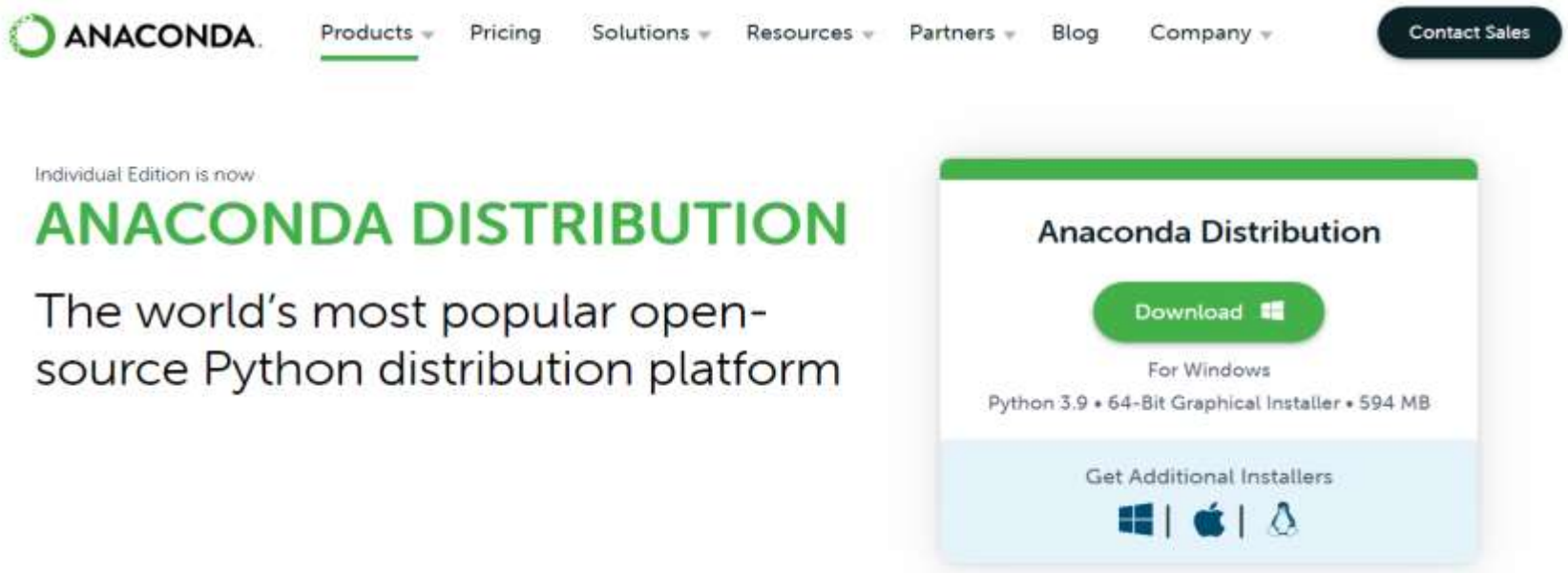
파이썬 기초

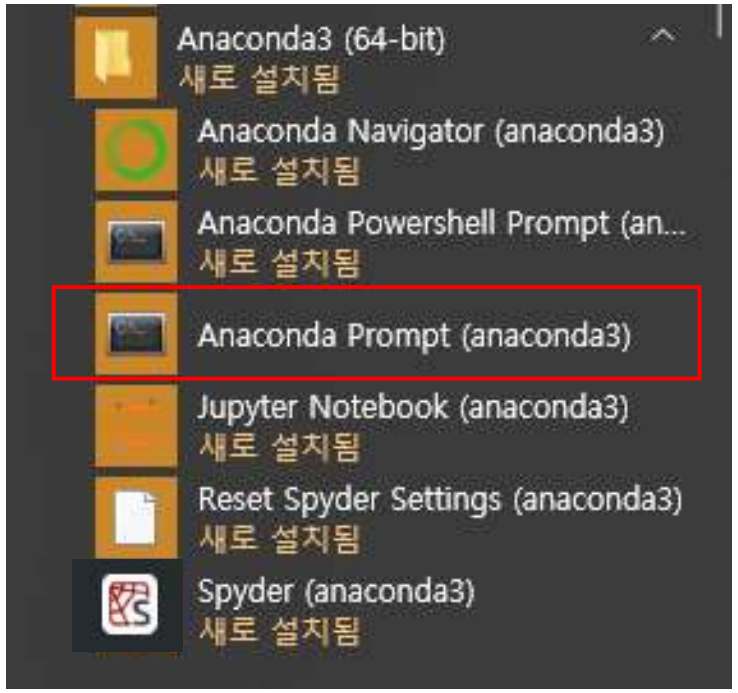
아나콘다 설치

- 데이터 분석 및 기계학습을 위한 패키지들이 기본적으로 포함
- 개발도구 제공: 주피터 노트북, 스파이더
- 설치파일 다운로드
  - https://www.anaconda.com/products/individual
  - PC 운영체제(설정->시스템->정보)에 맞게 다운로드(64-bit)
    - Anaconda3-2024.06-Windows-x86_64.exe //Python 3.12



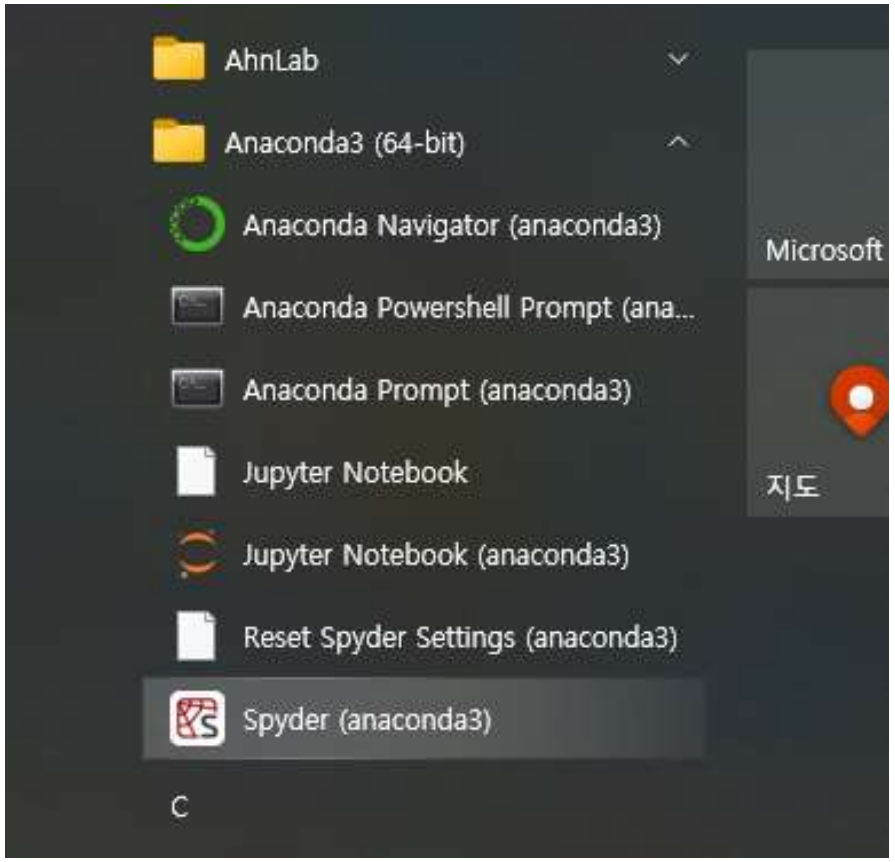
아나콘다 설치 및 팩케지 관리

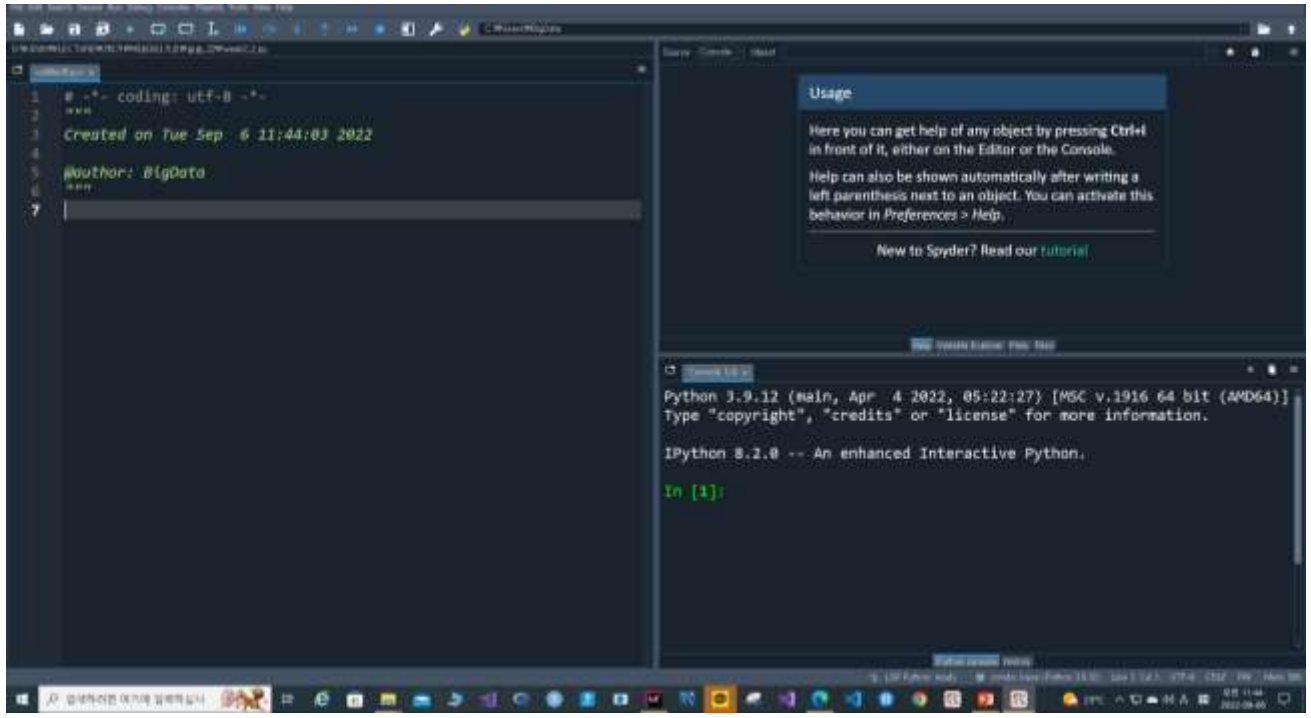
- 설치화면
- default 상태로 next 클릭(설치 폴더는 원하는 위치로 변경)
  - 설치 옵션 중에 Anaconda3를 Path environment에 추가하도록 선택할 것
- 설치가 끝난 후 시작메뉴 창에서 Anaconda Prompt 실행



- >python --version
  - #아나콘다에 포함되어 있는 파이썬의 버전 확인
- >conda --version #아나콘다 버전 확인
- >conda update –n base conda # 콘다 자체 업그레이드
- >conda update --all
  - # 설치된 파이썬 팩케지를 모두 최신으로 업그레이드

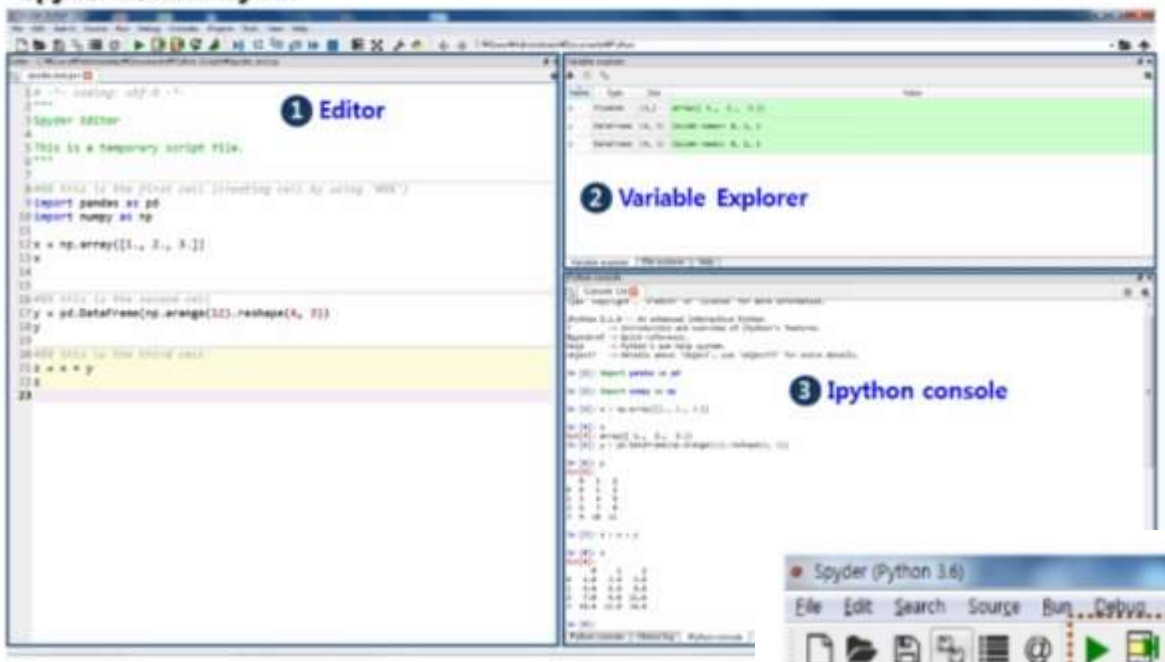
아나콘다(spyder) 실행





아나콘다 스파이더 실행

Spyder default layout



Spyder : Cell (#%%)

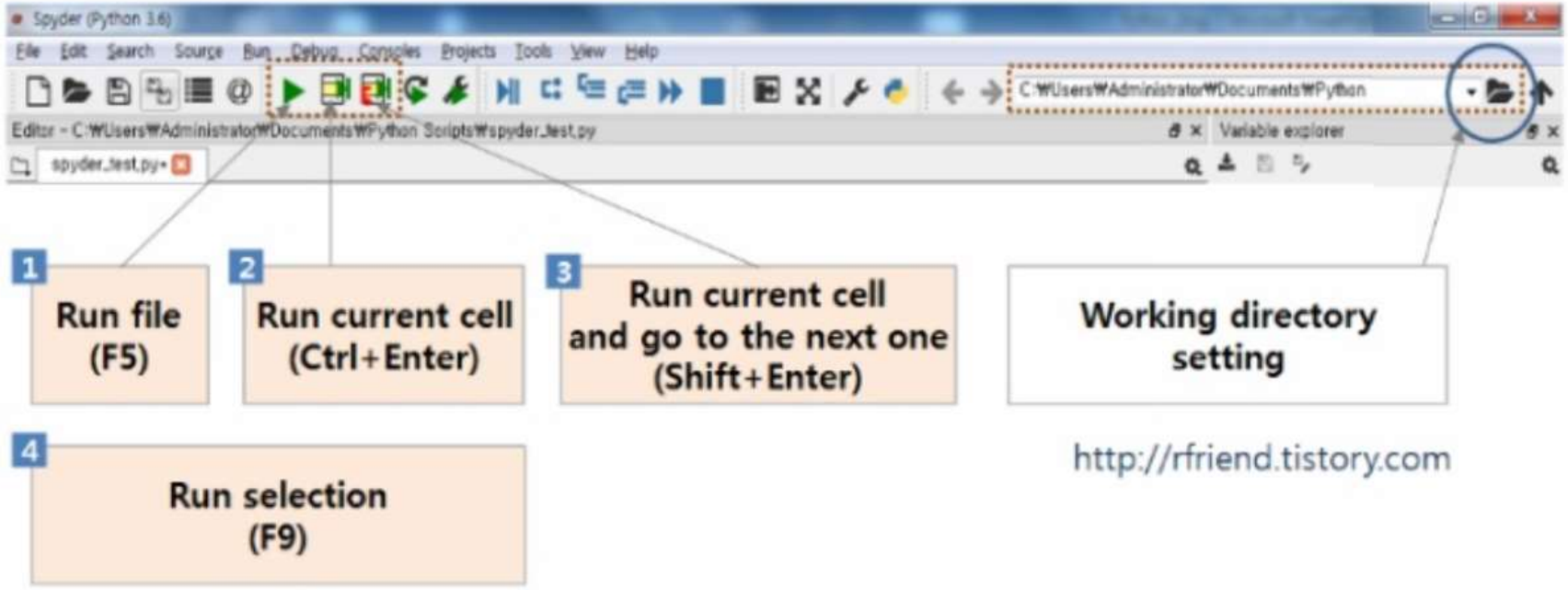
```
#%% (1) this is the first cell (creating cell by using '#%%')
import pandas as pd
import numpy as np

x = np.array([1., 2., 3.])
x


#%% (2) this is the second cell
y = pd.DataFrame(np.arange(12).reshape(4, 3))
y

#%% (3) this is the third cell
z = x + y
z
```

Cell 1

Cell 2

Cell 3

여기서 cell 은 '#%%' 로 구분이 되며, 위/아래로 '선(line)'으로 구분이 된 덩어리를 말합니다.



Python 기초

- 기본 데이터형: 정수형, 실수형, 불형, 문자열, 리스트, 튜플, 딕셔너리, 집합 등
- Python에서
  - 자료형

```
>>> type(10)
<class 'int'>
>>> type(2.718)
<class 'float'>
>>> type("hello")
<class 'str'>
```

  - >> type(x)
  - >> type(y)
  - >> type(True)

변수 선언

```
>>> x = 100      # 변수에 값 대입
>>> print(x)
100
>>> y = 3.14
>>> x * y
```

>>> z = True
>>> type(z)

변수 선언할 때 자료형을 명시하지 않는다

- Spyder에서는
  - print(x*y)
  - print(type(2.718))

Python 기초

1.3.4 리스트

여러 데이터를 **리스트**list로도 정리할 수 있습니다.

```
>>> a = [1, 2, 3, 4, 5] # 리스트 생성
>>> print(a)  # 리스트의 내용 출력
[1, 2, 3, 4, 5]
>>> len(a)    # 리스트의 길이 출력
5
>>> a[0]      # 첫 원소에 접근
1
>>> a[4]      # 다섯 번째 원소에 접근
5
>>> a[4] = 99 # 값 대입
>>> print(a)
[1, 2, 3, 4, 99]
```

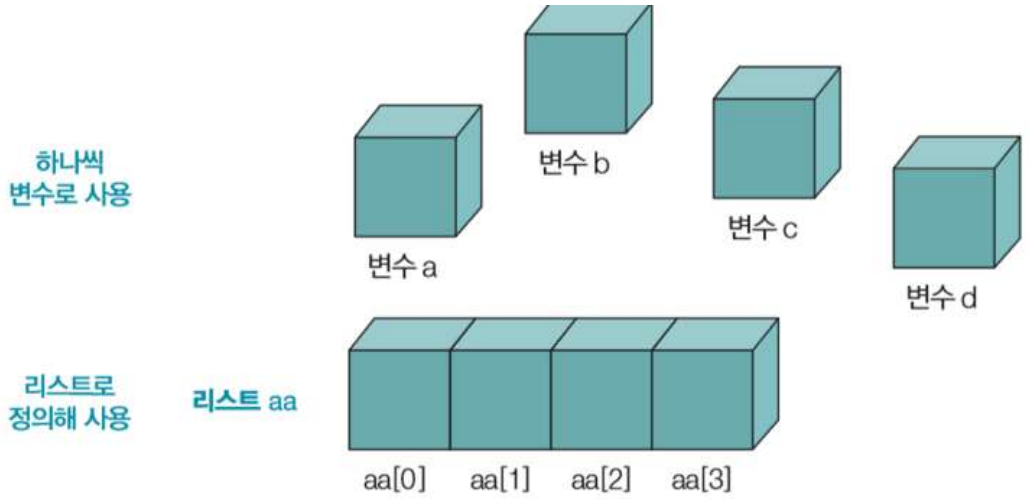


그림 7-1 **리스트의 개념**

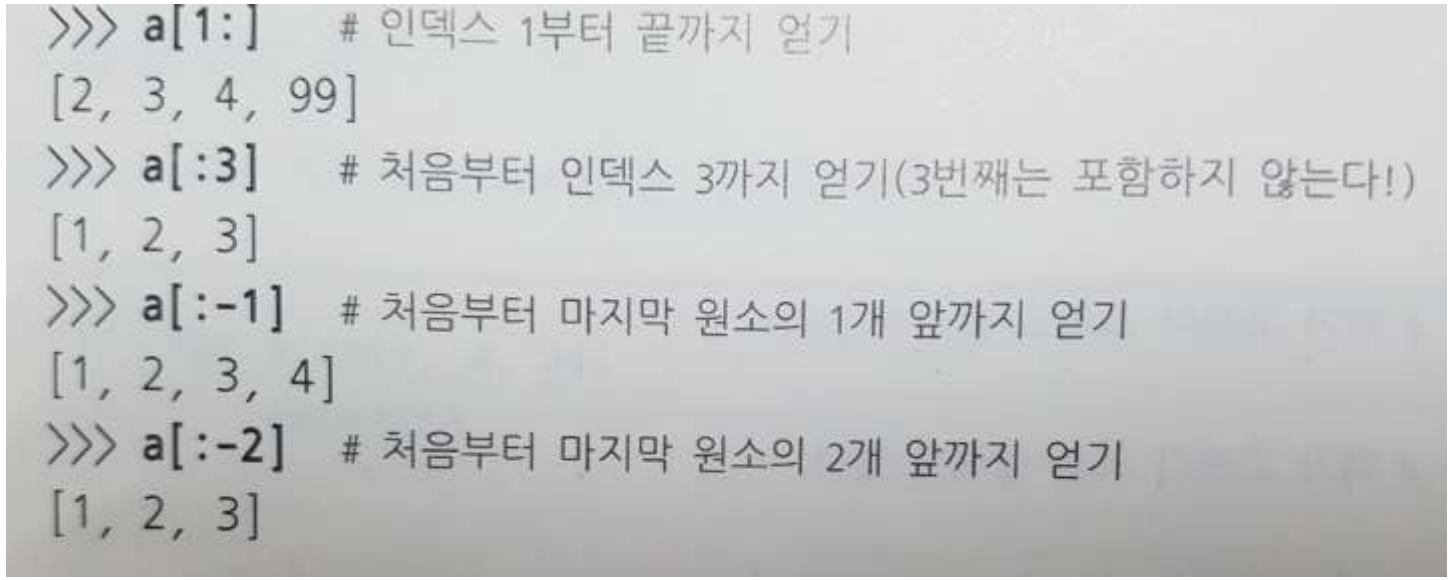
Tip • C/C++나 자바 같은 프로그래밍 언어에는 리스트가 없음, 리스트와 비슷한 개념인 배열(Array)을 사용. 리스트는 정수, 문자열, 실수 등 서로 다른 데이터형도 하나로 묶을 수 있지만, 배열은 동일한 데이터형만 묶을 수 있다. 정수 배열은 정수로만 묶어서 사용

파이썬 for Beginner -한빛아카데미 참조


```
>>>a[4]="학교"
>>>print(a)
[1, 2, 3, 4, '학교']
```

Python 기초

- 슬라이싱
  - a=[1, 2, 3, 4, 99]
  - a [0:2] # 인덱스 0부터 2까지(2는 포함되지 않음)
    - 출력결과=>[1, 2]



■ 리스트 조작 함수

표 7-1 **리스트 조작 함수**

함수	설명	사용법
append()	리스트 맨 뒤에 항목을 추가한다.	리스트명.append(값)
pop()	리스트 맨 뒤의 항목을 빼낸다(리스트에서 해당 항목이 삭제된다).	리스트명.pop()
sort()	리스트의 항목을 정렬한다.	리스트명.sort()
reverse()	리스트 항목의 순서를 역순으로 만든다.	리스트명.reverse()
index()	지정한 값을 찾아 해당 위치를 반환한다.	리스트명.index(찾을값)
insert()	지정된 위치에 값을 삽입한다.	리스트명.insert(위치, 값)
remove()	리스트에서 지정한 값을 삭제한다. 단 지정한 값이 여러 개면 첫 번째 값만 지운다.	리스트명.remove(지울값)
extend()	리스트 뒤에 리스트를 추가한다. 리스트의 더하기(+) 연산과 기능이 동일하다.	리스트명.extend(추가할리스트)
count()	리스트에서 해당 값의 개수를 센다.	리스트명.count(찾을값)
clear()	리스트의 내용을 모두 지운다.	리스트명.clear()
del()	리스트에서 해당 위치의 항목을 삭제한다.	del(리스트명[위치])
len()	리스트에 포함된 전체 항목의 개수를 센다.	len(리스트명)
copy()	리스트의 내용을 새로운 리스트에 복사한다.	새리스트=리스트명.copy()
sorted()	리스트의 항목을 정렬해서 새로운 리스트에 대입한다.	새리스트=sorted(리스트)

파이썬 for Beginner -한빛아카데미 참조

Code07-05.py

```
1  myList = [30, 10, 20]
2  print("현재 리스트 : %s" % myList)
3  print("현재 리스트: {}" .format( myList))
4  myList.append(40)
5  print("append(40) 후의 리스트 : %s" % myList)
6
7  print("pop()으로 추출한 값 : %s" % myList.pop())
8  print("pop() 후의 리스트 : %s" % myList)
9
10 myList.sort()
11 print("sort() 후의 리스트 : %s" % myList)
12
13 myList.reverse()
14 print("reverse() 후의 리스트 : %s" % myList)
15
16 print("20값의 위치 : %d" % myList.index(20))
17
18 myList.insert(2, 222)
19 print("insert(2, 222) 후의 리스트 : %s" % myList)
20
21 myList.remove(222)
22 print("remove(222) 후의 리스트 : %s" % myList)
23
24 myList.extend([77, 88, 77])
25 print("extend([77, 88, 77]) 후의 리스트 : %s" % myList)
26
27 print("77값의 개수 : %d" % myList.count(77))
```

출력 결과

```
현재 리스트 : [30, 10, 20]
append(40) 후의 리스트 : [30, 10, 20, 40]
pop()으로 추출한 값 : 40
pop() 후의 리스트 : [30, 10, 20]
sort() 후의 리스트 : [10, 20, 30]
reverse() 후의 리스트 : [30, 20, 10]
20값의 위치 : 1
insert(2, 222) 후의 리스트 : [30, 20, 222, 10]
remove(222) 후의 리스트 : [30, 20, 10]
extend([77, 88, 77]) 후의 리스트 : [30, 20, 10, 77, 88, 77]
77값의 개수 : 2
```

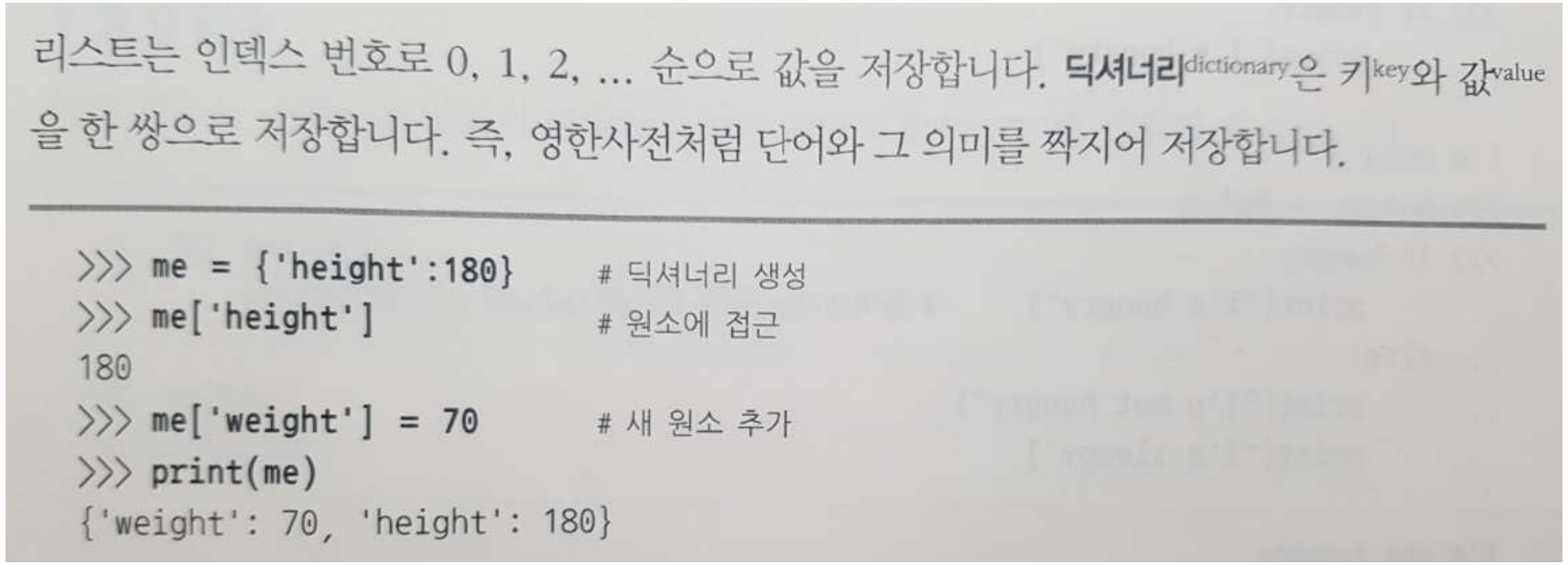
10행 : '리스트.sort()'는 리스트 자체 정렬 'sorted(리스트)'는 리스트는 그대로 두고 정렬된 결과만 반환

18행 : myList.insert(2, 222)에서 2는 myList[2]의 위치를 의미, 리스트는 0번부터 시작 하므로 세 번째 위치가 뒤로 밀리고 그 자리에 222가 삽입

파이썬 for Beginner -한빛아카데미 참조

Python 기초

1.3.5 딕셔너리



리스트는 인덱스 번호로 0, 1, 2, ... 순으로 값을 저장합니다. **딕셔너리**dictionary은 키key와 값value을 한 쌍으로 저장합니다. 즉, 영한사전처럼 단어와 그 의미를 짝지어 저장합니다.

```
>>> me = {'height':180}      # 딕셔너리 생성
>>> me['height']             # 원소에 접근
180
>>> me['weight'] = 70        # 새 원소 추가
>>> print(me)
{'weight': 70, 'height': 180}
```

딕셔너리의 개념

쌍 2개가 하나로 묶인 자료구조

예 : 'apple:사과'처럼 의미 있는 두 값을 연결해 구성

- 다른 프로그래밍 언어에서는 해시 테이블(Hash table), 맵(map)이라 함

중괄호 { }로 묶어 구성, 키(Key)와 값(Value)의 쌍으로 구성

```
딕셔너리변수 = {키1:값1, 키2:값2, 키3:값3, …}
```

파이썬 for Beginner -한빛아카데미 참조

딕셔너리의 생성

```
dic1 = {1 : 'a', 2 : 'b', 3 : 'c'}
dic1
```

출력 결과

```
{1 : 'a', 2 : 'b', 3 : 'c'}
```

- 키와 값을 반대로

```
dic2 = {'a': 1, 'b': 2, 'c': 3}
dic2
```

출력 결과

```
{'a': 1, 'b': 2, 'c': 3}
```

- 키와 값은 사용자가 지정하는 것이지 규정은 없음
- 주의할 점 : 딕셔너리에는 순서가 없어 생성한 순서대로 딕셔너리가 구성되어 있다는 보장 없음

- 여러 정보의 딕셔너리 표현할 때 유용

키	값
학번	1000
이름	홍길동
학과	컴퓨터학과

```
student1 = {'학번' : 1000, '이름': '홍길동', '학과': '컴퓨터학과'}
student1
```

출력 결과

```
{'학번': 1000, '이름': '홍길동', '학과': '컴퓨터학과'}
```

- student1에 연락 처 추가

```
student1['연락처'] = '010-1111-2222'
student1
```

출력 결과

```
{'학번': 1000, '이름': '홍길동', '학과': '컴퓨터학과', '연락처': '010-1111-2222'}
```

파이썬 for Beginner -한빛아카데미 참조

- 학과 수정

```
student1['학과'] = '파이썬학과'
student1
```

출력 결과

```
{'학번': 1000, '이름': '홍길동', '학과': '파이썬학과', '연락처': '010-1111-2222'}
```

- student1의 학과 삭제

```
del(student1['학과'])
student1
```

출력 결과

```
{'학번': 1000, '이름': '홍길동', '연락처': '010-1111-2222'}
```

- 동일한 키를 갖는 딕셔너리를 생성하는 것이 아니라 마지막에 있는 키가 적용

```
student1 = {'학번': 1000, '이름': '홍길동', '학과': '파이썬학과', '학번': 2000}
student1
```

출력 결과

```
{'학번': 2000, '이름': '홍길동', '학과': '파이썬학과'}
```

파이썬 for Beginner -한빛아카데미 참조

딕셔너리의 사용

- 키로 값에 접근하는 코드

```
student1['학번']
student1['이름']
student1['학과']
```

출력 결과

```
2000
'홍길동'
'파이썬학과'
```

- 딕셔너리명.keys()는 딕셔너리의 모든 키 반환

```
student1.keys()
```

출력 결과

```
dict_keys(['학번', '이름', '학과'])
```

파이썬 for Beginner -한빛아카데미 참조

딕셔너리명.values() 함수는 딕셔너리의 모든 값을 리스트로 만들어 반환

- 딕셔너리명.values() 함수도 출력 결과의 dict_values가 보기 싫으면 list(딕셔너리명.values()) 함수 사용

```
student1.values()
```

출력 결과

```
dict_values([2000, '홍길동', '파이썬학과'])
```

딕셔너리 안에 해당 키가 있는지 없는지는 in을 사용해 확인

- 딕셔너리에 키가 있다면 True를 반환하고, 없다면 False를 반환

```
'이름' in student1
'주소' in student1
```

출력 결과

```
True
False
```

- for 문을 활용해 딕셔너리의 모든 값을 출력하는 코드

```
1  singer = {}
2
3  singer['이름'] = '트와이스'
4  singer['구성원 수'] = 9
5  singer['데뷔'] = '서바이벌 식스틴'
6  singer['대표곡'] = 'SIGNAL'
7
8  for k in singer.keys() :
9      print('%s --> %s' % (k, singer[k]))
```

1행 : 빈 딕셔너리를 준비

3~6행 : 쌍을 만들어 딕셔너리에 추가

8~9행 : 모든 키와 값을 출력

출력 결과

```
이름 --> 트와이스
구성원 수 --> 9
데뷔 --> 서바이벌 식스틴
대표곡 --> SIGNAL
```

파이썬 for Beginner -한빛아카데미 참조

Python 기초

자료형 bool

```
>>> hungry = True          # 배가 고프다.
>>> sleepy = False         # 졸리지 않다.
```

>>> type(hungry)

```
<class 'bool'>
>>> not hungry
False
>>> hungry and sleepy    # 배가 고프다 그리고 졸리지 않다.
False
>>> hungry or sleepy     # 배가 고프다 또는 졸리지 않다.
True
```

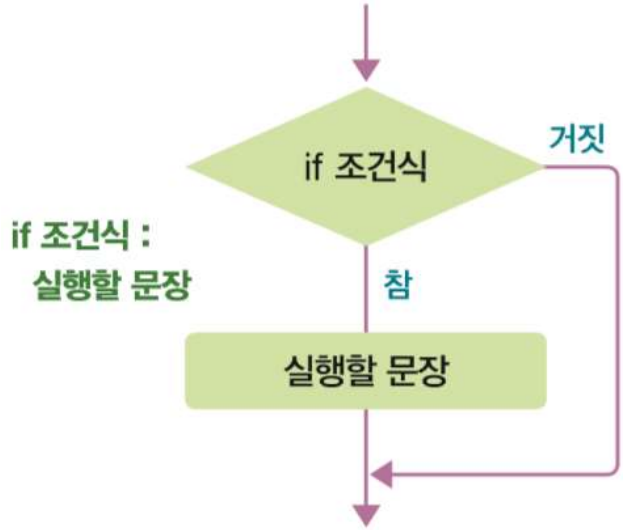
Python 기초

1.3.7 if 문

조건에 따라서 달리 처리하려면 **if/else 문**을 사용합니다.

```
>>> hungry = True
>>> if hungry:
...     print("I'm hungry")
...
I'm hungry
>>> hungry = False
>>> if hungry:
...     print("I'm hungry")      # 들여쓰기는 공백 문자로
... else:
...     print("I'm not hungry")
...     print("I'm sleepy")
...
I'm not hungry
I'm sleepy
```

if 문




```
a = 99
if a < 100 :
    print("100보다 작군요.")
```

출력 결과

```
100보다 작군요.
```

- 파이썬은 들여쓰기가 매우 중요.
- if 문 다음에 '실행할 문장'은 if 문 다음 줄에서 들여쓰기를 해서 작성.
- 들여쓰기 할 때는 Tab 보다 Space Bar 를 눌러 4칸 정도로 들여쓰기 권장,
- 대화형 모드에서는 '실행할 문장' 모두 끝나고 Enter 2번 눌러야 if 문이 끝나는 것으로 간주

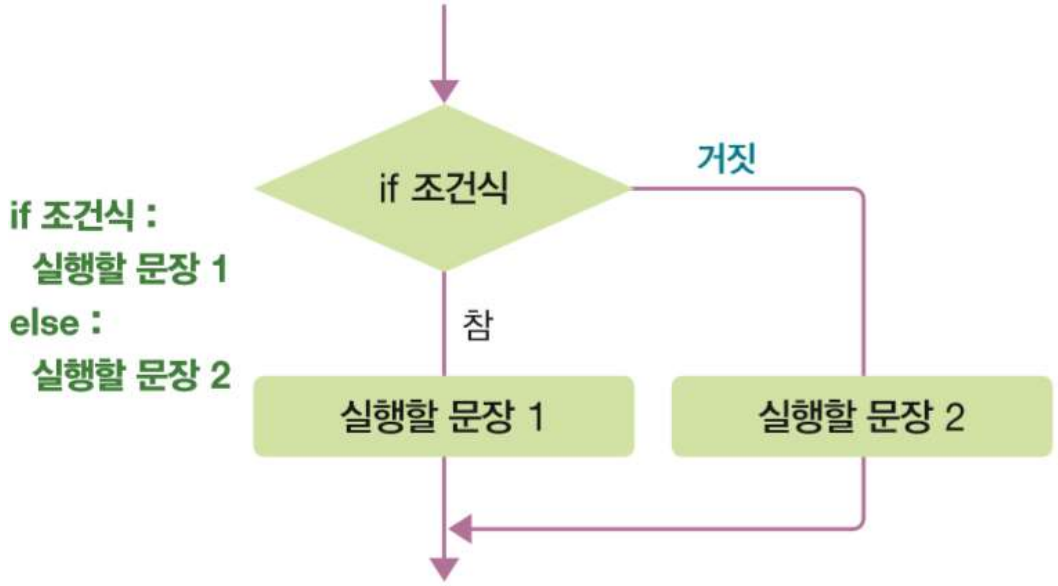
- if 문에서 두 문장 이상을 실행하고자 할 때

```
1  a = 200
2
3  if a < 100 :
4      print("100보다 작군요.")
5      print("거짓이므로 이 문장은 안 보이겠죠?")
6
7  print("프로그램 끝")
```

출력 결과

```
프로그램 끝
```

- **if~else 문**




```
1  a = 200
2
3  if a < 100 :
4      print("100보다 작군요.")
5  else :
6      print("100보다 크군요.")
```

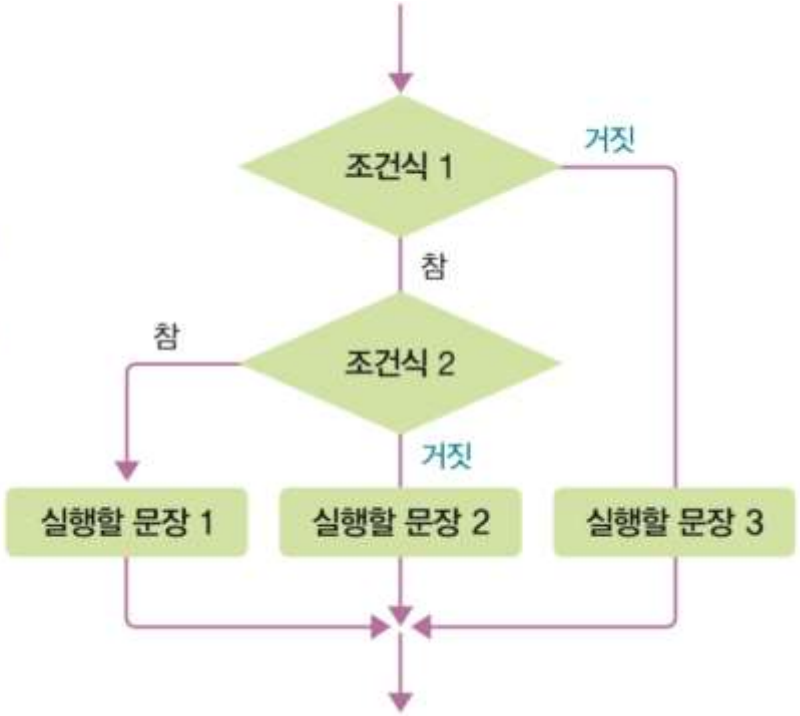
출력 결과

```
100보다 크군요.
```

if~else~if~else 문

- if 문을 한 번 실행한 후 그 결과에서 if 문을 다시 실행하는 것

```
if 조건식 1 :
    if 조건식 2 :
        실행할 문장 1
    else :
        실행할 문장 2
else :
    실행할 문장 3
```




```
1  a = 75
2
3  if a > 50 :
4      if a < 100 :
5          print("50보다 크고 100보다 작군요.")
6      else :
7          print("와~~ 100보다 크군요.")
8  else :
9      print("에고~ 50보다 작군요.")
```

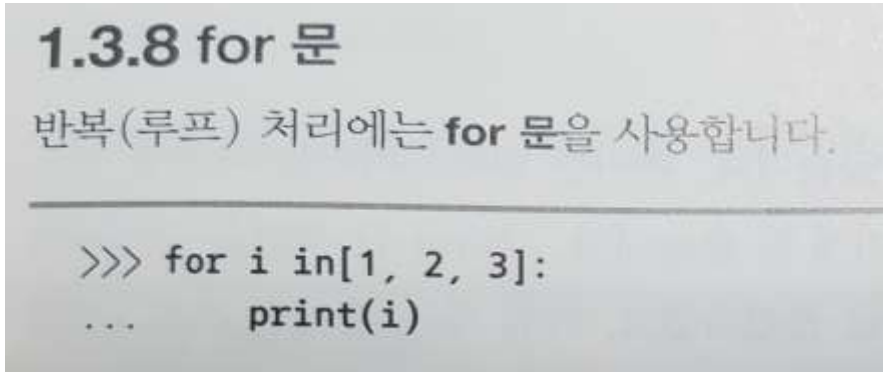
출력 결과

```
50보다 크고 100보다 작군요.
```

if~elif~else 문

```
1  score = int(input("점수를 입력하세요 : "))
2
3  if score >= 90 :
4      print("A")
5  elif score >= 80 :
6      print("B")
7  elif score >= 70 :
8      print("C")
9  elif score >= 60 :
10     print("D")
11 else :
12     print("F")
13
14 print("학점입니다. ^^")
```

Python 기초



1.3.8 for 문

반복(루프) 처리에는 **for 문**을 사용합니다.

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

for 문의 개념

- 기본 형식

```
for 변수 in range(시작값, 끝값+1, 증가값) :
    이 부분을 반복
```

range(3)은 range(0, 3, 1)과 같다

- 예 : range() 함수 사용과 내부적 변경

```
for i in range(0, 3, 1) :
    print("안녕하세요? for 문을 공부 중입니다. ^^")
```

```
for i in [0, 1, 2] :
    print("안녕하세요? for 문을 공부 중입니다. ^^")
```

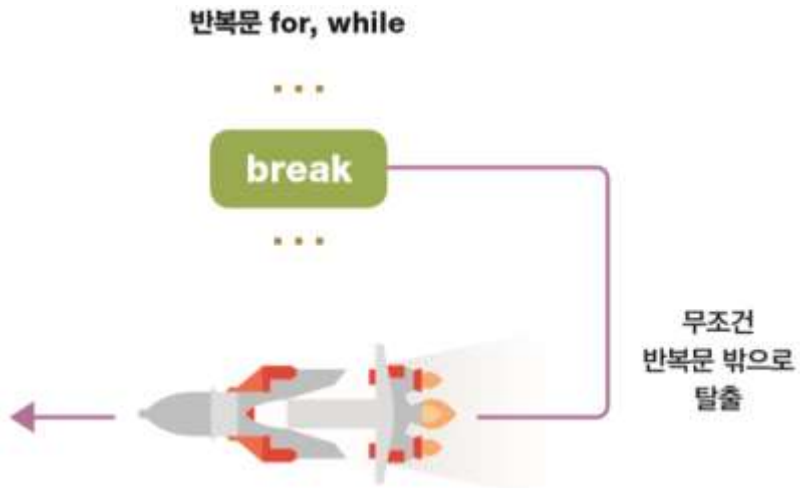
- 중첩 for 문의 기본 형식

```
for i in range(0, 3, 1) :
    for k in range(0, 2, 1) :
        print("파이썬은 꿀잼입니다. ^^ (i값 : %d, k값 : %d)" % (i, k))
```

출력 결과

```
파이썬은 꿀잼입니다. ^^ (i값 : 0, k값 : 0)
파이썬은 꿀잼입니다. ^^ (i값 : 0, k값 : 1)
파이썬은 꿀잼입니다. ^^ (i값 : 1, k값 : 0)
파이썬은 꿀잼입니다. ^^ (i값 : 1, k값 : 1)
파이썬은 꿀잼입니다. ^^ (i값 : 2, k값 : 0)
파이썬은 꿀잼입니다. ^^ (i값 : 2, k값 : 1)
```

- 반복문을 탈출시키는 break 문




```
for i in range(1, 100) :
    print("for 문을 %d번 실행했습니다." % i)
    break
```

출력 결과

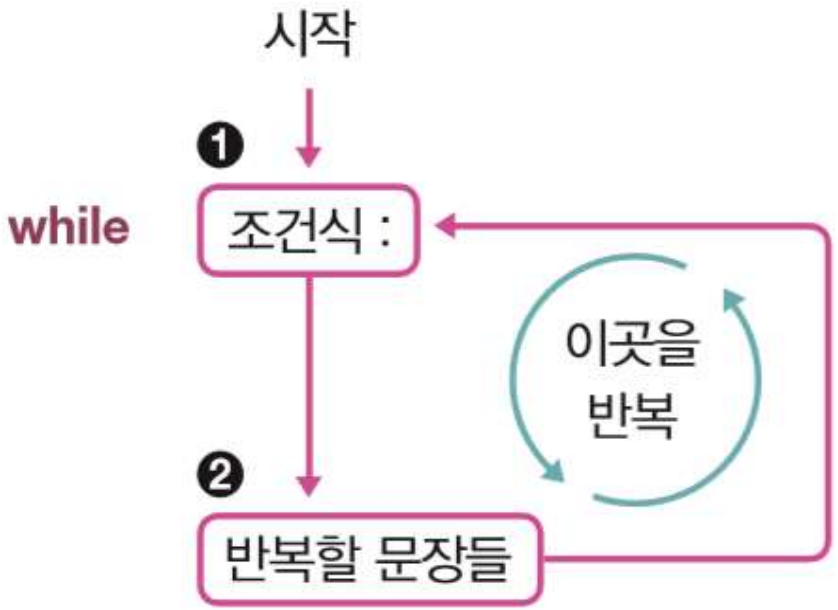
```
for 문을 1번 실행했습니다.
```

for 문과 while 문 비교

- for 문의 형식

```
for 변수 in range(시작값, 끝값+1, 증가값)
```

- for 문은 반복할 횟수를 range() 함수에서 결정 후 그 횟수만큼 반복,
- while 문은 반복 횟수를 결정하기보다는 조건식이 참일 때 반복하는 방식




```
i = 0
while i < 3 :
    print("%d : 안녕하세요? while 문을 공부 중입니다. ^^" % i)
    i = i + 1
```

출력 결과

```
0 : 안녕하세요? while 문을 공부 중입니다. ^^
1 : 안녕하세요? while 문을 공부 중입니다. ^^
2 : 안녕하세요? while 문을 공부 중입니다. ^^
```

Python 기초

1.3.9 함수

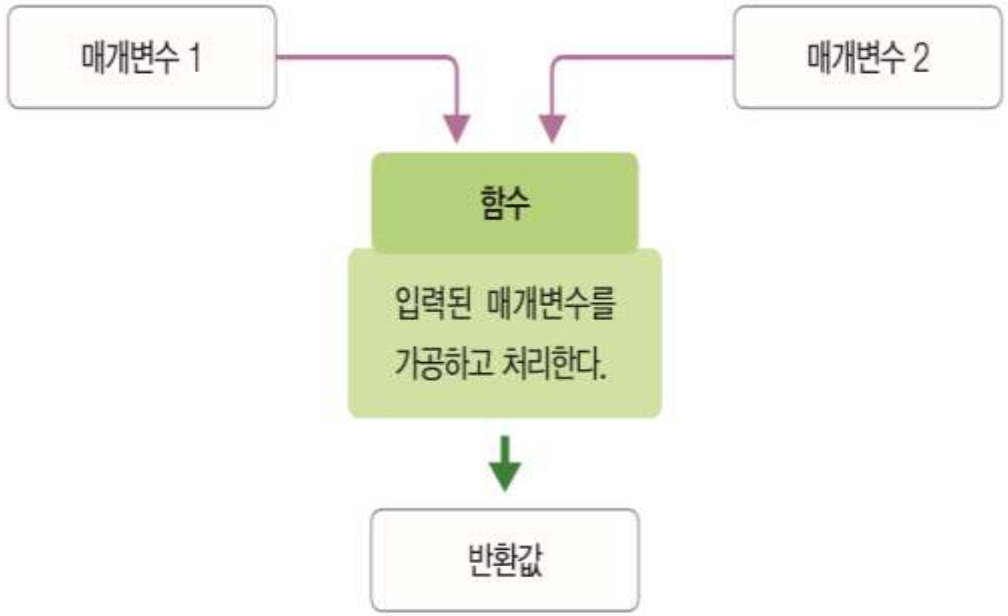
특정 기능을 수행하는 일련의 명령들을 묶어 하나의 **함수**function로 정의할 수 있습니다.

```
>>> def hello():
...     print("Hello World!")
...
>>> hello()
Hello World!
```

함수는 인수를 취할 수 있습니다. 또한, + 연산자를 사용하여 문자열을 이어 붙일 수 있습니다.

```
>>> def hello(object):
...     print("Hello " + object + "!")
...
>>> hello("cat")
Hello cat!
```

함수의 기본 형식

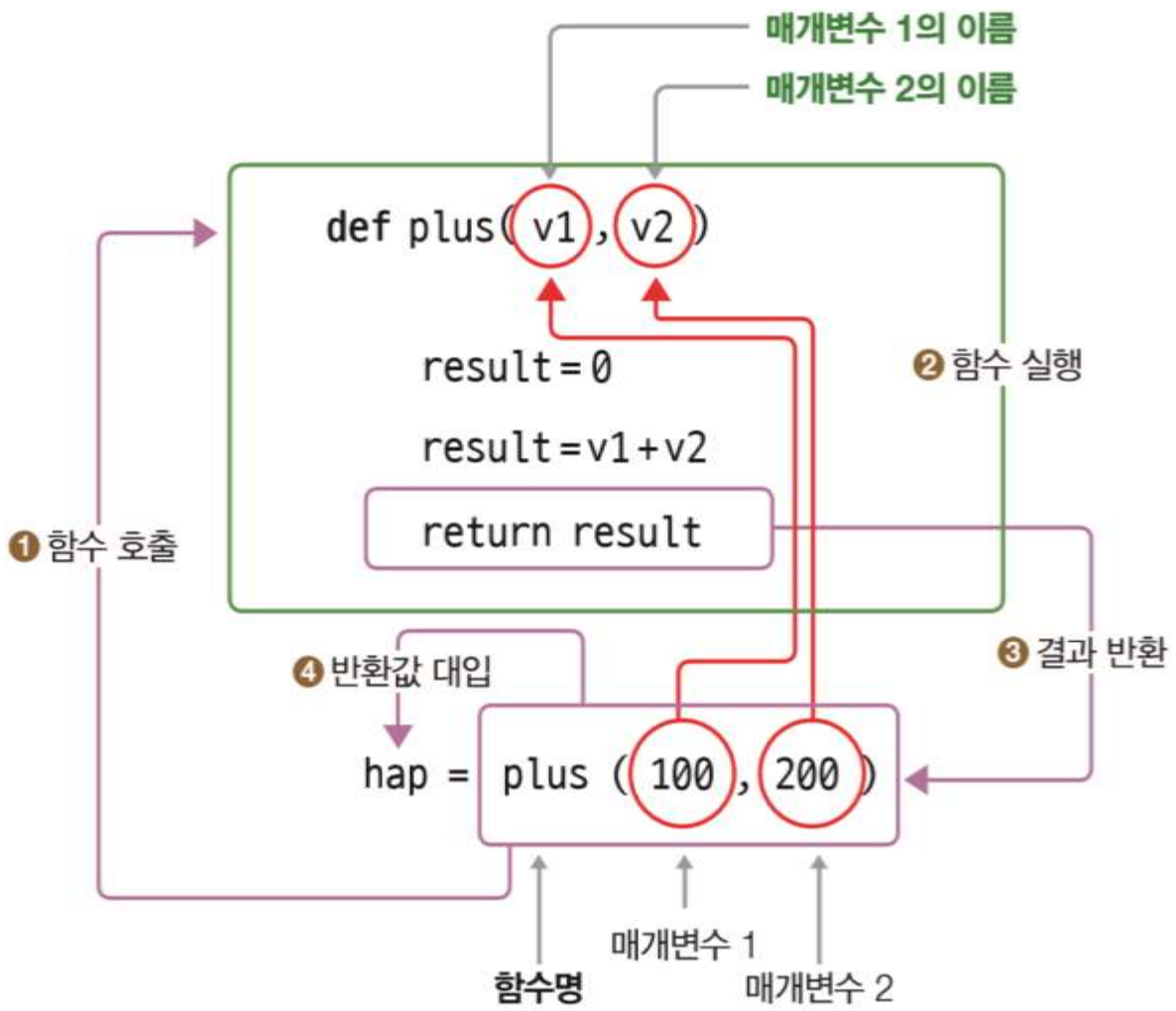


함수의 형식과 활용

```
1  ## 함수 선언 부분 ##
2  def plus(v1, v2) :
3      result = 0
4      result = v1 + v2
5      return result
6
7  ## 전역 변수 선언 부분 ##
8  hap = 0
9
10 ## 메인 코드 부분 ##
11 hap = plus(100, 200)
12 print("100과 200의 plus() 함수 결과는 %d" % hap)
```

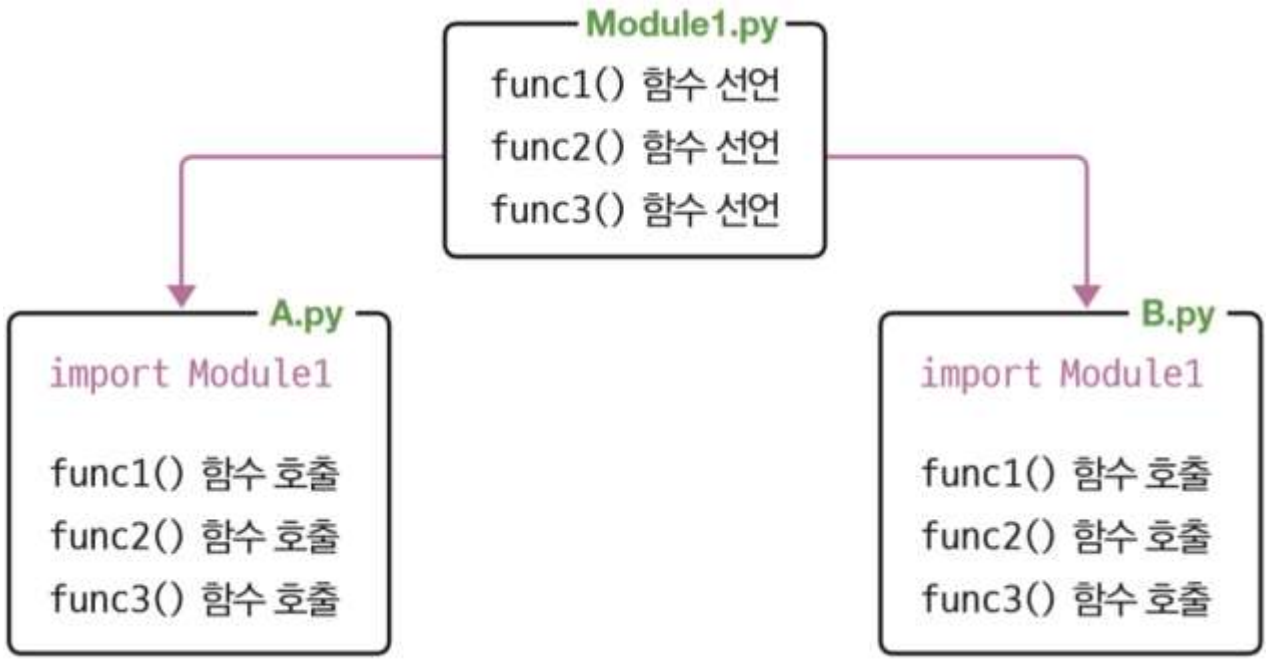
출력 결과

```
100과 200의 plus() 함수 결과는 300
```



모듈의 생성과 사용

- 모듈 : 함수의 집합



Module1.py

```
1  ## 함수 선언 부분 ##
2  def func1() :
3      print("Module1.py의 func1()이 호출됨.")
4
5  def func2() :
6      print("Module1.py의 func2()가 호출됨.")
7
8  def func3() :
9      print("Module1.py의 func3()이 호출됨.")
```

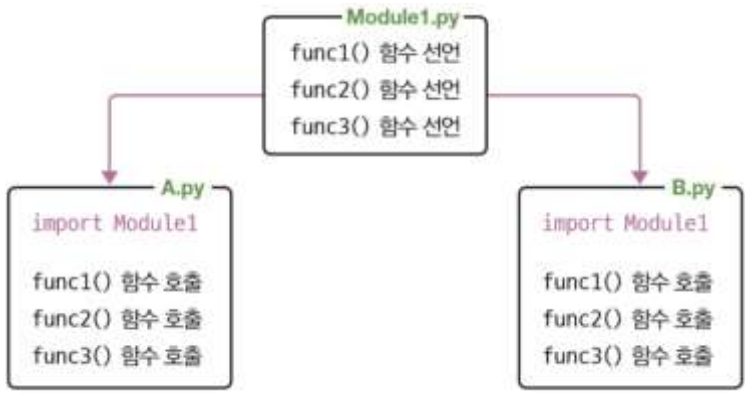
모듈의 생성과 사용

A.py

```
1  import Module1
2
3  ## 메인 코드 부분 ##
4  Module1.func1()
5  Module1.func2()
6  Module1.func3()
```

출력 결과

```
Module1.py의 func1()이 호출됨.
Module1.py의 func2()가 호출됨.
Module1.py의 func3()이 호출됨.
```



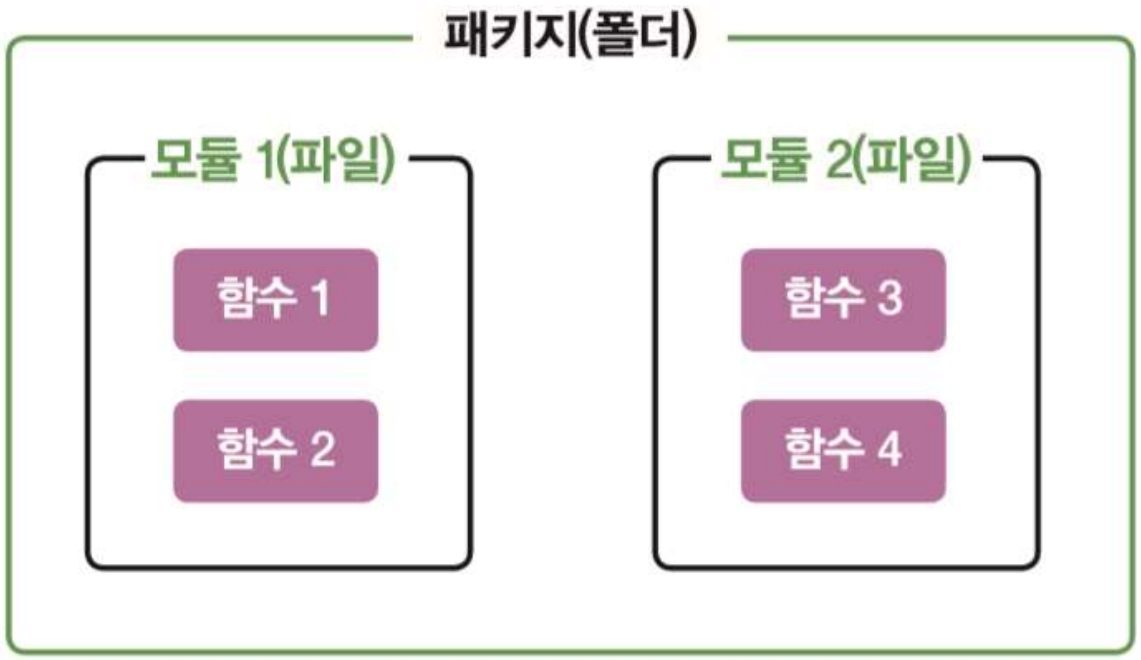
- 모듈명을 생략하고 함수명만 쓸 때 1행 형식

```
from 모듈명 import 함수1, 함수2, 함수3
또는
from 모듈명 import *
```

B.py

```
1  from Module1 import func1, func2, func3    # 또는 from Module1 import *
2
3  ## 메인 코드 부분 ##
4  func1()
5  func2()
6  func3()
```

- 패키지
  - 모듈이 하나의 *.py 파일 안에 함수가 여러 개 들어 있는 것이라면,
  - 패키지(Package)는 여러 모듈을 모아 놓은 것으로 폴더의 형태로 나타냄
  - 모듈을 주제별로 분리할 때 주로 사용



Python 기초: 클래스

```
class 클래스 이름:
    def __init__(self, 인수, ...):      # 생성자
        ...
    def 메서드 이름 1(self, 인수, ...):    # 메서드 1
        ...
    def 메서드 이름 2(self, 인수, ...):    # 메서드 2
        ...
```

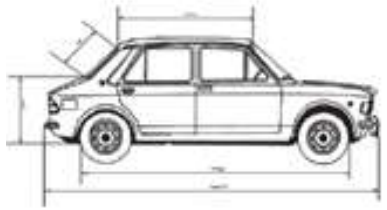
클래스 정의에는 **__init__**라는 특별한 메서드가 있는데, 클래스를 초기화하는 방법을 정의합니다. 이 초기화용 메서드를 **생성자**constructor라고도 하며, 클래스의 인스턴스가 만들어질 때 한 번만 불립니다. 또, 파이썬에서는 메서드의 첫 번째 인수로 자신(자신의 인스턴스)을 나타내는 self를 명시적으로 쓰는 것이 특징입니다(다른 언어를 쓰던 사람은 이처럼 self를 쓰는 규칙

클래스의 개념

- 클래스의 모양과 생성

```
class 클래스명 :
    # 이 부분에 관련 코드 구현
```

- 자동차를 클래스로 구현



class 자동차 :
  # 자동차의 속성
  색상
  속도
  # 자동차의 기능
  속도 올리기( )
  속도 내리기( )

- 자동차 클래스의 개념을 실제 **코**드로 구현
  - 자동차의 속성은 지금까지 사용 한 변수처럼 생성(필드(Field))
  - 자동차의 기능은 지금까지 사용한 함수 형식으로 구현
  - 클래스 안에서 구현된 함수는 함수라고 하지 않고 메서드라고 함.

```
## 클래스 선언 부분 ##
class Car :
    color = ""
    speed = 0

    def __init__(self) :
        self.color = "빨강"
        self.speed = 0

    def upSpeed(self, value) :
        self.speed += value

    def downSpeed(self, value) :
        self.speed -= value

## 메인 코드 부분 ##
myCar1 = Car()
myCar2 = Car()

print("자동차1의 색상은 %s이며, 현재 속도는 %dkm입니다." % (myCar1.color, myCar1.speed))
print("자동차2의 색상은 %s이며, 현재 속도는 %dkm입니다." % (myCar2.color, myCar2.speed))
```

출력 결과

```
자동차1의 색상은 빨강이며, 현재 속도는 0km입니다.
자동차2의 색상은 빨강이며, 현재 속도는 0km입니다.
```

- 생성자의 개념 : 인스턴스를 생성하면서 필드값을 초기화시키는 함수
- 생성자의 기본
  - 생성자의 기본 형태 : _ _ init _ _( )라는 이름
  - _ _ init _ _( )는 앞뒤에 언더바(_)가 2개씩, init 는 Initialize 의 약자로 초기화 의미
  - 언더바가 2 개 붙은 것은 파이썬에서 예약해 놓은 것,
  - 프로그램을 작성시 이 이름을 사용해서 새로운 함수나 변수명을 만들지 말 것

```
class 클래스명 :
    def __init__(self) :
        # 이 부분에 초기화할 코드 입력
```

```
class Car :
    color = ""
    speed = 0

    def __init__(self) :
        self.color = "빨강"
        self.speed = 0
```

  - 자동차 세 대의 인스턴스 생성 코드

```
myCar1 = Car()
myCar2 = Car()
myCar3 = Car()
```

메서드의 호출

```
myCar1.upSpeed(30)
myCar2.downSpeed(60)
```

Python 기초: 클래스

```
class Man:
    def __init__(self, name):
        self.name = name
        print("Initialized!")

    def hello(self):
        print("Hello " + self.name + "!")

    def goodbye(self):
        print("Good-bye " + self.name + "!")

m = Man("David")
m.hello()
m.goodbye()
```

이제 터미널에서 man.py를 실행합니다.

self.name: 인스턴스 변수
self: 클라스 자신을 가르킴