



**Hello, World**

# 식별자 (Identifier)

- 변수, 함수, 클래스, 모듈 등을 구별하기 위해 사용하는 이름
- 명명 규칙
  - » 대소문자 구분
  - » 영문자, 숫자, \_(underscores) 사용 가능
  - » 영문자 또는 \_(underscores)로 시작
  - » 예약어를 사용할 수 없음
- 권장 명명 규칙
  - » 클래스 이름은 대문자로 시작하고 다른 식별자는 소문자로 시작
  - » 한 개의 언더스코어(\_)로 시작하는 것은 private 변수
  - » 두 개의 언더스코어(\_\_)로 시작하는 것은 강력한 private 변수 (이름 변형)
  - » 두 개의 언더스코어(\_\_)로 시작하고, 끝나는 것은 언어에서 정의한 특별한 이름

# 예약어

- 파이썬이 특수한 목적으로 사용 → 식별자로 사용할 수 없음

and	exec	not
as	finally	or
assert	for	pass
break	from	print
class	global	raise
continue	if	return
def	import	try
del	in	while
elif	is	with
else	lambda	yield
except		

# 라인과 들여쓰기

- 파이썬은 클래스, 함수, 제어문 등의 코드 블록을 표시하기 위해 중괄호와 같은 특별한 표기를 사용하지 않음
  - » 코드 블록은 들여쓰기에 의해 구분되며 엄격하게 적용됨

## ■ 사례

### 정상 사례

```
if True:
    print "True"
else:
    print "False"
```

### 비정상 사례

```
if True:
    print "Answer"
    print "True"
else:
    print "Answer"
    print "False"
```

# 다중행 실행문

- 파이썬은 한 줄에 하나의 명령문 작성이 원칙이나 예외적으로 여러 줄에 하나의 명령을 작성할 수 있는 방법 제공

» \를 이용한 연결

```
total = item_one + \  
        item_two + \  
        item_three
```

» [], {}, ()와 같은 목록 표시 내부에서는 여러 줄에 나누어서 작성 가능

```
days = ['Monday', 'Tuesday', 'Wednesday',  
        'Thursday', 'Friday']
```

- 한 줄에 2개 이상의 실행문 작성 → ; 사용

```
import sys; x = 'foo'; sys.stdout.write(x + '\n')
```

# 문자열과 주석

- 문자열을 표시하기 위해 ", ' , \"\" , 사용 가능
  - » 이 중 \"\"은 여러 줄로 작성된 문자열 처리 지원

```
word = 'word'
sentence = "This is a sentence."
paragraph = """This is a paragraph. It is
made up of multiple lines and sentences."""
```

- 주석은 인터프리터가 해석하지 않는 영역 표시
  - » #으로 시작되며 그 줄 끝까지 주석

```
name = "Madisetti" # This is again comment
```

```
# This is a comment.
# This is a comment, too.
# This is a comment, too.
# I said that already.
```

# 실행문 블록

- Suite → 여러 개의 개별 실행문이 하나의 실행문 블록을 만든 것
- if, while, def, class 등과 같은 복합 구문은 헤더 라인과 Suite 필요
  - » 헤더 라인은 : (콜론)으로 끝나고 Suite를 구성하는 한 개 이상의 라인이 이어짐

```
if expression :  
    suite  
elif expression :  
    suite  
else :  
    suite
```

The background features a large, flowing green wave that starts from the left, peaks in the upper middle, and then descends towards the bottom right. The wave has a gradient, with lighter green at the top and darker green at the bottom. A solid dark green horizontal bar runs along the very bottom of the slide.

# 변수와 자료형



# 변수와 자료형

- 변수는 데이터를 저장하기 위해 예약된 메모리
- 자료형은 크기, 형식 등 데이터(변수)의 특성에 대한 설명
- 파이썬은 5개의 표준 자료형 제공
  - » Number, String, List, Tuple, Dictionary
- 변수의 자료형에 따라 인터프리터가 메모리를 할당하고 변수에 무엇이 저장될 수 있는지 결정
  - » 정수, 부동소수점, 문자열 등의 데이터를 구분해서 저장
  - » 자료형에 대한 명시적인 표기는 없으며 할당되는 데이터에 의해 판단

# 변수와 객체

- 객체는 프로그램으로 다루고자 하는 모든 것 또는 데이터이고 변수는 그 객체를 가리키는 것 (참조)

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

```
>>> b = 3
```

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

```
True
```

a와 b는 같은 객체를 가리키는 서로 다른 참조

- 파이썬에서는 3, 1.1과 같은 단순 데이터도 객체로 취급

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

```
>>> type(a)
```

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

테스트 환경에 따라 다르게 표시될 수 있음

# 변수

## ■ 변수 선언

- » 변수에 값을 할당할 때 자동으로 변수가 만들어짐
- » 할당할 때 = 연산자 사용

변수명 = 변수에 저장할 값

## ■ 변수 선언 사례

```
counter = 100    # An integer assignment
miles = 1000.0   # A floating point
name = "John"    # A string

print(counter)
print(miles)
print(name)
```



100  
1000.0  
John

# 변수

- 하나의 값을 여러 변수에 할당

```
a = b = c = 1
```

- 여러 개의 값을 여러 변수에 할당

```
a, b, c = 1, 2, "John"
```

```
a, b, c = (1, 2, "John")
```

```
(a, b, c) = 1, 2, "John"
```

```
[a, b, c] = 1, 2, "John"
```

- 변수 제거 (자동으로 제거되기 때문에 명시적으로 삭제할 필요 없음)

```
a = 3  
b = 3  
del a  
del b
```

# 숫자 자료형

## ■ 수치 데이터를 저장하는 자료형

항목	사용 예
정수	123, -345, 0
실수	123.45, -1234.5, 3.4e10
8진수 (숫자0 + 영문자o로 시작)	0o34, 0o25
16진수 (숫자 0 + 영문자 x로 시작)	0x2A, 0xFF

## ■ 수치 데이터 예제

정수형	실수형	8진수	16진수
>>> a = 123 >>> a = -178 >>> a = 0	>>> a = 1.2 >>> a = -3.45 >>> a = 4.24E10 >>> a = 4.24e-10	>>> a = 0o177 >>> a = 0o234 >>> a = 0o456	>>> a = 0x8ff >>> a = 0xABC >>> a = 0x12D

# 숫자 자료형

## ■ 숫자 자료형 데이터를 처리하는 연산자

### » 사칙연산 (+, -, \*, /)

```
>>> a = 3
>>> b = 4
>>> a + b
7
>>> a * b
12
>>> a / b
0.75
```

### » 제곱연산 (\*\*)

```
>>> a = 3
>>> b = 4
>>> a ** b
81
```

### » 나머지 연산 - 나눗셈의 나머지 반환 (%)

```
>>> 7 % 3
1
>>> 3 % 7
3
```

### » 소수점 아래 버림 연산 (//)

```
>>> 7 // 3
2
>>> 3 // 7
0
```

# 숫자 자료형 형변환

- 숫자 자료형으로 형 변환을 처리하는 함수 제공

함수	설명
int( x )	x를 정수로 변환 (정수를 담은 문자열도 가능)
float( x )	x를 실수로 변환 (실수를 담은 문자열도 가능)

- 형변환 예제

```
>>> print( int(11.11) )  
11
```

```
>>> print( float(11) )  
11.0
```

```
>>> print( int("11") )  
11
```

```
>>> print( float("11.11") )  
11.11
```

# 문자 자료형

- 따옴표 내부에 표현된 연속된 문자 집합
- 작은 따옴표 또는 큰 따옴표 사용 가능
  - » 시작 따옴표와 끝 따옴표는 같은 형식의 따옴표 사용
  - » 3개의 따옴표를 연속으로 사용하면 여러 줄로 이루어진 문자열 생성 가능

```
"Hello World"
```

```
'Python is fun'
```

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

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



# 문자 자료형

## ■ 문자열 연산

### » 문자열 더하기 (연결)

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

### » 문자열 곱하기 (반복)

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

```
print("=" * 50)
print("My Program")
print("=" * 50)
```

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

# 문자 자료형

- 문자열 연산 계속

- » 문자열 인덱싱과 슬라이싱

- › [], [:] 연산자를 사용해서 문자열 내의 문자 또는 문자 집합을 추출
- › 순서 번호는 0부터 시작
- › 음수 인덱스는 뒤에서부터 시작하고 1부터 시작

```
>>> a = "Life is too short, You need Python"
>>> a[3]
'e'
>>> a[0]
'L'
>>> a[12]
's'
>>> a[-1]
'n'
```

# 문자 자료형

- 문자열 연산 계속

- » 문자열 인덱싱과 슬라이싱 계속

```
>>> a = "Life is too short, You need Python"
>>> a[0:4]
'Life'
>>> a[0:2]
'Li'
>>> a[5:7]
'is'
>>> a[12:17]
'short'
```

```
>>> a[19:]
'You need Python'
>>> a[:17]
'Life is too short'
>>> a[:]
'Life is too short, You need Python'
>>> a[19:-7]
'You need'
```

# 문자 자료형

## ■ 문자열 포매팅

- » 데이터와 문자열을 조합해서 새 문자열 생성할 때 사용
- » 형식

"문자열과 서식 조합" % 단일 값 또는 튜플

"문자열과 { index } 조합".format(값 목록)

- » 포맷 코드 : 문자열에 데이터를 채우기 위해 자료형에 따라 사용하는 표기법

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

- › %s는 문자열 뿐만 아니라 다른 자료형에도 사용할 수 있음

# 문자 자료형

## ■ 탈출 문자

- » 문자열 내부에 enter, tab, backspace와 같은 특수한 문자를 표기하기 위해 미리 정의한 문자 집합

코드	설명	코드	설명
\n	개행 (줄바꿈)	\r	캐리지 리턴
\t	수평 탭	\"	이중 인용부호(")
\\	문자 "\"	\'	단일 인용부호(')
\b	백 스페이스		

## » 탈출 문자 예제

```
>>> multiline = "Life is too short!!!\n\tYou need \"python\""
>>> print(multiline)
```

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

# 문자 자료형

## ■ 문자열 포매팅 예제

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

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

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

>>> 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.'
```

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

%s는 문자열이 아닌 데이터에도 사용 가능

# 문자 자료형

## ■ 문자열 포매팅 예제

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

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

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

```
>>> 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.'
```

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

# 문자 자료형

## ■ 문자열 포매팅 예제

```
>>> name = '홍길동'
>>> age = 30
>>> f'나의 이름은 {name}입니다. 나이는 {age}입니다.'
'나의 이름은 홍길동입니다. 나이는 30입니다.'

>>> age = 30
>>> f'나는 내년이면 {age+1}살이 됩니다.'
'나는 내년이면 31살이 됩니다.'

>>> d = {'name': '홍길동', 'age': 30}
>>> f'나의 이름은 {d["name"]}입니다. 나이는 {d["age"]}입니다.'
'나의 이름은 홍길동입니다. 나이는 30입니다.'

>>> d = [ '홍길동', 30 ]
>>> f'나의 이름은 {d[0]}입니다. 나이는 {d[1]}입니다.'
'나의 이름은 홍길동입니다. 나이는 30입니다.'
```



# 문자 자료형

## ■ 문자열 관련 함수

### » 문자 개수 세기

```
>>> a = "hobby"  
>>> a.count('b')  
2
```

### » 위치 찾기 ( find vs index )

```
>>> a = "Python is best choice"  
>>> a.find('b')  
10  
>>> a.find('k')  
-1
```

```
>>> a = "Life is too short"  
>>> a.index('t')  
8  
>>> a.index('k')  
Traceback (most recent call last):  
File "<stdin>", line 1, in <module>  
ValueError: substring not found
```

없으면 예외 발생

# 문자 자료형

## ■ 문자열 관련 함수 (계속)

### » 문자열 결합 (join)

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

### » 대소문자 변경

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

### » 좌우 공백 제거

```
>>> a = " hi "  
>>> a.lstrip()  
'hi '  
>>> a = "111hi "  
>>> a.lstrip("1")  
'hi '
```

```
>>> a = " hi "  
>>> a.rstrip()  
' hi'  
>>> a = " hi111"  
>>> a.rstrip("1")  
' hi'
```

```
>>> a = " hi "  
>>> a.strip()  
'hi'  
>>> a = "111hi111"  
>>> a.strip("1")  
'hi'
```

# 문자 자료형

- 문자열 관련 함수 (계속)

- » 문자열 대체 (replace)

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

- » 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']
```

# 리스트 자료형

- 다목적의 복합 데이터 타입
- [] 내부에 ,로 구분되는 항목들로 구성
- 서로 다른 자료형의 데이터를 목록에 포함할 수 있음
- 포함된 멤버들은 []와 [:] 연산자를 사용해서 접근
  - » 0부터 시작하는 인덱스 사용
  - » -(음수) 인덱스는 뒤에서부터 시작하는 위치 값
- + 연산자는 두 리스트를 결합
- \* 연산자는 반복 연산 수행

# 리스트 자료형

## ■ 리스트 만들기

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

## ■ 리스트 인덱싱

```
>>> a = [1, 2, 3]
>>> a
[1, 2, 3]
>>> a[0]
1
>>> a[-1]
3
```

```
>>> a = [1, 2, 3, ['a', 'b', 'c']]
>>> a[0]
1
>>> a[-1]
['a', 'b', 'c']
>>> a[3]
['a', 'b', 'c']
>>> a[-1][1]
'b'
>>> a[-1][2]
'c'
```

# 리스트 자료형

## ■ 리스트 슬라이싱

```
>>> a = [1, 2, 3, 4, 5]
>>> a[0:2]
[1, 2]
>>> a = "12345"
>>> a[0:2]
'12'
```

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

## ■ 리스트 연산자

» + 연산자 (리스트 결합)

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

» \* 연산자 (리스트 반복)

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

# 리스트 자료형

## ■ 리스트 변경

### » 단일 값 변경 (1)

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

### » 다중 값 변경 (2)

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

## ■ 리스트 요소 삭제 (3)

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

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

# 리스트 자료형

## ■ 리스트 관련 함수

### » 리스트에 요소 추가

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

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

### » 리스트 정렬

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

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

### » 리스트 뒤집기

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



# 리스트 자료형

## ■ 리스트 관련 함수

### » 위치 반환

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

### » 리스트에 요소 삽입

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

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

### » 리스트 요소 제거

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

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

# 리스트 자료형

## ■ 리스트 관련 함수

### » 데이터 꺼내기 (읽기 + 삭제)

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

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

### » 리스트에 포함된 요소 개수 세기

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

### » 리스트 확장

```
>>> 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]
```

# 튜플 자료형

- List와 비슷한 데이터 목록 자료형
- ()안에 ,로 구분되는 여러 개의 데이터로 구성됨
  - » 1개의 값을 갖는 경우 끝에 ,를 붙이지 않으면 단일 값 처리
- List는 요소와 크기를 변경할 수 있으나 Tuple은 변경 불가능 (읽기 전용 List)
- Tuple 예제

```
>>> t1 = ()  
>>> t2 = (1,) # ,가 없으면 단일 값  
>>> t3 = (1, 2, 3)  
>>> t4 = 1, 2, 3  
>>> t5 = ('a', 'b', ('ab', 'cd'))
```

# 튜플 자료형

- 튜플 값 삭제 → 오류

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

```
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
TypeError: 'tuple' object doesn't support item deletion
```

- 튜플 값 변경 → 오류

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

```
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
TypeError: 'tuple' object does not support item assignment
```

# 튜플 자료형

## ■ 튜플 인덱싱

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

## ■ 튜플 슬라이싱

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

## ■ 튜플 더하기 (결합)과 튜플 곱하기 (반복)

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

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

# 딕셔너리 자료형

- key - value 세트로 구성되는 일종의 hash table 타입
  - » key로 대부분의 자료형을 사용할 수 있지만 일반적으로 숫자 또는 문자형 사용
  - » value로 모든 자료형의 데이터 사용
- {} 내부에 ,로 구분되는 key : value 세트의 목록으로 구성됨
- 항목에 접근할 때는 [] 사용
- 형식

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

# 딕셔너리 자료형

## ■ 딕셔너리 만들기

```
>>> dic = {'name': 'pey', 'phone': '0119993323', 'birth': '1118'}  
>>> a = {1: 'hi'}  
>>> a = {'a': [1,2,3]}
```

## ■ 딕셔너리 키:값 쌍 추가

```
>>> a = {1: 'a'}  
>>> a[2] = 'b'  
>>> a  
{1: 'a', 2: 'b'}  
>>> a['name'] = 'pey'  
>>> a  
{1: 'a', 2: 'b', 'name': 'pey'}  
>>> a[3] = [1,2,3]  
>>> a  
{1: 'a', 2: 'b', 'name': 'pey', 3: [1, 2, 3]}  
>>> a.update({1: 'modified', 4: 'appended'})  
>>> a  
{1: 'modified', 2: 'b', 'name': 'pey', 3: [1, 2, 3], 4: 'appended'}
```

# 딕셔너리 자료형

- 딕셔너리 값 삭제

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

- 딕셔너리 사용 (키로 값 읽기)

» 튜플과 리스트에 사용했던 인덱싱이나 슬라이싱은 사용할 수 없음

```
>>> grade = {'pey': 10, 'julliet': 99}
>>> grade['pey']
10
>>> grade['julliet']
99
>>> a = {1:'a', 2:'b'}
>>> a[1]
'a'
>>> a[2]
'b'
```



# 딕셔너리 자료형

- 딕셔너리 관련 함수

- » Key 리스트 만들기

- › keys() 함수로 dict\_keys 형식의 값 반환 (2.7 버전에서는 리스트 반환)

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

- » dict\_keys 값 읽기

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

- » dict\_keys를 리스트로 변환

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

# 딕셔너리 자료형

- 딕셔너리 관련 함수

- » Value 리스트 만들기

- › values() 함수로 dict\_values 형식의 값 반환 (2.7 버전에서는 리스트 반환)

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

- » Key:Value 값 쌍 읽기

- › items() 함수로 dict\_items 형식의 값 반환

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

- » Key:Value 쌍 모두 지우기

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

# 딕셔너리 자료형

- 딕셔너리 관련 함수

- » Key로 Value 읽기 → get(key) 함수 사용

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

- » Key로 Value 읽기 → get(key, value) : 키가 없을 경우 기본 값 사용

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

- » 특정 키가 딕셔너리에 있는지 조사 → in

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

# 집합 자료형

- 집합에 관련된 것들을 쉽게 처리하기 위해 제공되는 형식
  - » 중복을 허용하지 않는 집합 데이터 형식
  - » 사용자가 데이터의 순서에 개입할 수 없는 데이터 형식
    - › 인덱싱을 통해 데이터 접근하려면 리스트나 튜플로 변경 후 사용
- set 키워드를 사용해서 생성

```
>>> s1 = set([1,2,3])
>>> s1
{1, 2, 3}
>>> s2 = set("Hello")
>>> s2
{'e', 'l', 'o', 'H'}
```

# 집합 자료형

## ■ 집합 자료형 활용

### » 교집합

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

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

### » 합집합

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

# 집합 자료형

- 집합 자료형 활용

- » 차집합

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

# 집합 자료형

## ■ 집합 자료형 관련 함수

### » 값 1개 추가


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

### » 값 여러 개 추가

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

### » 특정 값 제거

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

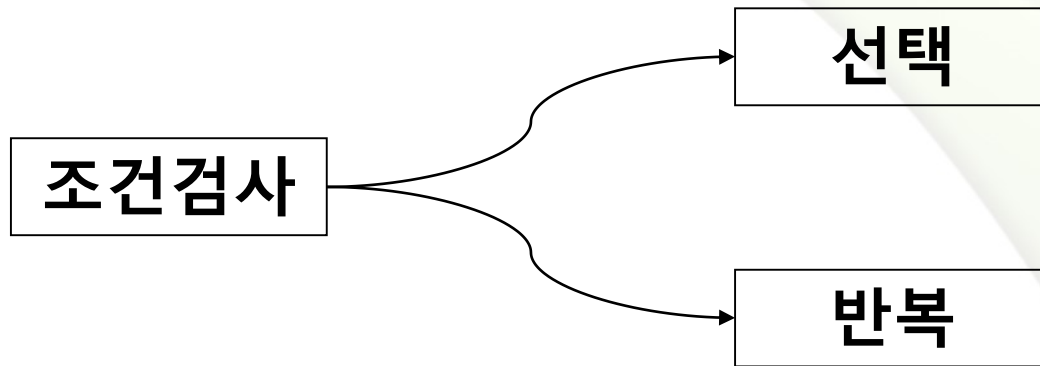


제어문



# 제어문

- 프로그램의 실행 흐름을 논리적으로 제어하는 구문
- 동작 구조



- 지원 문장
  - » 선택문 : if문
  - » 반복문 : while문, for문

# 선택문 (if)

## ▪ 단순 if문

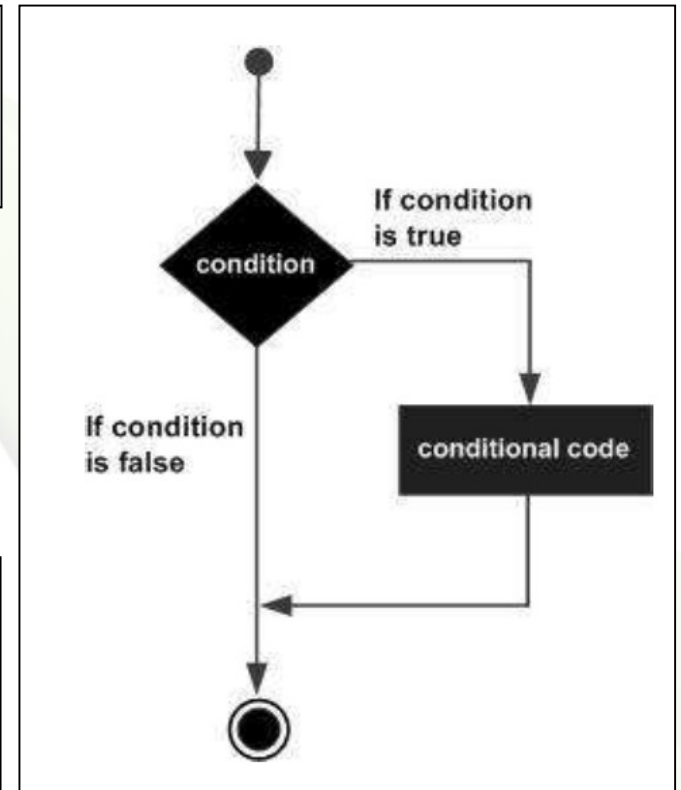
» 조건 검사 결과가 참일때 제어 대상 실행문을 실행하는 선택문

```
if expression:  
    statement(s)
```

- › expression이 참이면 문장 실행
- › expression이 거짓이면 문장 실행 생략

## » 예제

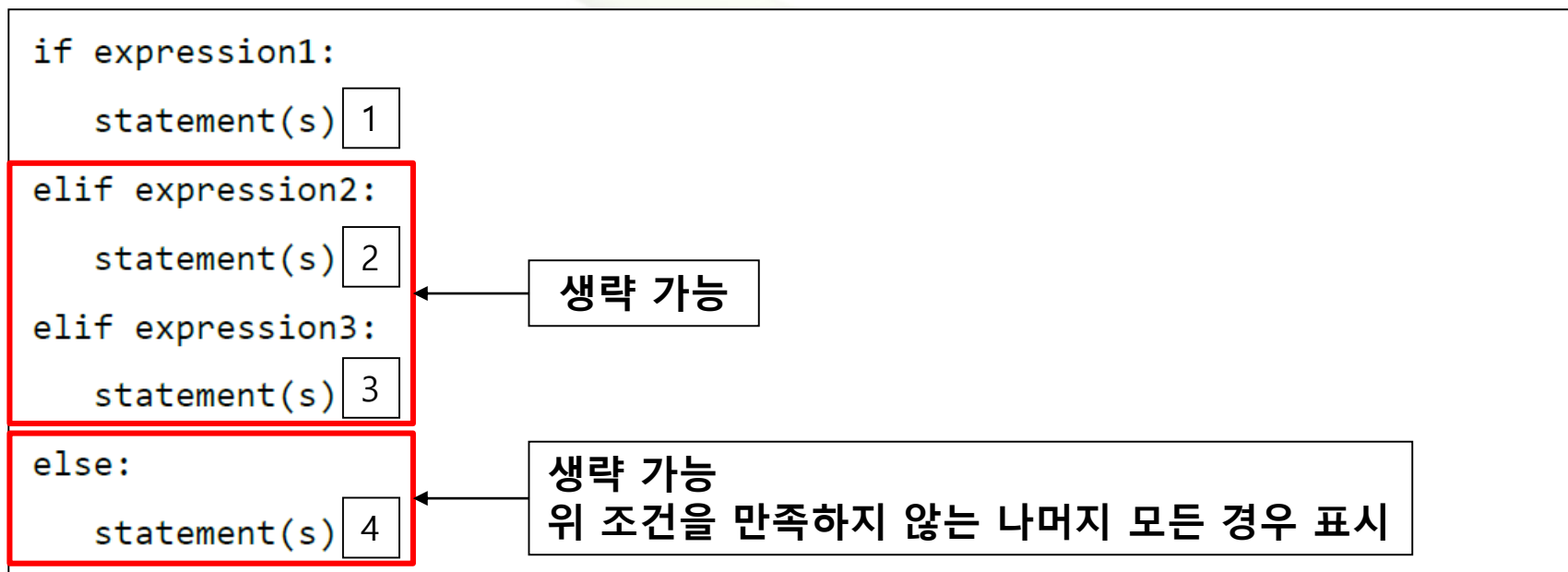
```
var1 = 100  
if var1:  
    print ("1 - Got a true expression value")  
    print (var1)  
var2 = 0  
if var2:  
    print ("2 - Got a true expression value")  
    print (var2)  
    print ("Good bye!")
```



# 선택문 (if)

## ■ 다중 선택 if문

» 여러 실행문 중 조건이 참이 되는 하나의 문장을 실행하는 선택문



- › expression1이 참일 경우 문장 1 실행
- › expression2가 참일 경우 문장 2 실행
- › expression3이 참일 경우 문장 3 실행
- › 나머지 모든 경우 문장 4 실행

# 선택문 (if)

## ■ 다중 선택 if문 예제

```
amount=int(input("Enter amount: "))

if amount<1000:
    discount=amount*0.05
    print ("Discount",discount)
elif amount<5000:
    discount=amount*0.10
    print ("Discount",discount)
else:
    discount=amount*0.15
    print ("Discount",discount)
print ("Net payable:",amount-discount)
```



```
Enter amount: 600
Discount 30.0
Net payable: 570.0

Enter amount: 3000
Discount 300.0
Net payable: 2700.0

Enter amount: 6000
Discount 900.0
Net payable: 5100.0|
```

# 조건식 만들기

- 자료형에 따라 참/거짓을 판단하는 조건식

자료형	참	거짓
숫자	0이 아닌 숫자	0
문자열	"abc"	""
리스트	[1,2,3]	[]
튜플	(1,2,3)	()
딕셔너리	{"a":"b"}	{}

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

# 조건식 만들기

- 관계 연산자 (비교 연산자)를 사용하는 조건식
  - » 두 데이터의 대소를 비교해서 참/거짓 값을 반환하는 연산자
  - » 조건식에 많이 사용되는 연산자

비교연산자	설명
$x < y$	x가 y보다 작으면 참 크거나 같으면 거짓
$x > y$	x가 y보다 크면 참 작거나 같으면 거짓
$x == y$	x와 y가 같으면 참 다르면 거짓
$x != y$	x와 y가 같지 않으면 참 같으면 거짓
$x >= y$	x가 y보다 크거나 같으면 참 작으면 거짓
$x <= y$	x가 y보다 작거나 같으면 참 크면 거짓

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

# 선택문 (if)

- and, or, not 을 사용하는 조건식
  - » 두 논리값을 결합해서 하나의 논리 값을 반환 (and, or)
  - » 대상 논리값의 반대 논리값 반환 (not)

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

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

# 선택문 (if)

- 데이터의 포함 여부로 조건 분기

- » `x in s` / `x not in s`

in	not in
<code>x in 리스트</code>	<code>x not in 리스트</code>
<code>x in 튜플</code>	<code>x not in 튜플</code>
<code>x in 문자열</code>	<code>x not in 문자열</code>

```
>>> 1 in [1, 2, 3]
True
>>> 1 not in [1, 2, 3]
False
>>> pocket = ['paper', 'cellphone', 'money']
>>> if 'money' in pocket:
...     print("택시를 타고 가라")
... else:
...     print("걸어가라")
...
택시를 타고 가라
>>>
```



# 선택문 (if)

- 조건 분기된 실행문이 없는 경우 pass 구문 사용

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

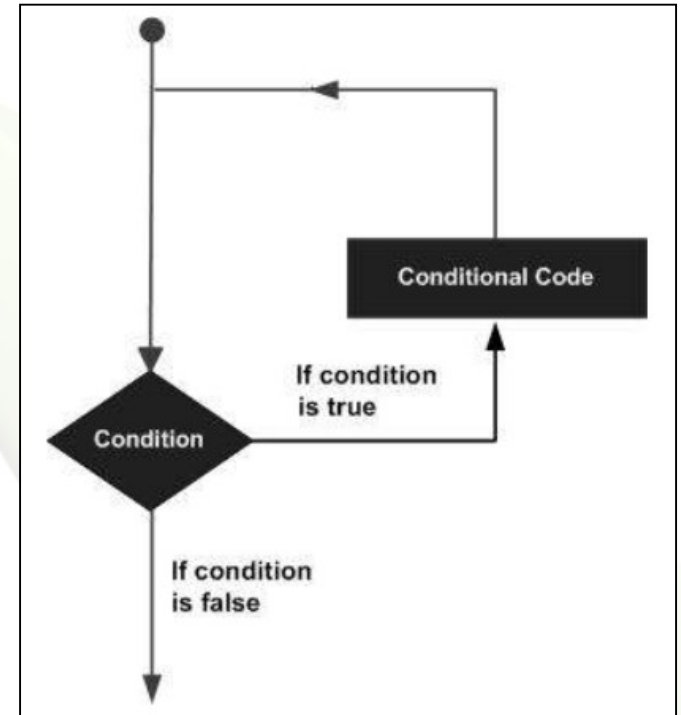
# 반복문 (while)

- 조건에 따라 대상 실행문 집합을 여러 번 반복할 수 있는 구문
- 형식

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

- 예제

```
>>> 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 10번 찍었습니다.  
나무 넘어갑니다.
```



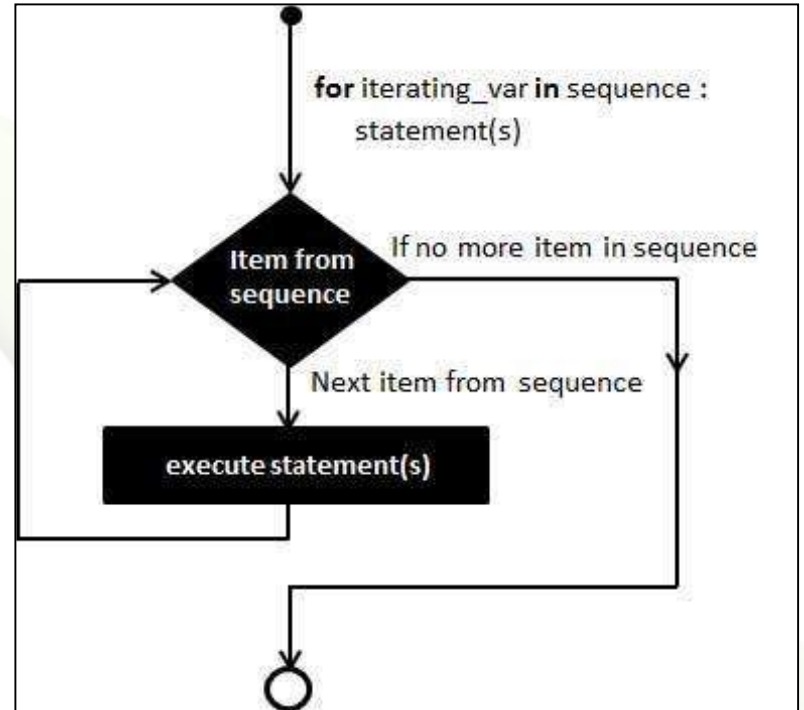
# 반복문 (for)

- 반복의 사이클이 명확한 경우에 적합한 반복 구문

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

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

```
>>> a = [(1,2), (3,4), (5,6)]  
>>> for (first, last) in a:  
...     print(first + last)  
...  
3  
7  
11
```



# range 함수

- range 함수 → 범위를 만드는 함수로 for문과 함께 사용

```
>>> a = range(10)
>>> a
range(0, 10)
>>> a = range(1, 11)
>>> a
range(1, 11)
```

```
>>> sum = 0
>>> for i in range(1, 11):
...     sum = sum + i
...
>>> print(sum)
55
```

# iterator

- 리스트, 튜플 등의 컬렉션 객체에 포함된 모든 요소들을 순차적으로 접근할 수 있는 객체
- 파이썬의 iterator는 `iter()`, `next()` 두 개의 함수 제공
  - » `iter()` 함수는 반복자를 만드는 함수
  - » `next()` 함수는 반복자의 다음 요소를 반환하는 함수 (없으면 `None`)

```
import sys

mylist=[1,2,3,4]
it = iter(mylist)
print (next(it))

it = iter(mylist)
for x in it:
    print (x, end=" ")
```

```
it = iter(mylist)
print()
while True:
    try:
        print (next(it))
    except StopIteration:
        break;
```

# list 내포

- list 내포 → list와 for문을 함께 사용

» 형식

```
[표현식 for 항목 in 반복가능객체 if 조건]
```

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

» 예제

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

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

# else 구문을 사용하는 반복문

- 반복문에 else를 사용하면 반복문이 종료된 후 else의 실행문이 실행됨
- 예제

```
count = 0
while count < 5:
    print (count, " is less than 5")
    count = count + 1
else:
    print (count, " is not less than 5")
```

```
0 is less than 5
1 is less than 5
2 is less than 5
3 is less than 5
4 is less than 5
5 is not less than 5
```

# break 문

- 반복문 탈출 (강제 종료) → break 문 사용

```
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
```



# continue 문

- 다음 반복으로 점프 (반복문의 처음으로 실행 위치 이동) → continue

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

# pass 문

- 구문 규칙상 문장이 필요하지만 실제로는 작성할 코드가 없는 경우 사용
  - » pass문이 있는 곳에서는 어떤 실행도 발생하지 않음
  - » 나중에 작성될 코드에 대한 Placeholder 용도로도 사용

```
for letter in 'Python':  
    if letter == 'h':  
        pass  
        print ('This is pass block')  
    print ('Current Letter :', letter)  
  
print ("Good bye!")
```



```
Current Letter : P  
Current Letter : y  
Current Letter : t  
This is pass block  
Current Letter : h  
Current Letter : o  
Current Letter : n  
Good bye!
```

함수

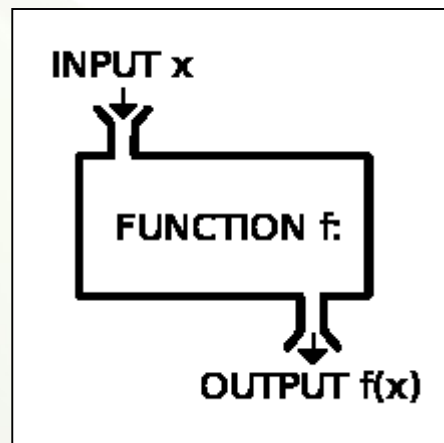
The background features a large, flowing green wave that spans the width of the slide. A thin, light gray horizontal band cuts across the middle of the image. At the bottom, there is a solid dark green horizontal bar. The Korean text '함수' is positioned on the left side, within the white space above the horizontal band.

# 함수?

- 실행문 집합
- 반복적으로 사용되는 코드가 있을 경우 효과적인 재사용을 위해 함수로 구현
- 복잡한 코드를 단위 기능별로 분리하고 효과적으로 관리하기 위한 목적으로도 함수 사용

- 형식

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



- 구현된 함수는 호출을 통해 실행됨

```
변수 = 함수명(입력 인수)
```

# 함수의 입력 값과 출력 값

- 입력 값은 함수에게 전달되는 데이터이고 출력 값은 함수가 호출한 곳으로 반환하는 값
  - » 입력 값을 전달 인자 또는 입력 인수로 부름
  - » 출력은 return문을 통해서 반환

- 입력 값과 출력 값이 모두 있는 함수

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

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

- 입력 값이 없는 함수

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

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

# 함수의 입력 값과 출력 값

## ■ 출력 값이 없는 함수

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

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

## ■ 입력 값과 출력 값이 모두 없는 함수

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

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

# 함수의 입력 값과 출력 값

- 입력 값의 개수를 미리 확정할 수 없는 경우
  - » \*를 사용해서 입력변수 표현

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

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

```
>>> 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
```

# 함수의 결과 값

- 함수의 결과 값은 언제나 한 개
  - » 두 개 이상의 값을 반환할 경우 반환 값을 목록으로 하는 하나의 튜플을 만들어서 반환

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

- » 함수를 호출하고 튜플로 반환 값 저장

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

- » 함수를 호출하고 각각 개별 변수에 튜플 값을 나누어서 저장

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



# 입력 인수에 기본 값 사용

- 입력 인수 = 기본 값 형식으로 입력 인수에 기본 값 설정
  - » 입력 인수에 기본 값이 지정된 경우 호출할 때
    - › 값을 전달하지 않으면 기본 값이 사용되고
    - › 값을 전달하면 전달한 값이 사용됨

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

```
say_myself("장동건", 40)  
say_myself("장동건", 40, True)
```

```
나의 이름은 장동건입니다.  
나이는 27살입니다.  
남자입니다.
```

# 입력 인수에 기본 값 사용

- 기본 값을 지정한 입력 인수는 모든 입력 인수 중 마지막에만 사용할 수 있음
  - » 일반 입력 인수 중간에 기본 값을 지정한 입력 인수를 사용할 수 없음

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

실행할 경우 오류 발생

# 함수 내부에 선언된 변수의 사용 범위

- 함수 내부에서 선언된 변수는 함수 안에서만 사용할 수 있음 → 지역 변수

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

vartest(a)
print(a)
```

위 코드의 실행 결과는 1 (함수 안의 a와 함수 밖의 a는 별개의 변수)

- 함수 내부에서 함수 외부의 변수를 변경하는 방법 → global 변수 사용  
» 이 방법은 일반적으로 권장되지 않음

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

위 코드의 실행 결과는 2 (함수 안의 a와 함수 밖의 a는 같은 변수)

The background features a large, flowing green wave that starts from the left, peaks in the upper middle, and then descends towards the bottom right. The wave has a gradient, with lighter green at the top and darker green at the bottom. A solid dark green horizontal bar is positioned at the very bottom of the image.

# **사용자 입력과 출력**

# 사용자 입력

- 다양한 방식으로 외부의 데이터가 프로그램으로 유입되며 그 중 하나는 사용자의 입력을 통해서 이루어짐
  - » 파이썬은 명령행 환경에서 사용자의 입력을 처리하는 input 함수 제공 (파이썬 2.7에서는 raw\_input 함수 사용)

```
>>> a = input()
Life is too short, you need python
>>> a
'Life is too short, you need python'
>>>
```

```
>>> number = input("숫자를 입력하세요: ")
숫자를 입력하세요: 3
>>> print(number)
3
>>>
```

# 화면 출력

- 파이썬은 명령행 환경에서 메시지를 출력하기 위해 `print` 함수 제공
  - » `print` 함수 사용 사례

```
>>> a = 123
>>> print(a)
123
>>> a = "Python"
>>> print(a)
Python
>>> a = [1, 2, 3]
>>> print(a)
[1, 2, 3]
```

- » 따옴표로 구분된 문자열을 여러 개 사용하면 `+` 연산을 자동으로 수행

```
>>> print("life" "is" "too short")
lifeistoo short
>>> print("life"+"is"+"too short")
lifeistoo short
```


# 화면 출력

- 파이썬은 명령행 환경에서 메시지를 출력하기 위해 `print` 함수 제공
  - » ,로 구분된 여러 개의 문자열은 공백을 삽입해서 문자열 결합

```
>>> print("life", "is", "too short")
life is too short
```

- » 한 줄에 여러 개의 문자열을 출력하려면 `end` 전달인자 사용
  - » 파이썬 2.7에서는 `print(i, )` 형식 사용

```
>>> for i in range(10):
...     print(i, end=' ')
...
0 1 2 3 4 5 6 7 8 9
```

The background features a large, flowing green wave that starts from the left, peaks in the upper middle, and then descends towards the bottom right. The wave has a gradient, with lighter green at the top and darker green at the bottom. A solid dark green horizontal bar runs across the very bottom of the image.

# 객체 지향 프로그래밍

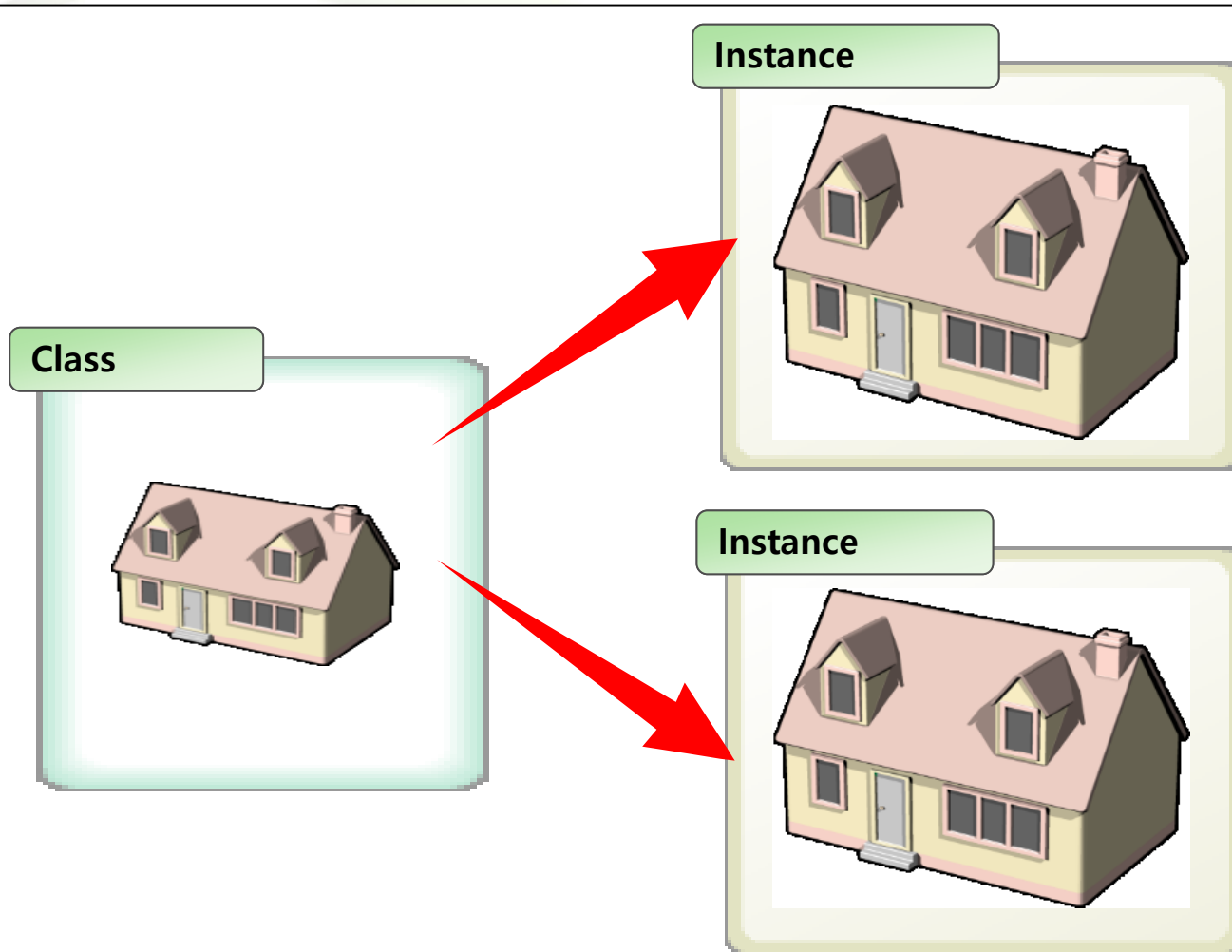


# 객체 지향 프로그래밍

- 추상화 (Abstraction)
  - » 대상세계의 처리 대상(객체)을 프로그래밍 영역의 표현단위인 클래스 등으로 변환하는 과정
  - » 중요한 것과 중요하지 않은 것을 구분하고 선택적으로 재구성
- 객체, 클래스, 인스턴스
  - » 객체는 프로그램으로 다루고자 하는 모든 대상
  - » 클래스는 제어 대상을 프로그래밍 수준에서 정의한 사용자 정의 자료형
  - » 인스턴스는 클래스를 기반으로 메모리상에 생성된 데이터
  - » 하나의 클래스를 기반으로 여러 인스턴스를 생성하고 각 인스턴스는 서로 구분되는 독립적인 단위
- 파이썬은 객체 지향 프로그래밍을 지원
  - » 쉽게 클래스를 만들고 사용할 수 있음

# 객체 지향 프로그래밍

- 객체, 클래스, 인스턴스



# 객체 지향 프로그래밍

## ■ 용어 정리

용어	설명
클래스	객체를 만들기 위한 사용자 정의 자료형
클래스 변수	모든 인스턴스에 의해 공유되는 멤버
데이터 멤버	객체의 특성을 반영하는 클래스 내부에 선언된 변수
함수 오버로딩	같은 함수에 여러 기능을 부여하는 구현 기법
인스턴스 변수	메서드 내부에 선언되고 현재 인스턴스를 통해서만 사용할 수 있는 변수
상속	어떤 클래스의 특성을 다른 클래스에 전달하는 기법
인스턴스	클래스를 기반으로 만들어진 구체적인 객체
메서드	객체의 기능을 반영하는 클래스 내부에 선언된 함수
객체	클래스에 의해 정의된 데이터 구조의 인스턴스
연산자 재정의	연산자에 하나 이상의 기능을 부여하는 구현

# 클래스 만들기

- 클래스를 정의하기 위해 class 구문 사용
- 형식

```
class ClassName:  
    'Optional class documentation string'  
    class_suite
```

» class\_suite는 멤버 변수, 메서드 등 클래스 내부에 포함되는 모든 요소

# 클래스 만들기

## ■ 클래스 만들기 예제

```
class Employee:
    'Common base class for all employees'
    empCount = 0

    def __init__(self, name, salary):
        self.name = name
        self.salary = salary
        Employee.empCount += 1

    def displayCount(self):
        print("Total Employee %d" % Employee.empCount)

    def displayEmployee(self):
        print("Name : ", self.name, " , Salary: ", self.salary)
```

- » empCount는 모든 인스턴스에 의해 공유되는 클래스 멤버
- » \_\_init\_\_ 함수는 생성자 또는 초기화 함수로 불리는 특별한 함수 (인스턴스 생성시 자동 호출)
- » 메서드의 첫 번째 전달인자인 self는 객체 자신을 참조하는 특별한 변수

# 클래스 사용

- 인스턴스 만들기

- » 정의된 클래스를 기반으로 메모리에 공간을 할당

```
# This would create first object of Employee class
emp1 = Employee("Zara", 2000)
# This would create second object of Employee class
emp2 = Employee("Manni", 5000)
```

- 멤버 접근

- » 객체의 멤버는 .(dot) 연산자를 사용해서 접근

```
emp1.displayEmployee()
emp2.displayEmployee()
print ("Total Employee %d" % Employee.empCount)
```

```
Name :  Zara ,Salary:  2000
Name :  Manni ,Salary:  5000
Total Employee 2
```

# 클래스 사용

- 객체 변경

- » 객체 생성 후에도 멤버 추가 및 삭제 가능

```
emp1.salary = 7000 # Add an 'salary' attribute.  
emp1.name = 'xyz' # Modify 'age' attribute.  
del emp1.salary # Delete 'age' attribute.
```

# 상속

- 이미 만들어진 클래스의 내용을 재사용해서 새로운 클래스를 만드는 기법
  - » 클래스 수준의 재사용 원리
- 상속 받은 클래스는 부모 클래스의 멤버를 자동으로 포함하게 됨
  - » 부모 클래스의 멤버를 재정의 할 수 있음
- 형식

```
class SubClassName (ParentClass1[, ParentClass2, ...]):  
    'Optional class documentation string'  
    class_suite
```



# 상속

## ■ 상속 예제

```
class Parent:          # define parent class
    parentAttr = 100
    def __init__(self):
        print ("Calling parent constructor")

    def parentMethod(self):
        print ('Calling parent method')

    def setAttr(self, attr):
        Parent.parentAttr = attr

    def getAttr(self):
        print ("Parent attribute :", Parent.parentAttr)

class Child(Parent): # define child class
    def __init__(self):
        print ("Calling child constructor")

    def childMethod(self):
        print ('Calling child method')
```

# 상속

## ■ 상속 예제 (계속)

```
c = Child()           # instance of child
c.childMethod()       # child calls its method
c.parentMethod()      # calls parent's method
c.setAttr(200)        # again call parent's method
c.getAttr()           # again call parent's method
```

```
Calling child constructor
Calling child method
Calling parent method
Parent attribute : 200
```

모음

# 모듈 정의와 사용

- 다른 파이썬 프로그램에서 불러와 사용할 수 있도록 함수, 변수 또는 클래스들을 모아서 만들어진 파이썬 파일

- 모듈 정의

- » mod1.py 파일을 만들고 아래의 코드를 작성 후 저장

```
# mod1.py

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

- 모듈 사용

- » mod1.py 파일이 저장된 장소에서 파이썬 대화형 프로그램 실행
  - » import 구문으로 모듈 로딩

```
import 모듈이름
```

```
>>> import mod1
>>> print(mod1.sum(3,4))
7
```

# 모듈 정의와 사용

## ■ 모듈 사용 (계속)

### » 모듈 이름 없이 함수 사용 가능

```
from 모듈이름 import 함수
```

```
>>> from mod1 import sum  
>>> sum(3, 4)  
7
```

### » 여러 개의 함수를 동시에 import

```
from 모듈이름 import 함수1, 함수2, ...
```

```
from mod1 import sum, safe_sum
```

또는

```
from 모듈이름 import *
```

```
from mod1 import *
```

# 선택적 import

- 모듈을 import 할 때 선택적 실행 관리 가능
  - » `__name__`은 파이썬 파일을 직접 실행했을 때 사용되는 변수
    - › `python abc.py`로 실행한 경우 `__name__`에는 `"__main__"`이 저장됨
- 아래 구문은 파일을 직접 실행한 경우에만 실행되도록 조건 처리
  - » 임포트 한 경우에는 실행되지 않음

```
if __name__ == "__main__":  
    print(safe_sum('a', 1))  
    print(safe_sum(1, 4))  
    print(sum(10, 10.4))
```

```
>>> import mod1  
>>>
```

# 클래스나 변수를 포함한 모듈

- 클래스 또는 변수를 포함한 모듈 생성

```
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))
```

```
C:\Python>python mod2.py
3.141592
12.566368
7.541592
```

직접 실행한 경우 실행 결과 출력

```
C:\Python>python
>>> import mod2
>>>
```

import 한 경우 실행되지 않음

# 클래스나 변수를 포함한 모듈

- 모듈에 포함된 변수, 클래스, 함수 사용하기

```
>>> print(mod2.PI)  
3.141592
```

```
>>> a = mod2.Math()  
>>> print(a.solve(2))  
12.566368
```

```
>>> print(mod2.sum(mod2.PI, 4.4))  
7.541592
```





패키지

# 패키지

- 패키지는 파이썬 모듈을 계층적으로 관리하는 도구
  - » 디렉터리와 파이썬 모듈로 구성됨
  - » 각 디렉터리와 모듈은 .(dot)를 사용해서 연결
- 패키지 예제

```
game/  
  __init__.py  
  sound/  
    __init__.py  
    echo.py  
    wav.py  
  graphic/  
    __init__.py  
    screen.py  
    render.py  
  play/  
    __init__.py  
    run.py  
    test.py
```

# 패키지 사용

- 디렉터리와 모듈을 결합한 형태로 임포트해서 사용

```
>>> import game.sound.echo
>>> game.sound.echo.echo_test()
echo
```

- 디렉터리와 모듈을 분리해서 임포트 구문 작성

```
>>> from game.sound import echo
>>> echo.echo_test()
echo
```

- 메서드를 임포트해서 사용

```
>>> from game.sound.echo import echo_test
>>> echo_test()
echo
```

# \_\_init\_\_.py 파일

- 특정 디렉터리가 패키지의 일부임을 표시하는 파일
  - » 폴더에 \_\_init\_\_.py 파일이 없는 경우 패키지로 인식되지 않음 (파이썬 3.3 버전부터 없어도 패키지로 인식됨)
  - » \_\_init\_\_.py 파일이 없을 경우 import 할 때 아래와 같은 오류 발생


```
>>> import game.sound.echo
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ImportError: No module named sound.echo
```

- 모듈을 와일드카드(\*) 형태로 임포트 할 때 \_\_init\_\_.py 파일에 \_\_all\_\_이라는 변수를 선언하고 임포트 할 수 있는 모듈을 정의해야 함

```
# C:/Python/game/sound/__init__.py
__all__ = ['echo']
```

- » 단, 위 규칙은 모듈을 와일드카드로 임포트 할 경우에만 해당되며 메서드에 대한 와일드카드 사용은 항상 가능

# 예외 처리

The background of the slide features abstract, flowing green shapes. A thick, dark green band runs horizontally across the bottom. Above it, lighter green, wavy, ribbon-like shapes curve across the frame, creating a sense of movement and depth. The overall aesthetic is clean and modern.

# 예외 처리

- 프로그램 실행 중 다양한 형태의 오류 발생

```
>>> f = open("나없는파일", 'r')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
FileNotFoundError: [Errno 2] No such file or directory: '나없는파일'
```

```
>>> 4 / 0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
```

```
>>> a = [1,2,3]
>>> a[4]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

- 예외 처리는 프로그램 실행 중 발생하는 오류를 적절하게 관리하는 기법
- 파이썬은 try, except 를 통해서 오류 처리 구문 지원

# 예외 처리

## ■ 오류 처리 구문 형식

» try, except 만 사용

```
try:  
    ...  
except:  
    ...
```

» 발생 오류만 포함한 except

```
try:  
    ...  
except 발생 오류:  
    ...
```

» 발생 오류와 오류 메시지 변수까지 포함한 except

```
try:  
    ...  
except 발생 오류 as 오류 메시지 변수:  
    ...
```

# 예외 처리

## ■ 오류 처리 구문 형식 (계속)

» try ... else (예외가 발생하지 않을 경우 실행할 구문)

```
try:
    f = open('foo.txt', 'r')
except FileNotFoundError as e:
    print(str(e))
else:
    data = f.read()
    f.close()
```

» try ... finally (예외 발생 여부와 상관 없이 실행되는 코드 구성)

```
f = open('foo.txt', 'w')
try:
    # 무언가를 수행한다.
finally:
    f.close()
```



# 예외 처리

- 오류 처리 구문 형식

- » 다중 예외 처리 구문

```
try:
    a = [1,2]
    print(a[3])
    4/0
except ZeroDivisionError as e:
    print(e)
except IndexError as e:
    print(e)
```

- » 오류 회피

- › 발생한 오류에 대해 특별한 처리 없이 정상 흐름으로 돌리는 방법

```
try:
    f = open("i_am_not_exist.txt", 'r')
except FileNotFoundError:
    pass
```

# 강제 예외 발생

- 파이썬은 `raise` 명령을 사용해서 예외를 상위 예외처리 영역으로 전달

```
class Bird:
    def fly(self):
        raise NotImplementedError
```

```
class Eagle(Bird):
    pass
```

```
eagle = Eagle()
eagle.fly()
```

```
Traceback (most recent call last):
  File "...", line 33, in <module>
    eagle.fly()
  File "...", line 26, in fly
    raise NotImplementedError
NotImplementedError
```