = 40
```

```
>>> print('{0} * {1} = {2}'.format(a, b, a/b))
10 / 4 = 2.5
>>> print('{1} * {0} = {2}'.format(a, b, b/a))
4 / 10 = 0.4
```

- 문자열의 중괄호 {n:md}로 정수를 형식 유형으로 출력 처리

10진수(decimal) 정수는 d, 부동소수(float)는 f로 표기한다. 실수인 경우, 꼭 10.3은 소수점이 포함된 전체 꼭 10, 소수점 이하 꼭 3을 의미한다.

```
>>> a, b = 10, 3
>>> print('{0:5d} / {1:5d} = {2:10.3f}'.format(a, b, a/b))
10 /      3 =      3.333
```

2.6 C언어의 포매팅 스타일인 %d와 %f등으로 출력

코드	설명
%s	문자열(String)
%c	문자 1개(character)
%d	정수(Integer)
%f	부동소수(floating-point)
%o	8진수
%x	16진수
%%	Literal % (문자 % 자체)

### 3. 논리 자료와 다양한 연산

#### 3.1 논리 값과 논리 연산

- 논리 유형 bool과 함수 bool()

정수의 0, 실수의 0.0, 빈 문자열 등은 False이며, 음수와 양수, 뭔가가 있는 'java' 등의 문자열은 모두 True다. 내장 함수 bool(인자)로 인자인 변수나 상수의 논리 값을 알 수 있다.

```
>>> bool(0), bool(0.0), bool('')
(False, False, False)
>>> bool(10), bool(3.14), bool('python')
(True, True, True)
```

논리 값 True와 False는 int(논리값) 함수에 의해 각각 1과 0으로 변환된다.

```
>>> int(True), int(False)
```

```
(1, 0)
```

## Chapter 4. 일상생활과 비유되는 조건과 반복

### 1. 조건에 따른 선택 if ~ else

#### 1.1 조건의 논리 값에 따른 선택 if

- 조건에 따른 선택을 결정하는 if문

- if 문에서 논리 표현식 이후에는 반드시 콜론이 있어야 한다.
- 콜론 이후 다음 줄부터 시작되는 블록은 반드시 들여쓰기(indentation)를 해야 한다. 그렇지 않으면 오류가 발생하니 주의하자.

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

#### 1.2 조건에 따라 하나를 선택하는 if ~ else

- 논리 표현식 결과인 True와 False에 따라 나뉘는 if ~ else문

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

#### 1.4 여러 조건 중에서 하나를 선택하는 구문 if ~ elif

- 다중 택일 결정 구조인 if ~ elif

if와 else만으로는 다양한 조건을 판단하기 어렵다. 다음 예를 보더라도 if와 else만으로는 조건을 판단하는 데 어려움을 겪게 된다.

"주머니에 돈이 있으면 택시를 타고, 주머니에 돈은 없지만 카드가 있으면 택시를 타고, 돈도 없고 카드도 없으면 걸어 가라."

위 문장을 보면 조건을 판단하는 부분이 두 군데가 있다. 먼저 주머니에 돈이 있는지를 판단해야 하고 주머니에 돈이 없으면 다시 카드가 있는지 판단해야 한다.

위 예를 elif 를 사용하면 다음과 같이 바꿀 수 있다.

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

즉 elif는 이전 조건문이 거짓일 때 수행된다. if, elif, else를 모두 사용할 때 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>
    ...
```

## 2. 반복을 제어하는 for문과 while문

### 2.1 시퀀스의 내부 값으로 반복을 실행하는 for문

<반복 조건>에 따른 반복

while <반복 조건>:

반복 몸체인 문장들

<시퀀스> 항목마다 반복

for 변수 in <시퀀스>:

반복 몸체인 문장들

- 정해는 있는 시퀀스의 항목 값으로 반복을 실행하는 for문

for 변수 in 시퀀스 :

문장1

문장2

else :

문장3

else: 이후의 블록인 문장 3은 for 시퀀스의 모든 항목으로, 반복이 종료된 마지막에 실행된다.

```

sum = 0
for i in 1.1, 2.5, 3.6, 4.2, 5.4:
    sum += i
    print(i, sum)
else:
    print('합: %.2f, 평균: %.2f' % (sum, sum / 5))

```

결과 1.1 1.1  
 2.5 3.6  
 3.6 7.2  
 4.2 11.4  
 5.4 16.8  
 합: 16.80, 평균: 3.36

- 내장 함수 range()를 사용한 for문

내장 함수 range(5)는 정수 0에서 4까지 5개의 항목인 정수로 구성되는 시퀀스를 만든다.

Range(1, 10, 2)은 1에서 9까지 간격이 2씩 증가하면서 5개의 정수를 반복할 수 있다.

2.2 횟수를 정하지 않은 반복에 적합한 while 반복

- 반복 구조가 간단한 while 반복

while 논리 표현식:

문장 1

문장 2

else:

문장 3

### 3. 임의의 수인 난수와 반복을 제어하는 break문, continue문

3.1 임의의 수인 난수 발생과 반복에 활용

파이썬에서는 모듈 random의 함수 randint(시작, 끝)를 사용해 정수 시작과 끝 수 사이에서 임의의 정수를 얻을 수 있다.

```
>>> import random
```

```
>>> random.randint(1, 3)
```

```
3
```

```
>>> random.randint(1, 3)
```

```
1
```

3.2 반복을 제어하는 break문과 continue문

- 반복을 종료하는 break문

반복 while의 논리 표현식이 True라면 무한히 반복된다. 이를 무한 반복이라고 한다. for나 while 반복 내부에서 문장 break는 else: 블록을 실행시키지 않고 반복을 무조건 종료한다.

반복 while이나 for문에서 break 문장은 반복을 종료하고 반복 문장 이후를 실행한다. 즉, break문은 특정한 조건에서 즉시 반복을 종료할 경우에 사용된다. break문으로 반복이 종료되는 경우, 반복의 else: 블록은 실행되지 않는다.

```
for 변수 in 시퀀스:           혹은      while 논리 표현식:
    break_앞_문장들
    if 조건:
        break
    break_뒤_문장들
else:
    반복이_정상적으로_종료되면_실행되는_문장들
```

## Chapter 5. 항목의 나열인 리스트와 튜플

### 1. 여러 자료 값을 편리하게 처리하는 리스트

#### 1.1 리스트의 개념과 생성

- 관련된 나열 항목을 관리하는 리스트

리스트는 항목의 나열인 시퀀스다. 각 항목은 모두 같은 자료형일 필요는 없다. 즉, 정수, 실수, 문자열, 리스트 등이 모두 가능하다. 항목 순서는 의미가 있으며, 항목 자료 값은 중복돼도 상관 없다.

■ 리스트는 대괄호 [] 사이에 항목을 기술한다.

#### 5-1. 다양한 커피 종류가 저장된 리스트

```
coffee = ['에스프레소', '아메리카노', '카페라떼', '카페모카']
print(coffee)

num = 0
for s in coffee:
    num += 1
    print('%d. %s' % (num, s))
```

결과

```
['에스프레소', '아메리카노', '카페라떼', '카페모카']
1. 에스프레소
2. 아메리카노
3. 카페라떼
4. 카페모카
>>>
```

- 메소드 append()로 리스트에 요소 추가

append를 사전에서 검색해 보면 "덧붙이다, 첨부하다"라는 뜻이 있다. 이 뜻을 안다면 다음 예가 바로 이해될 것이다. append(x)는 리스트의 맨 마지막에 x를 추가하는 함수이다.



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

### 1.3 리스트의 항목 수정

- 리스트의 메소드 count( )와 index( )

count( )는 리스트 안에 x가 몇 개 있는지 조사하여 그 개수를 돌려주는 함수이다.

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

index( ) 함수는 리스트에 x 값이 있으면 x의 위치 값을 돌려준다.

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

### 1.4 리스트 내부에 다시 리스트를 포함하는 중첩 리스트

- 리스트의 항목으로 리스트 구성

```
>>> animal = [['사자', '코끼리', '호랑이'], '조류', '어류']
>>> print(animal)
[['사자', '코끼리', '호랑이'], '조류', '어류']
>>> print(animal[0])
['사자', '코끼리', '호랑이']
```

리스트 animal에서 '코끼리를 참조하려면 첨자를 두 번 사용해 animal[0][1]로 표시해야 한다.

## 2. 리스트의 부분 참조와 항목의 삽입과 삭제

- 리스트 메소드 insert(첨자, 항목)으로 삽입

```
>>> kpop = []
>>> kpop.insert(0, '블랙핑크')
>>> kpop.insert(0, 'BTS')
>>> kpop
['BTS', '블랙핑크']
```

계속 첨자 1에 '장범준'을 삽입하면 두 번째 항목으로 삽입된다.

```
>>> kpop.insert(1, '장범준')
>>> kpop
['BTS', '장범준', '블랙핑크']
```

- 리스트 메소드 remove(항목)과 pop(첨자), pop( )로 항목 삭제

인자가 없는 pop( )은 마지막 항목을 삭제하고 삭제된 값을 반환하며, 첨자를 사용한 pop(첨자)는 지정된 첨자의 항목을 삭제하고 반환한다. 그러므로 pop( )의 호출로 삭제된 값을 대입하거나 출력할 수 있다.

```
>>> kpop = ['BTS', '장범준', '블랙핑크', '잔나비']
```

```
>>> print(kpop.pop(1))
```

장범준

```
>>> print(kpop.pop( ))
```

잔나비

```
>>> print(kpop)
```

['BTS', '블랙핑크']

- 문장 del에 의한 항목이나 변수의 삭제

del a[x]는 x번째 요솟값을 삭제한다. 여기에서는 a 리스트에서 a[1]을 삭제하는 방법을 보여준다.

del 함수는 파이썬이 자체적으로 가지고 있는 삭제 함수이며 다음과 같이 사용한다.

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

## 2.4 리스트의 추가, 연결과 반복

- 리스트에 리스트를 추가하는 메소드 extend( )

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

```
>>> a = [1,2,3]
>>> a.extend([4,5])
>>> a
[1, 2, 3, 4, 5]
>>> b = [6, 7]
>>> a.extend(b)
>>> a
[1, 2, 3, 4, 5, 6, 7]
```

## 2.5 리스트 항목의 순서와 정렬

- 리스트 항목 순서를 뒤집는 메소드 reverse( )

reverse 메소드는 리스트를 역순으로 뒤집어 준다. 이때 리스트 요소들을 순서대로 정렬한 다음 다시 역순으로 정렬하는 것이 아니라 그저 현재의 리스트를 그대로 거꾸로 뒤집는다.

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

- 리스트 항목 순서를 정렬하는 메소드 `sort()`  
`sort` 메소드는 리스트의 요소를 순서대로 정렬해 준다.

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

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

### 3. 항목의 순서나 내용을 수정할 수 없는 튜플

#### 3.1 괄호로 정의하는 시퀀스 튜플

- 수정할 수 없는 항목의 나열인 튜플

튜플은 리스트와 달리 항목의 순서나 내용의 수정이 불가능하다. 튜플도 모두 콤마로 구분된 항목들의 리스트로 표현되며, 각각의 항목은 정수, 실수, 문자열, 리스트, 튜플 등 제한이 없다.

- 튜플은 괄호(...) 사이에 항목을 기술한다. 괄호는 생략할 수 있다.

```
singer = ('BTS', '볼빨간사춘기', 'BTS', '블랙핑크', '태연')
```

```
credit = ([2020, 1, 18], [2020, 2, 17])
```

```
space = '밤', '낮', '해', '달'
```

- 튜플의 생성

빈 튜플은 `()`로 만든다. 튜플의 자료형 이름은 클래스 `tuple`이다. 빈 튜플은 함수 `tuple()`로도 생성할 수 있다.

다음 변수 `inta`는 `int`형 1이 저장된 변수다. 그러나 1, 로 저장한 변수 `tupa`는 튜플이며, 항목이 1, 하나다. 항목이 하나인 튜플을 표현할 때는 마지막에 콤마를 반드시 붙여야 한다. (1, )과 같이 출력되는 것을 알 수 있다.

```
>>> inta = 1
```

```
>>> tupa = 1,
```

```
>>> print(tupa)
```

(1, )      # 항목이 1 하나인 튜플을 나타낸다.

- 튜플 항목 참조와 출력

튜플도 리스트와 같이 첨자 참조 `tuple[index]`와 `tuple[start:stop:step]` 슬라이스 가능하다.

그러나 튜플은 수정이 불가능하므로 첨자와 슬라이스로 수정할 수 없다.

## Chapter 6. 일상에서 활용되는 문자열과 논리 연산

### 1. 키와 값인 쌍의 나열인 딕셔너리

#### 1.1 딕셔너리의 개념과 생성

딕셔너리는 키와 값의 쌍인 항목을 나열한 시퀀스다. 각각의 항목은 키:값과 같이 키와 값을 콜론으로 구분하고 전체는 중괄호 `{...}`로 묶는다.

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

- 빈 딕셔너리의 생성과 항목 추가

```
>>> lect = {}  
>>> print(lect)  
{}
```

인자가 없는 내장 함수 `dict()` 호출로도 빈 딕셔너리를 생성할 수 있다. 항목을 딕셔너리에 넣으려면 '딕셔너리[키] = 값'의 문장을 사용해야 한다. 다음은 키-값의 쌍인 '강좌명': '파이썬 기초'의 항목을 삽입하는 문장이다.

```
>>> lect = dict() #빈 딕셔너리  
>>> lect['강좌명'] = '파이썬 기초';
```

딕셔너리 항목 값으로 리스트나 튜플 등이 가능하다.

#### 1.2 다양한 인자의 함수 `dict()`로 생성하는 딕셔너리

- 리스트 또는 튜플로 구성된 키-값을 인자로 사용

내장 함수 `dict()` 함수에서 인자로 리스트나 튜플 1개를 사용해 딕셔너리를 만들 수 있다.

```
>>> day = dict([ ])
>>> day = dict( )
```

- 키가 문자열이면 키 = 값 항목의 나열로도 딕셔너리 생성

```
>>> day = dict (월='Monday', 화='Tuesday', 수='Wednesday' )
```

// 키가 문자열이면 따옴표 없이 기술할수 있다.

```
>>> print(day)
```

```
{'월' : 'Monday', '화' : 'Tuesday', '수' : 'Wednesday' }
```

1.3 딕셔너리 키는 수정 불가능한 객체로 사용

- 딕셔너리의 키로 정수, 실수 등 사용 가능

딕셔너리의 키는 수정 불가능한 객체는 모두 가능하다. 따라서 정수는 물론 실수도 가능하다.

```
>>> real = {3.14: '원주율'}
```

- 튜플은 키로 가능하지만 리스트는 키로 사용 불가능

1.4 딕셔너리 항목의 순회

- 딕셔너리 메소드 `keys()`

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

`a.keys()`는 딕셔너리 `a`의 Key만을 모아서 `dict_keys` 객체를 돌려준다.

`dict_keys` 객체는 다음과 같이 사용할 수 있다. 리스트를 사용하는 것과 차이가 없지만, 리스트 고유의 `append`, `insert`, `pop`, `remove`, `sort` 함수는 수행할 수 없다.

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

`dict_keys` 객체를 리스트로 변환하려면 다음과 같이 하면 된다.

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

- 딕셔너리 메소드 `items()`

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

`items` 함수는 Key와 Value의 쌍을 튜플로 묶은 값을 `dict_items` 객체로 돌려준다. `dict_values` 객체와 `dict_items` 객체 역시 `dict_keys` 객체와 마찬가지로 리스트를 사용하는 것과 동일하게 사용할 수 있다.

- 딕셔너리 메소드 `values()`

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

Key를 얻는 것과 마찬가지로 방법으로 Value만 얻고 싶다면 values 함수를 사용하면 된다. values 함수를 호출하면 dict\_values 객체를 돌려준다.

### 1.5 딕셔너리 항목의 참조와 삭제

- 키로 조회하는 딕셔너리 메소드 get(키[, 키가\_없을\_때\_반환\_값])

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

get(x) 함수는 x 라는 Key 에 대응되는 Value 를 돌려준다. 앞에서 살펴보았듯이 a.get('name')은 a['name']을 사용했을 때와 동일한 결과값을 돌려받는다.

다만 다음 예제에서 볼 수 있듯이 a['nokey']처럼 존재하지 않는 키(nokey)로 값을 가져오려고 할 경우 a['nokey']는 Key 오류를 발생시키고 a.get('nokey')는 None 을 돌려준다는 차이가 있다. 어떤것을 사용할지는 여러분의 선택이다.

※ 여기에서 None 은 "거짓"이라는 뜻이라고만 알아두자.

```
>>> a = {'name': 'pey', 'phone': '0119993323', 'birth': '1118'}
>>> print(a.get('nokey'))
None
>>> print(a['nokey'])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'nokey'
```

딕셔너리 안에 찾으려는 Key 값이 없을 경우 미리 정해 둔 디폴트 값을 대신 가져오게 하고 싶을 때에는 get(x, '디폴트 값')을 사용하면 편리하다.

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

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

- 키로 삭제하는 딕셔너리 메소드 pop(키[, 키가\_없을\_때\_반환\_값])

딕셔너리 메소드 pop(키)는 키인 항목을 삭제하고, 삭제되는 키의 해당 값을 반환한다.

```
>>> city = {'대한민국': '부산', '뉴질랜드': '웰링톤', '캐나다': '몬트리올'}
>>> print(city.pop('뉴질랜드'))
웰링톤
>>> city
{'대한민국': '부산', '캐나다': '몬트리올'}
```

- 임의의 항목을 삭제하는 딕셔너리 메소드 popitem( )

딕셔너리 메소드인 `popitem()`은 임의의 ( 키, 값 )의 튜플을 반환하고 삭제한다. 만일 데이터가 하나도 없다면 오류가 발생한다.

```
>>> city = {'대한민국': '부산', '뉴질랜드': '웰링턴'}
>>> print(city.popitem())
('뉴질랜드': '웰링턴')
>> city
{'대한민국': '부산'}
>>> print(city.popitem())
('대한민국': '부산')
>>> city
{}
```

- 문장 `del`로 딕셔너리 항목 삭제

딕셔너리를 문장 `del`에 이어 키로 지정하면 해당 항목이 삭제된다.

```
>>> city = {'대한민국': '부산', '뉴질랜드': '웰링턴'}
>>> del city['뉴질랜드']
>>> city
{'대한민국': '부산'}
```

## 1.6 딕셔너리 결합과 키의 멤버십 검사 연산자 `in`

- 딕셔너리를 결합하는 메소드 `update()`

```
>>> kostock = {'Samsung Elec.' : 40000, 'Daum KAKAO' : 80000}
>>> usstock = {'MS' : 150, 'Apple' : 180}
>>> kostock.update(usstock)
>>> kostock
{'Samsung Elec.' : 40000, 'Daum KAKAO' : 80000, 'MS' : 150, 'Apple' : 180}
```

- 문장 `in`으로 쉽게 검사

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

'name' 문자열은 a 딕셔너리의 Key 중 하나이다. 따라서 'name' in a를 호출하면 참(True)을 돌려준다. 반대로 'email'은 a 딕셔너리 안에 존재하지 않는 Key이므로 거짓(False)을 돌려준다.

## 2. 중복과 순서가 없는 집합

### 집합 자료형은 어떻게 만들까?

집합(set)은 파이썬 2.3부터 지원하기 시작한 자료형으로, 집합에 관련된 것을 쉽게 처리하기 위해 만든 자료형이다.

집합 자료형은 다음과 같이 set 키워드를 사용해 만들 수 있다.

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

위와 같이 set()의 괄호 안에 리스트를 입력하여 만들거나 다음과 같이 문자열을 입력하여 만들 수도 있다.

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

※ 비어 있는 집합 자료형은 s = set()로 만들 수 있다.

### 집합 자료형의 특징

자, 그런데 위에서 살펴본 set("Hello")의 결과가 좀 이상하지 않은가? 분명 "Hello" 문자열로 set 자료형을 만들었는데 생성된 자료형에는 l 문자가 하나 빠져 있고 순서도 뒤죽박죽이다. 그 이유는 set에 다음과 같은 2가지 큰 특징이 있기 때문이다.

- 중복을 허용하지 않는다.
- 순서가 없다(Unordered).

리스트나 튜플은 순서가 있기(ordered) 때문에 인덱싱을 통해 자료형의 값을 얻을 수 있지만 set 자료형은 순서가 없기(unordered) 때문에 인덱싱으로 값을 얻을 수 없다. 이는 마치 02-5에서 살펴본 딕셔너리와 비슷하다. 딕셔너리 역시 순서가 없는 자료형이라 인덱싱을 지원하지 않는다.

만약 set 자료형에 저장된 값을 인덱싱으로 접근하려면 다음과 같이 리스트나 튜플로 변환한 후 해야 한다.

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

```
>>> s1 = set([1,2,3])
>>> l1 = list(s1)
>>> l1
[1, 2, 3]
>>> l1[0]
1
>>> t1 = tuple(s1)
>>> t1
(1, 2, 3)
>>> t1[0]
1
```



## 교집합, 합집합, 차집합 구하기

set 자료형을 정말 유용하게 사용하는 경우는 교집합, 합집합, 차집합을 구할 때이다.

우선 다음과 같이 2개의 set 자료형을 만든 후 따라 해 보자. s1은 1부터 6까지의 값을 가지게 되었고, s2는 4부터 9까지의 값을 가지게 되었다.

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

### 1. 교집합

s1과 s2의 교집합을 구해 보자.

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

"&" 기호를 이용하면 교집합을 간단히 구할 수 있다.

또는 다음과 같이 intersection 함수를 사용해도 동일한 결과를 돌려준다.

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

s2.intersection(s1)을 사용해도 결과는 같다.

### 2. 합집합

합집합은 다음과 같이 구할 수 있다. 이때 4, 5, 6처럼 중복해서 포함된 값은 한 개씩만 표현된다.

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

"|" 기호를 사용한 방법이다.

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

또는 union 함수를 사용하면 된다. 교집합에서 사용한 intersection 함수와 마찬가지로 s2.union(s1)을 사용해도 동일한 결과를 돌려준다.

### 3. 차집합

차집합은 다음과 같이 구할 수 있다.

```
>>> s1 - s2
{1, 2, 3}
>>> s2 - s1
{8, 9, 7}
```

빼기(-) 기호를 사용한 방법이다.

```
>>> s1.difference(s2)
{1, 2, 3}
>>> s2.difference(s1)
{8, 9, 7}
```

difference 함수를 사용해도 차집합을 구할 수 있다.

## 집합 자료형 관련 함수들

### 값 1개 추가하기(add)

이미 만들어진 set 자료형에 값을 추가할 수 있다. 1개의 값만 추가(add)할 경우에는 다음과 같이 한다.

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

### 값 여러 개 추가하기(update)

여러 개의 값을 한꺼번에 추가(update)할 때는 다음과 같이 하면 된다.

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

### 특정 값 제거하기(remove)

특정 값을 제거하고 싶을 때는 다음과 같이 하면 된다.

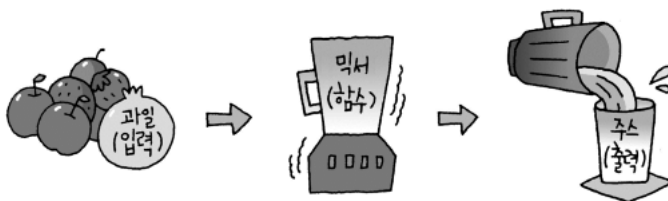
```
>>> s1 = set([1, 2, 3])
>>> s1.remove(2)
>>> s1
{1, 3}
```

## Chapter 7. 특정 기능을 수행하는 사용자 정의 함수와 내장 함수

### 함수란 무엇인가?

함수를 설명하기 전에 믹서를 생각해 보자. 우리는 믹서에 과일을 넣는다. 그리고 믹서를 사용해서 과일을 갈아 과일 주스를 만든다. 우리가 믹서에 넣는 과일은 "입력"이 되고 과일 주스는 "출력(결과값)"이 된다.

그렇다면 믹서는 무엇인가?



(믹서는 과일을 입력받아 주스를 출력하는 함수와 같다.)

우리가 배우려는 함수가 바로 믹서와 비슷하다. 입력값을 가지고 어떤 일을 수행한 다음에 그 결과물을 내어놓는 것, 이것이 바로 함수가 하는 일이다. 우리는 어려서부터 함수에 대해 공부했지만 함수에 관해 깊이 생각해 본 적은 별로 없다. 예를 들어  $y = 2x + 3$ 도 함수이다. 하지만 이를 수학 시간에 배운 직선 그래프만 알고 있지  $x$ 에 어떤 값을 넣었을 때 어떤 변화에 의해서  $y$  값이 나오는지 그 과정에 대해서는 별로 관심을 두지 않았을 것이다.

이제 우리는 함수에 대해 조금 더 생각해 보는 시간을 가져야 한다. 프로그래밍에서 함수는 정말 중요하기 때문이다. 자, 이제 파이썬 함수의 세계로 깊이 들어가 보자.

## 함수를 사용하는 이유는 무엇일까?

프로그래밍을 하다 보면 똑같은 내용을 반복해서 작성하고 있는 자신을 발견할 때가 종종 있다. 이때가 바로 함수가 필요한 때이다. 즉 반복되는 부분이 있을 경우 "반복적으로 사용되는 가치 있는 부분"을 한 용치로 묶어서 "어떤 입력값을 주었을 때 어떤 결과값을 돌려준다"라는 식의 함수로 작성하는 것이 현명하다.

함수를 사용하는 또 다른 이유는 자신이 만든 프로그램을 함수화하면 프로그램 흐름을 일목요연하게 볼 수 있기 때문이다. 마치 공장에서 원재료가 여러 공정을 거쳐 하나의 상품이 되는 것처럼 프로그램에서도 입력한 값이 여러 함수를 거치면서 원하는 결과값을 내는 것을 볼 수 있다. 이렇게 되면 프로그램 흐름도 잘 파악할 수 있고 오류가 어디에서 나는지도 바로 알아차릴 수 있다. 함수를 잘 사용하고 함수를 적절하게 만들 줄 아는 사람이 능력 있는 프로그래머이다.

## 파이썬 함수의 구조

파이썬 함수의 구조는 다음과 같다.

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

def는 함수를 만들 때 사용하는 예약어이며, 함수 이름은 함수를 만드는 사람이 임의로 만들 수 있다. 함수 이름 뒤 괄호 안의 매개변수는 이 함수에 입력으로 전달되는 값을 받는 변수이다. 이렇게 함수를 정의한 다음 if, while, for문 등과 마찬가지로 함수에서 수행할 문장을 입력한다.

간단하지만 많은 것을 설명해 주는 다음 예를 보자.

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

위 함수는 다음과 같이 풀이된다.

"이 함수의 이름(함수 이름)은 add이고 입력으로 2개의 값을 받으며 결과값은 2개의 입력값을 더한 값이다."

여기에서 return은 함수의 결과값을 돌려주는 명령어이다. 먼저 다음과 같이 add 함수를 만들자.

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

이제 직접 add 함수를 사용해 보자.

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

변수 a에 3, b에 4를 대입한 다음 앞에서 만든 add 함수에 a와 b를 입력값으로 넣어 준다. 그리고 변수 c에 add 함수의 결과값을 대입하면 print(c)로 c의 값을 확인할 수 있다.

## 매개변수와 인수

매개변수(parameter)와 인수(arguments)는 혼용해서 사용되는 헛갈리는 용어이므로 잘 기억해 두자. 매개변수는 함수에 입력으로 전달된 값을 받는 변수를 의미하고 인수는 함수를 호출할 때 전달하는 입력값을 의미한다.

```
def add(a, b): # a, b는 매개변수  
    return a+b  
  
print(add(3, 4)) # 3, 4는 인수
```

[같은 의미를 가진 여러 가지 용어들에 주의하자]

프로그래밍을 공부할 때 어려운 부분 중 하나가 용어의 혼용이라고 할 수 있다. 우리는 공부하면서 원서를 보기도 하고 누군가의 번역본을 보기도 하면서 의미는 같지만 표현이 다른 용어를 자주 만나게 된다. 한 예로 입력값을 다른 말로 함수의 인수, 매개변수 등으로 말하기도 하고 결과값을 출력값, 반환 값, 돌려주는 값 등으로 말하기도 한다. 이렇듯 많은 용어가 여러 가지 다른 말로 표현되지만 의미는 동일한 경우가 많다. 따라서 이런 용어를 기억해 놓아야 머리가 덜 아플 것이다.

## 입력값과 결과값에 따른 함수의 형태

함수는 들어온 입력값을 받아 어떤 처리를 하여 적절한 결과값을 돌려준다.

입력값 ---> 함수 ----> 결과값

함수의 형태는 입력값과 결과값의 존재 유무에 따라 4가지 유형으로 나뉜다. 자세히 알아보자.

### 일반적인 함수

입력값이 있고 결과값이 있는 함수가 일반적인 함수이다. 앞으로 여러분이 프로그래밍을 할 때 만들 함수는 대부분 다음과 비슷한 형태일 것이다.

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

다음은 일반 함수의 전형적인 예이다.

```
def add(a, b):  
    result = a + b  
    return result
```

add 함수는 2개의 입력값을 받아서 서로 더한 결과값을 돌려준다.

이 함수를 사용하는 방법은 다음과 같다. 입력값으로 3과 4를 주고 결과값을 돌려받아 보자.

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

이처럼 입력값과 결과값이 있는 함수의 사용법을 정리하면 다음과 같다.

결과값을 받을 변수 = 함수이름(입력인수1, 입력인수2, ...)

### 입력값이 없는 함수

입력값이 없는 함수가 존재할까? 당연히 존재한다. 다음을 보자.

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

say라는 이름의 함수를 만들었다. 그런데 매개변수 부분을 나타내는 함수 이름 뒤의 괄호 안이 비어 있다. 이 함수는 어떻게 사용하는 걸까?

다음은 직접 입력해 보자.

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

위 함수를 쓰기 위해서는 `say()` 처럼 괄호 안에 아무 값도 넣지 않아야 한다. 이 함수는 입력값은 없지만 결과값으로 Hi라는 문자열을 돌려준다. `a = say()` 처럼 작성하면 a에 Hi 문자열이 대입되는 것이다.

이처럼 입력값이 없고 결과값만 있는 함수는 다음과 같이 사용된다.

```
함수이름()
결과값을 받을 변수 = 함수이름()
```

## 결과값이 없는 함수

결과값이 없는 함수 역시 존재한다. 다음 예를 보자.

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

결과값이 없는 함수는 호출해도 돌려주는 값이 없기 때문에 다음과 같이 사용한다.

```
>>> add(3, 4)
3, 4의 합은 7입니다.
```

즉 결과값이 없는 함수는 다음과 같이 사용한다.

```
함수이름(입력인수1, 입력인수2, ...)
```

결과값이 진짜 없는지 확인하기 위해 다음 예를 직접 입력해 보자.

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

아마도 여러분은 '3, 4의 합은 7입니다.'라는 문장을 출력해 주었는데 왜 결과값이 없다는 것인지 의아하게 생각할 것이다. 이 부분이 초보자들이 혼란스러워하는 부분이기도 한데 print문은 함수의 구성 요소 중 하나인 [<수행할 문장>](#)에 해당하는 부분일 뿐이다. 결과값은 당연히 없다. 결과값은 오직 return 명령어로만 돌려받을 수 있다.

이를 확인해 보자. 돌려받을 값을 a 변수에 대입하여 출력해 보면 결과값이 있는지 없는지 알 수 있다.

```
>>> a = add(3, 4)
>>> print(a)
None
```

a 값은 None이다. None이란 거짓을 나타내는 자료형이라고 언급한 적이 있다. add 함수처럼 결과값이 없을 때 `a = add(3, 4)` 처럼 쓰면 함수 add는 반환 값으로 a 변수에 None을 돌려준다. 이것을 가지고 결과값이 있다고 생각하면 곤란하다.

## 입력값도 결과값도 없는 함수

입력값도 결과값도 없는 함수 역시 존재한다. 다음 예를 보자.

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

입력 인수를 받는 매개변수도 없고 return문도 없으니 입력값도 결과값도 없는 함수이다.

이 함수를 사용하는 방법은 단 한 가지이다.

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

즉 입력값도 결과값도 없는 함수는 다음과 같이 사용한다.

```
함수이름()
```

## 매개변수 지정하여 호출하기

함수를 호출할 때 매개변수를 지정할 수도 있다. 다음 예를 보자.

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

앞에서 알아본 add 함수이다. 이 함수를 다음과 같이 매개변수를 지정하여 사용할 수 있다.

```
>>> result = add(a=3, b=7) # a에 3, b에 7을 전달  
>>> print(result)  
10
```

매개변수를 지정하면 다음과 같이 순서에 상관없이 사용할 수 있다는 장점이 있다.

```
>>> result = add(b=5, a=3) # b에 5, a에 3을 전달  
>>> print(result)  
8
```

## 입력값이 몇 개가 될지 모를 때는 어떻게 해야 할까?

입력값이 여러 개일 때 그 입력값을 모두 더해 주는 함수를 생각해 보자. 하지만 몇 개가 입력될지 모를 때는 어떻게 해야 할까? 아마도 난감할 것이다. 파이썬은 이런 문제를 해결하기 위해 다음과 같은 방법을 제공한다.

```
def 함수이름(*매개변수):  
    <수행할 문장>  
    ...
```

일반적으로 볼 수 있는 함수 형태에서 괄호 안의 매개변수 부분이 **\*매개변수**로 바뀌었다.

여러 개의 입력값을 받는 함수 만들기

다음 예를 통해 여러 개의 입력값을 모두 더하는 함수를 직접 만들어 보자. 예를 들어 `add_many(1, 2)` 이면 3을, `add_many(1,2,3)` 이면 6을, `add_many(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)` 이면 55를 돌려주는 함수를 만들어 보자.

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

위에서 만든 add\_many 함수는 입력값이 몇 개이든 상관없다. `*args` 처럼 매개변수 이름 앞에 `*` 을 붙이면 입력값을 전부 모아서 튜플로 만들어 주기 때문이다. 만약 `add_many(1, 2, 3)` 처럼 이 함수를 쓰면 args는 `(1, 2, 3)` 이 되고, `add_many(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)` 처럼 쓰면 args는 `(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)` 이 된다. 여기에서 `*args` 는 임의로 정한 변수 이름이다. `*pey`, `*python` 처럼 아무 이름이나 써도 된다.

※ args는 매개변수를 뜻하는 영어 단어 arguments의 약자이며 관례적으로 자주 사용한다.

### 키워드 파라미터 kwargs

이번에는 키워드 파라미터에 대해 알아보자. 키워드 파라미터를 사용할 때는 매개변수 앞에 별 두 개(\*\*)를 붙인다. 역시 이것도 예제로 알아보자. 먼저 다음과 같은 함수를 작성한다.

```
>>> def print_kwargs(**kwargs):
...     print(kwargs)
... 
```

print\_kwargs 함수는 매개변수 kwargs를 출력하는 함수이다. 이제 이 함수를 다음과 같이 사용해 보자.

```
>>> print_kwargs(a=1)
{'a': 1}
>>> print_kwargs(name='foo', age=3)
{'age': 3, 'name': 'foo'}
```

입력값 a=1 또는 name='foo', age=3 이 모두 딕셔너리로 만들어져서 출력된다는 것을 확인할 수 있다. 즉 \*\*kwargs 처럼 매개변수 이름 앞에 \*\* 을 붙이면 매개변수 kwargs는 딕셔너리가 되고 모든 key=value 형태의 결과값이 그 딕셔너리에 저장된다.

※ 여기에서 kwargs는 keyword arguments의 약자이며 args와 마찬가지로 관례적으로 사용한다.

## 함수의 결과값은 언제나 하나이다

먼저 다음 함수를 만들어 보자.

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

※ add\_and\_mul 함수는 2개의 입력 인수를 받아 더한 값과 곱한 값을 돌려주는 함수이다.

이 함수를 다음과 같이 호출하면 어떻게 될까?

```
>>> result = add_and_mul(3,4)
```

결과값은 a+b 와 a\*b 2개인데 결과값을 받아들이는 변수는 result 하나만 쓰였으니 오류가 발생하지 않을까? 당연한 의문이다. 하지만 오류는 발생하지 않는다. 그 이유는 함수의 결과값은 2개가 아니라 언제나 1개라는 데 있다. add\_and\_mul 함수의 결과값 a+b 와 a\*b 는 튜플값 하나인 (a+b, a\*b) 로 돌려준다.

따라서 result 변수는 다음과 같은 값을 갖게 된다.

```
result = (7, 12)
```

즉 결과값으로 (7, 12)라는 튜플 값을 갖게 되는 것이다.

만약 이 하나의 튜플 값을 2개의 결과값처럼 받고 싶다면 다음과 같이 함수를 호출하면 된다.

```
>>> result1, result2 = add_and_mul(3, 4)
```

이렇게 호출하면 result1, result2 = (7, 12) 가 되어 result1은 7이 되고 result2는 12가 된다.

## 함수 안에서 선언한 변수의 효력 범위

함수 안에서 사용할 변수의 이름을 함수 밖에서도 동일하게 사용한다면 어떻게 될까? 이런 궁금증이 생겼던 독자라면 이번에 확실하게 답을 찾을 수 있을 것이다.

다음 예를 보자.

```
# vartest.py
a = 1
def vartest(a):
    a = a + 1

vartest(a)
print(a)
```

먼저 `a`라는 변수를 생성하고 1을 대입한다. 다음 입력으로 들어온 값에 1을 더해 주고 결과값은 돌려주지 않는 `vartest` 함수를 선언한다. 그리고 `vartest` 함수에 입력값으로 `a`를 주었다. 마지막으로 `a`의 값을 출력하는 `print(a)`를 입력한다. 과연 결과값은 무엇이 나올까?

당연히 `vartest` 함수에서 매개변수 `a`의 값에 1을 더했으니까 2가 출력될 것 같지만 프로그램 소스를 작성해서 실행해 보면 결과값은 1이 나온다. 그 이유는 함수 안에서 새로 만든 매개변수는 함수 안에서만 사용하는 "함수만의 변수"이기 때문이다. 즉 `def vartest(a)`에서 입력값을 전달받는 매개변수 `a`는 함수 안에서만 사용하는 변수이지 함수 밖의 변수 `a`가 아니라는 뜻이다.

따라서 `vartest` 함수는 다음처럼 변수 이름을 `hello`로 한 `vartest` 함수와 완전히 동일하다.

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

즉 함수 안에서 사용하는 매개변수는 함수 밖의 변수 이름과는 전혀 상관이 없다는 뜻이다.

## 함수 안에서 함수 밖의 변수를 변경하는 방법

그렇다면 `vartest`라는 함수를 사용해서 함수 밖의 변수 `a`를 1만큼 증가시킬 수 있는 방법은 없을까? 이 질문에는 2가지 해결 방법이 있다.

### 1. return 사용하기

```
# vartest_return.py
a = 1
def vartest(a):
    a = a + 1
    return a

a = vartest(a)
print(a)
```

첫 번째 방법은 `return`을 사용하는 방법이다. `vartest` 함수는 입력으로 들어온 값에 1을 더한값을 돌려준다. 따라서 `a = vartest(a)`라고 대입하면 `a`가 `vartest` 함수의 결과값으로 바뀐다. 여기에서도 물론 `vartest` 함수 안의 `a` 매개변수는 함수 밖의 `a`와는 다른 것이다.

### 2. global 명령어 사용하기

```
# vartest_global.py
a = 1
def vartest():
    global a
    a = a + 1

vartest()
print(a)
```

두 번째 방법은 `global` 명령어를 사용하는 방법이다. 위 예에서 볼 수 있듯이 `vartest` 함수 안의 `global a` 문장은 함수 안에서 함수 밖의 `a` 변수를 직접 사용하겠다는 뜻이다. 하지만 프로그래밍을 할 때 `global` 명령어는 사용하지 않는 것이 좋다. 왜냐하면 함수는 독립적으로 존재하는 것이 좋기 때문이다. 외부 변수에 종속적인 함수는 그다지 좋은 함수가 아니다. 그러므로 가급적 `global` 명령어를 사용하는 이 방법은 피하고 첫 번째 방법을 사용하기를 권한다.



## lambda

---

lambda는 함수를 생성할 때 사용하는 예약어로 def와 동일한 역할을 한다. 보통 함수를 한줄로 간결하게 만들 때 사용한다. 우리말로는 "람다"라고 읽고 def를 사용해야 할 정도로 복잡하지 않거나 def를 사용할 수 없는 곳에 주로 쓰인다.

사용법은 다음과 같다.

```
lambda 매개변수1, 매개변수2, ... : 매개변수를 이용한 표현식
```

한번 직접 만들어 보자.

```
>>> add = lambda a, b: a+b
>>> result = add(3, 4)
>>> print(result)
7
```

add는 두 개의 인수를 받아 서로 더한 값을 돌려주는 lambda 함수이다. 위 예제는 def를 사용한 다음 함수와 하는 일이 완전히 동일하다.

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

※ lambda 예약어로 만든 함수는 return 명령어가 없어도 결과값을 돌려준다.