파이썬 기초
프로그래밍

파이썬 시작

리터럴 (상수) : **5, "python" , 3.14** 와 같은 고정된 데이터 , 변수와 대비

표현식 : **a+b** 값이 **5**라면 값을 의미하는 표현, 값을 반환하는 표현이 됨에 때라 표현식이라고 함
표현식은 결국 어떤 값을 의미, 변수에 할당 가능

구문 : 값의 의미를 지니지 않으며 어떤 목적을 수행하는 코드를 의미

식별자 : 변수, 함수, 클래스 등을 식별하기 위한 이름

키워드 : **if, for, True** 등 특정 목적으로 사용되는 명령어

식별자 규칙 : 일반적인 **C** 프로그래밍 변수명 규칙과 유사하며 보통 첫문자로 숫자가 올수 없고
대소문자 구별, 특수문자 포함 안됨, 키워드 사용불가 등으로 규칙이 정해짐

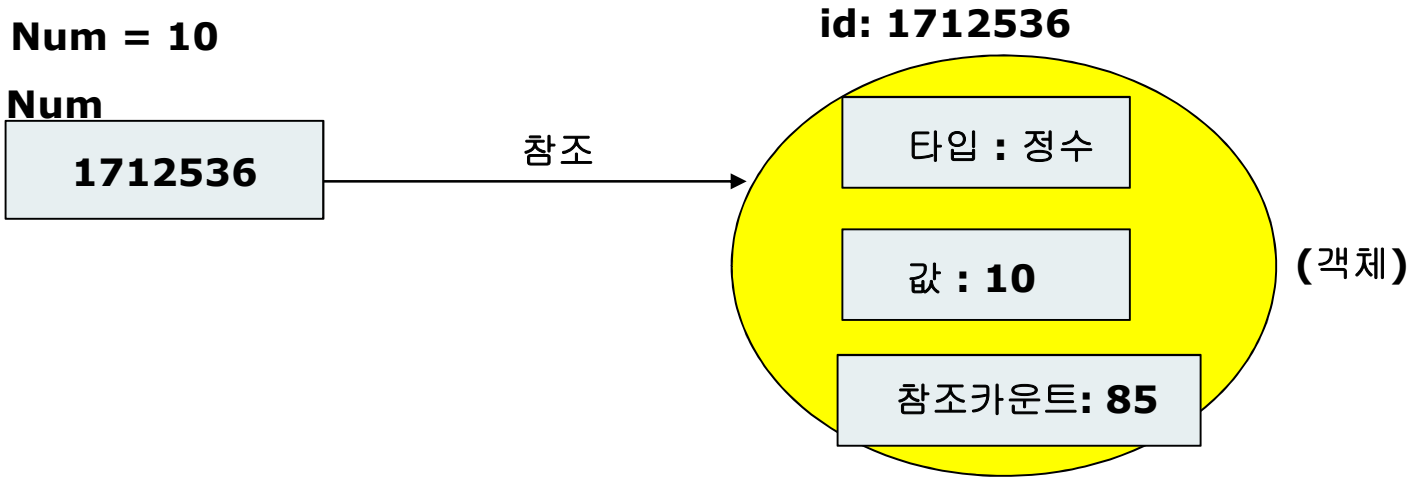
주석 : **# (**한 라인 주석**) , """** ~~~ **"""** (여러 라인 주석)

들여쓰기 : **C** 프로그래밍의 **{ }(**중괄호**)** 처럼 코드의 블록을 의미
파이썬 프로그래밍 할때 들여쓰기 중요함

파이썬 변수

- 동적 타이핑 언어로서 파이썬은 실행시간에 변수의 데이터 타입이 결정
  따라서 타 언어 와 달리 변수 선언이 필요 없음

- 파이썬 변수 : 객체 **id**를 저장하는 변수 이며 **id**를 이용해 객체를 참조하는 개념



파이썬이 제공하는 특수한 객체의 내부에 저장되면서 데이터의 타입이 검사된후
타입에 대한 정보도 같이 저장

```
a = 3.14
print(type(a))
a = 8
print(type(a))
a = "python"
print(type(a))
```

```
<class 'float'>
<class 'int'>
<class 'str'>
```

파이썬 객체 메소드


```
string_object = "python programming"

total = [ ]  // 리스트
for x in string_object:
        if x == " ":
                total.append(" ")
        else:
                total.append(chr(ord(x)-32))

string_big = "".join(total)

print("for statement used :")
print(string_big)

print("====================")

print("upper method used :")
string_up = string_object.upper()
print(string_up)
```

리스트 객체 메소드

객체의 내장 메소드 활용 시 손쉽게 코드 구현 가능

문자열 객체 메소드

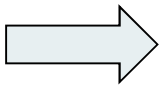
결과

```
for statement used :
PYTHON PROGRAMMING
====================
upper method used :
PYTHON PROGRAMMING
```

파이썬 내장 데이터 type

분류	내장 타입	타입명	상수
숫자	정수	int	520
	실수	Float	3.14
	복소수	Complex	5+2j
	참, 거짓	bool	True

```
a = 520
print(type(a))
a = 3.14
print(type(a))
a = 5+2j
print(type(a))
a = True
print(type(a))
```

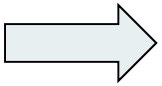


```
<class 'int'>
<class 'float'>
<class 'complex'>
<class 'bool'>
```

파이썬 내장 데이터 type

분류	내장 타입	타입명	상수
시퀀스	문자열	str	“programming”
	리스트	list	[1,2,3,4]
	튜플	tuple	(5,6,7)
셋	셋	Set	{2,4,6,8}
매핑	딕셔너리	dict	{“a”:1, “b”:2, “c”:3}

```
a = "programming"
print(type(a))
a= [1,2,3,4]
print(type(a))
a = (5,6,7)
print(type(a))
a = {2,4,6,8}
print(type(a))
a = {"a":1, "b":2, "c":3 }
print(type(a))
```



```
<class 'str'>
<class 'list'>
<class 'tuple'>
<class 'set'>
<class 'dict'>
```

기본 숫자 연산

print(100/9) ➔ 나눗셈 연산하되 결과는 실수 타입
print(100//9) ➔ 나눗셈 연산하되 소수점 이하는 버리고 결과는 정수 타입
Print(100%9) ➔ 나머지 값 반환
print(divmod(100,9)) ➔ 몫 과 나머지를 튜플 형태로 반환

```
a = 1
b = 2.0
c = a + b
print(c)
```
➔ 서로 다른 수치 자료형 연산시 상위 자료형을 따름

```
print(2**3)
```
➔ 2의 3승

```
x = 77.0
x //= 7
print(x)
```
➔ x = x // 7 과 동일 연산

파이썬 2.x	정수와 정수 나눗셈 결과는 무조건 정수
파이썬 3.x	정수와 정수 나눗셈 결과는 항상 실수 타입

비교 연산

비교(관계) 연산자

연산 기호	결과값
A > B	A항이 B항보다 크면 1(TRUE), 그렇지 않을 경우 0(FALSE)
A >= B	A항이 B항보다 크거나 같으면 1(TRUE), 그렇지 않을 경우 0(FALSE)
A < B	A항이 B항보다 작으면 1(TRUE), 그렇지 않을 경우 0(FALSE)
A <= B	A항이 B항보다 작거나 같으면 1(TRUE), 그렇지 않을 경우 0(FALSE)
A == B	A항이 B항과 같으면 1(TRUE), 그렇지 않을 경우 0(FALSE)
A != B	A항과 B항이 다르면 1(TRUE), 그렇지 않을 경우 0(FALSE)

비트 연산

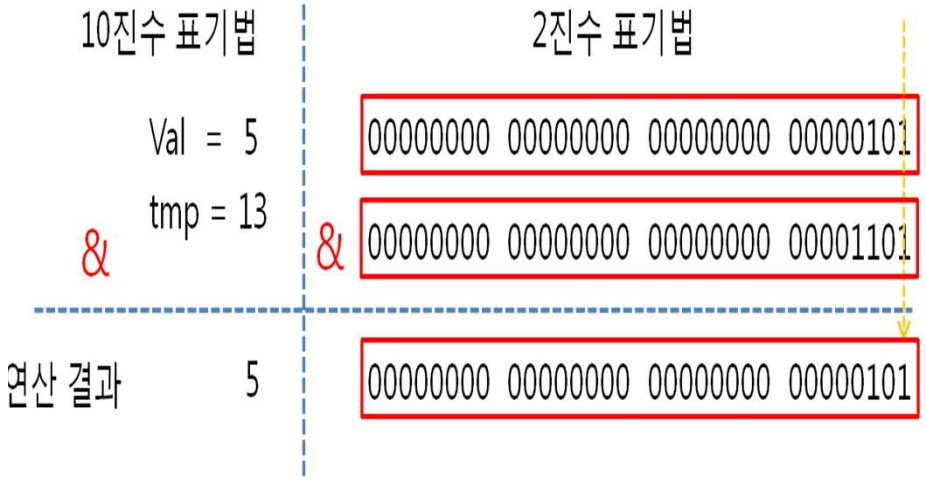
비트 연산자

연산 기호	기능
&	비트 단위 논리AND(곱) 연산자
|	비트 단위 논리OR(합) 연산자
~	비트 단위 논리NOT(부정) 연산자
^	비트 단위 배타적 논리OR(XOR) 연산자
<<	비트 왼쪽 이동(Shift) 연산자
>>	비트 오른쪽 이동(Shift) 연산자

비트 연산

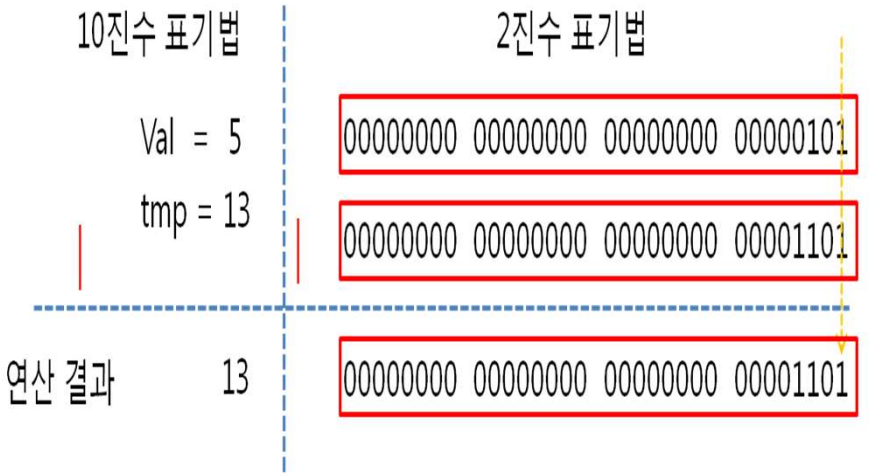
HANCOM 한컴MDS 아카데미

& 연산자 (비트 and)



비트 & (AND) 연산은 위 그림의 화살표 방향으로 각각의 두 비트를 위치별로 논리AND 연산 수행

| 연산자 (비트 or)

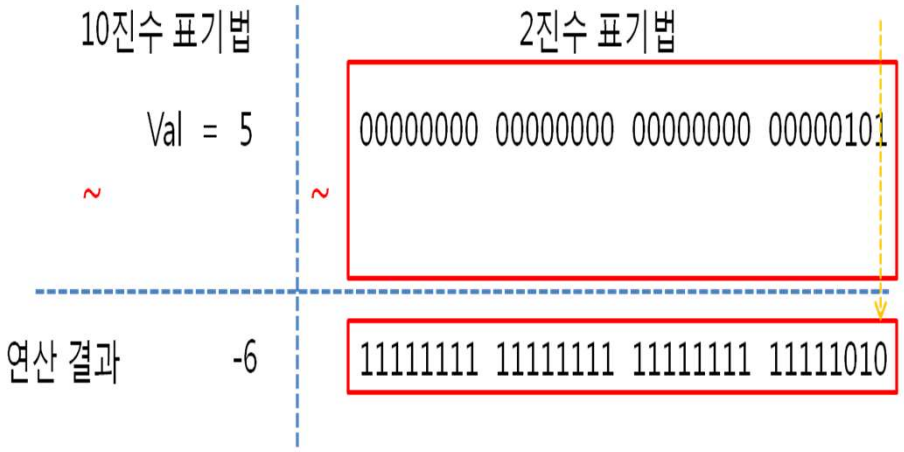


비트 | (OR) 연산은 위 그림의 화살표 방향으로 각각의 두 비트를 위치별로 논리OR 연산 수행

비트 연산

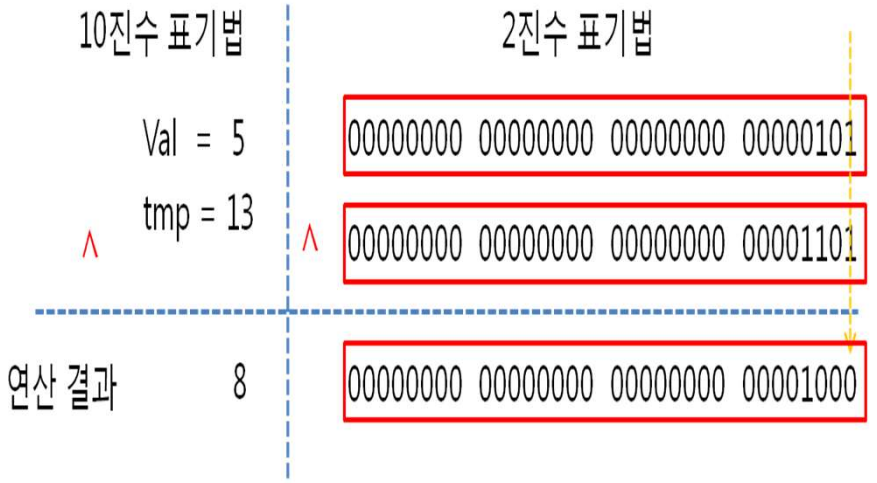
HANCOM 한컴MDS 아카데미

~ 연산자 (비트 not)



비트 ~ (NOT) 연산은 위 그림의 화살표 방향으로 각각의 비트를 반전 시키는 논리NOT 연산 수행

^ 연산자 (비트 xor)



비트 ^ (XOR) 연산은 위 그림의 화살표 방향으로 각각의 두 비트를 위치별로 배타적 논리OR(exclusive or) 연산 수행

논리 연산


```
a = 10 and 20
print(a)
a = 20 and 10
print(a)
a = 0 and 10
print(a)
a = 0 and 111
print(a)
```

- and 연산 : 좌항이 True 이면 우항 값을 취함
  좌항이 False 이면 무조건 False(0)

```
a = 10 or 0
print(a)
a = 20 or 10
print(a)
a = 0 or 10
print(a)
a = 0 or 111
print(a)
```

- or 연산 : 좌항이 True 이면 좌항 값을 취함
  좌항이 False 이면 우항 값을 취함

```
for x in range(1, 10):
        res = x > 5 and 10 or 20
        print(x, res)
```

```
x : 1, res : 20
x : 2, res : 20
x : 3, res : 20
x : 4, res : 20
x : 5, res : 20
x : 6, res : 10
x : 7, res : 10
x : 8, res : 10
x : 9, res : 10
```

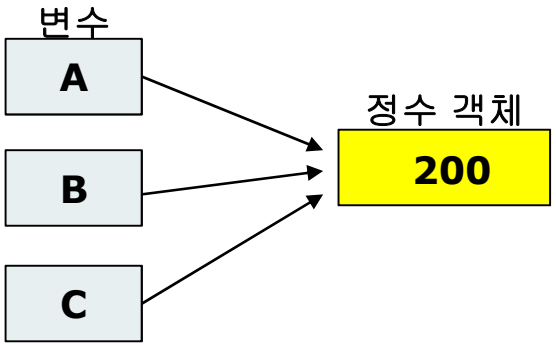
- 연산자 우선순위

  기본 연산 > 비트연산 > 비교연산 > 논리 연산 > 기타연산

- ()괄호 연산이 우선순위가 가장 높기 때문에 연산순위 변경 시 활용

정수 타입 id

- 수명이 겹치지 않는 정수 객체는 다른 id 값을 갖음



-5 ~ 256 까지 정수 객체를 미리 생성해 놓고 변수에 대입할 때 객체 **id**를 전달해 참조토록 한다

```
def  func1():
        c = 257
        print(id(c))

d = 257
print(id(d))
func1()
```

-5 ~ 256 범위를 벗어난 정수 객체는 새로 생성되기 때문에 **id** 값은 달라짐

시퀀스 타입 – 문자열

HANCOM 한컴MDS 아카데미

```
str1 = 'python'
print(str1)
str2= "I'm python"
print(str2)
str3 = '"Good python"'
print(str3)
str4 = """python
is
good programm"""
print(str4)
```

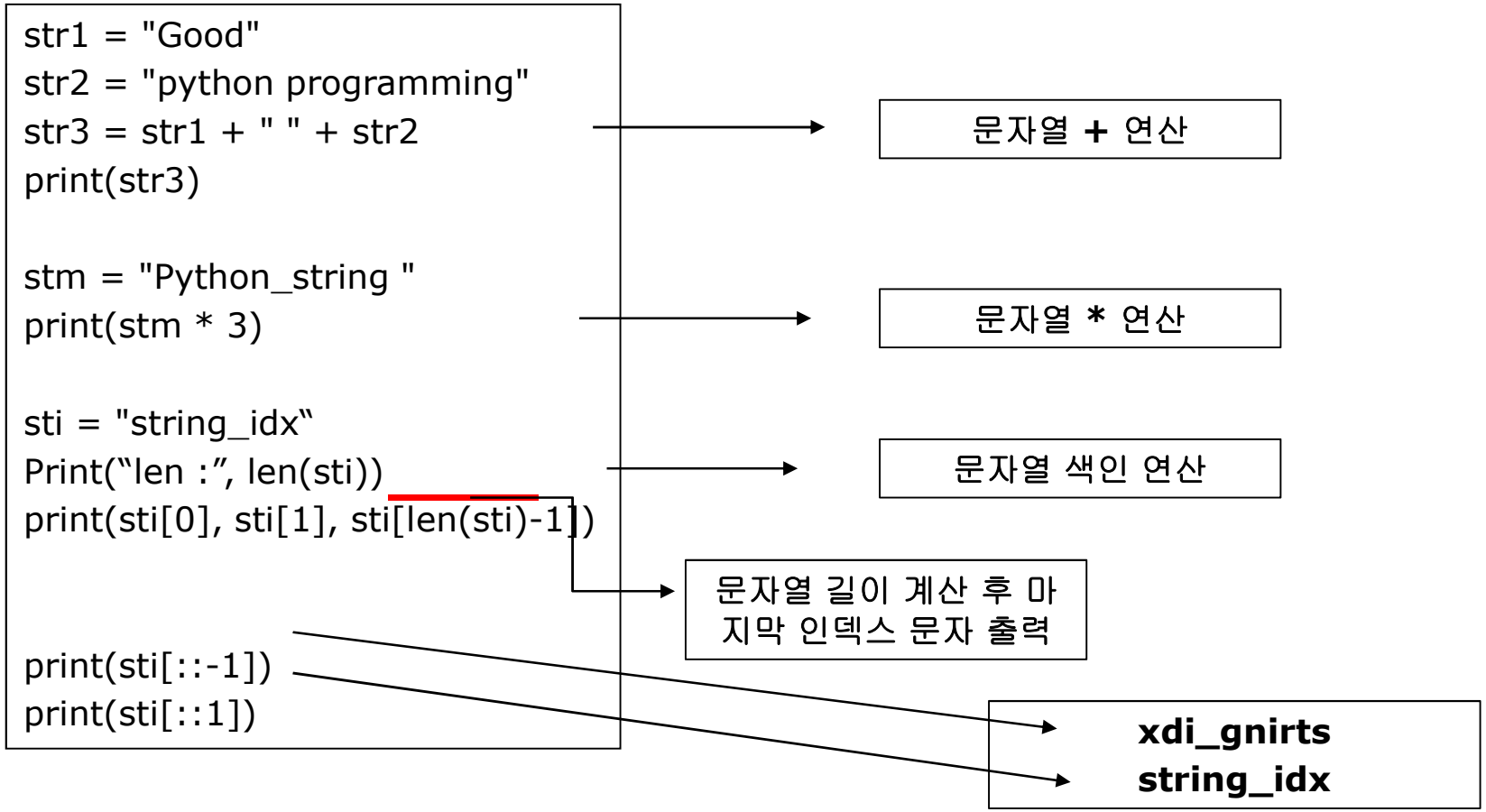
```
python
I'm python
"Good python"
python
is
good programm
```

- 문자열 안에 큰 따옴표가 들어갈 경우 작은 따옴표로 문자열 생성
- 작은 따옴표가 있는 문자열은 큰 따옴표로 문자열 생성

이스케이프 시퀀스

기호	설명
\n	문자열에서 행(줄) 바꿈
\r	커서의 위치를 맨 앞으로 이동
\t	수평 탭 이동

문자열 – 기본 연산



- 시퀀스 타입 인덱스**(index)** 색인은 항상 **0** 부터 시작

문자열 – 기본 연산

HANCOM 한컴MDS 아카데미

in 연산

```
mystr = "Hello python programming"

print('pro' in mystr)

count = 0
for s in mystr:
        if s == 'o':
                count += 1
print(" o char count : %d" %count)
```

- 시퀀스 타입에 적용 가능한 **in** 연산
- 어떤 문자 또는 문자열이 있는지 확인 가능
- **bool** 타입 결과를 반환

문자열 비교 연산

```
print('a' < 'b')
print("python" < "programming")
print("py" < "python")
```

True
False
True

- 문자의 크기는 공백문자 **<** 대문자 **<** 소문자 순서
- 알파벳**(**사전**)** 순서로 크기가 증가

문자열-포맷팅

- 서식 지정자 이용한 문자열 포맷팅

```
print("%s is %d age" %("Hong" , 50))
```

- **%d, %x, %c, %s** 등 다양한 서식 지정자가 있음

- **format()**함수를 이용한 문자열 포맷팅

```
mystr = "이름 {} , 나이 {}, 몸무게 {}".format("홍길동", 30, 68.5)
print(mystr)
```

```
import time
mynowtime = time.localtime()

print("현재 시각은 {year}년 {mon}월 {mday}일 {hour}시 입니다.".format(year =
mynowtime.tm_year, mon = mynowtime.tm_mon,
            mday = mynowtime.tm_mday, hour = mynowtime.tm_hour))
```

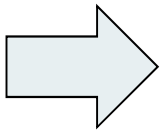
문자열-포맷팅

- 고급 문자열 포맷팅 – 실수 표현

```
fdata = 3.421356
fdata2 = 5.345679
print("{0:0}".format(fdata))
print("{0:0.4f}".format(fdata))
print("{0:10.4f} or {1:10.3f}".format(fdata, fdata2))
```

포맷팅할 데이터 지정 **idx**

지정한 실수 데이터를 자리수 **10**지정하고 소수점 이하 **4**자리까지만 출력



```
3.421356
3.4214
   3.4214 or      5.346
```

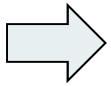
문자열-메소드

- **capitalize() –** 문자열의 첫 문자를 대문자로 변환

```
mystr = "python"
print(mystr.capitalize())
```

- **center() –** 문자열의 형식에 맞게 중앙 정렬

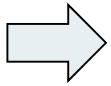
```
menu_start = " menu titie "
print(menu_start.center(30,'#'))
```



######### menu titie #########

- **count() –** 전달된 문자열 과 동일한 부분 문자의 개수 반환

```
mystr = """Time is like Gold.
Time is like an arrow
study Time is import."""
print(mystr.count("Time"))
```



3

문자열-메소드

- **join() –** 인수로 전달된 시퀀스 객체의 각 항목 사이에 특정 문자를 삽입

```
mylist = ["Beautiful!","Explicit!","Simple!","Complex!"]
mystrjoin = '/'.join(mylist)
print(mystrjoin)
```

Beautiful!/Explicit!/Simple!/Complex!

- **split() –** 인수로 전달된 문자로 문자열을 분할하여 리스트 형태로 반환 **// join()**메소드와 반대 개념

```
mysplitlist = mystrjoin.split('/')
print(mysplitlist)
```

['Beautiful!', 'Explicit!', 'Simple!', 'Complex!']

```
myjoinstr = '\n'.join(mysplitlist)
print(myjoinstr)
```

Beautiful!
Explicit!
Simple!
Complex!

문자열-메소드

HANCOM 한컴MDS 아카데미

```
msg = "time is gold"
print(msg)
msg = msg.replace("gold", "arrow")
print(msg)
msg = msg.replace('i', '%')
print(msg)
```

```
time is gold
time is arrow
t%me %s arrow
[Finished in 0.4s]
```

- **replace()** 함수 : 문자열에 있는 특정 문자 또는 문자열을 다른 문자 또는 문자열로 바꿈

```
pin_number = "990415-1234112"
yymmdd = pin_number[:6]
print(yymmdd)
number_id = pin_number[7:]
print(number_id)
```

```
년월일 :  990415
Id :  1234112
[Finished in 1.3s]
```

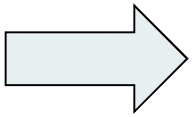
```
if pin_number[7] == '1':
        print("남성")
else:
        print("여성")
```

시퀀스 타입 – 리스트

기본 연산

- 시퀀스 타입이 갖는 공통된 특징으로 색인 연산등 문자열에 사용된 연산이 같은 형태로 사용

```
mystr = "programming"
print(mystr[1])
mylist = ['p','r','o','g','r','a','m','m','i','n','g']
print(mylist[1])
```



색인 연산

```
print(mystr+mystr)
print(mylist+mylist)
```

+ 연산

```
print(mystr*2)
print(mylist*2)
```

* 연산

```
print(mystr[0:5])
print(mylist[0:5])
```

분할 연산

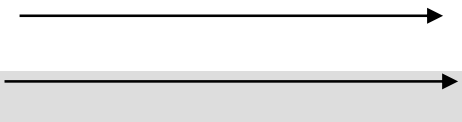
```
print(mystr[0::2])
print(mylist[0::2])
```

확장 분할 연산

```
print(mystr[-1::-1])
print(mylist[-1::-1])
```

역순 접근

```
print(list(mystr))
print("".join(mylist))
```



문자열을 **list** 로

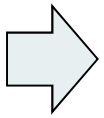
list를 문자열로

시퀀스 타입 – 리스트

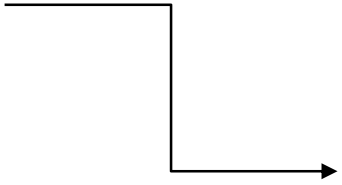
리스트만의 특징

- 문자열과 달리 리스트에 내포되는 객체들의 타입에는 제약이 없으며 객체라면 어떤 타입이든 리스트에 포함 가능

```
mylist = [ 3.14, 5, "python", [100,200,300], ["Good", "programming"]]

print(mylist[0])
print(mylist[1])
print(mylist[2][0])
print(mylist[3][1])
print(mylist[4][1])
print(mylist[4][0][3])
```



```
3.14
5
p
200
programming
d
```



- 실수, 정수, 문자열, 리스트 등 다양한 객체를 항목으로 내포 가능
- 내포된 객체에 대한 색인 연산

시퀀스 타입 – 리스트

리스트만의 특징

```
mystr = "python"
#mystr[0] = 'p'
```

→ **immutable(불변)** 객체로서 수정 불가

```
mylist = ['p','y','t','h','o','n']
mylist[0] = 'P'
print(mylist)
```

→ **mutable(가변)** 객체로서 수정 가능

```
mylistsrc = [ 3, 6, 9]
mylistadd = mylistsrc + [20]
```

→ 새로운 객체 생성

```
print(mylistadd)
print( "mylistsrc id :", id(mylistsrc) )
print( "mylistadd id :", id(mylistadd) )
```

⇒

```
[3, 6, 9, 20]
mylistsrc id : 78857680
mylistadd id : 79218248
```

mutable 객체로서 리스트 항목을 직접 수정하기 위해서는
+=(복합할당연산), apped(), extend(), insert() 메소드 활용

시퀀스 타입 – 리스트

리스트 항목 추가 방법

```
mylistsrc = [ 3, 6, 9]
print(mylistsrc)
print("mylistsrc id :", id(mylistsrc))
mylistsrc += [200,300]
print(mylistsrc)
print("mylistsrc id :", id(mylistsrc))
```

```
[3, 6, 9]
mylistsrc id : 53757392
[3, 6, 9, 200, 300]
mylistsrc id : 53757392
```

복합 할당 연산자 활용

```
mylistsrc = [ 3, 6, 9]
print(mylistsrc)
print("mylistsrc id :", id(mylistsrc))
mylistsrc.append([300,200])
print(mylistsrc)
print("mylistsrc id :", id(mylistsrc))
```

append() 메소드 활용

```
[3, 6, 9]
mylistsrc id : 55264720
[3, 6, 9, [300, 200]]
mylistsrc id : 55264720
```

append 메소드 : 전달된 객체를 그대로 맨 마지막에 추가

시퀀스 타입 – 리스트

리스트 항목 추가 방법

```
mylistsrc = [ 3, 6, 9]
print(mylistsrc)
print("mylistsrc id :", id(mylistsrc))
mylistsrc.extend([200,300])
print(mylistsrc)
print("mylistsrc id :", id(mylistsrc))
```

```
[3, 6, 9]
mylistsrc id : 60310992
[3, 6, 9, 200, 300]
mylistsrc id : 60310992
```

extend() 메소드 활용

extend 메소드 : 전달된 객체**(iterable)**를 맨 마지막에 하나씩 풀어서 추가
전달된 객체는 **iterable** 객체여야 하며 **+=(**복합 할당 연산자**)**와 동일한 동작

```
mylistsrc = [ 3, 6, 9]
print(mylistsrc)
print("mylistsrc id :", id(mylistsrc))
mylistsrc.insert(len(mylistsrc),[200,300])
print(mylistsrc)
print("mylistsrc id :", id(mylistsrc))
```

insert() 메소드 활용

```
[3, 6, 9]
mylistsrc id : 3163600
[3, 6, 9, [200, 300]]
mylistsrc id : 3163600
```

insert 메소드 : **append()**메소드 와 동일 동작이지만 추가될 위치를 지정할 수 있음

- 리스트 정렬 (메소드)

```
mylistdata = [4, 5, 8, 2, 9]
mylistdata.sort()              # 항목의 오름차순 정렬 메소드
print(mylistdata)
mylistdata.reverse()           # 정렬된 항목을 반대 저장
print(mylistdata)
```

- 리스트 정렬 (내장 함수)

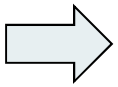
```
mylistdata = [4, 5, 8, 2, 9]
mylistdata2 = sorted(mylistdata)    # 정렬된 리스트 반환
print(mylistdata)                   # 기존 리스트 변화 없음
print(mylistdata2)
```

- Join 메소드 리스트 활용

```
mylistdata = [ "Time ", "is ", "Gold" ]

res = "".join(mylistdata)
print(res)
```

- 리스트 활용 예제

```
myscorelist = [ ["math", 89] , ["english"]]
print(myscorelist)

myscorelist[1].append(95)
print(myscorelist)

myscorelist.append(["korean"])
print(myscorelist)

myscorelist[2].append(79)
print(myscorelist)

sum_data = 0
for data in myscorelist:
        sum_data += data[1]

print(sum_data)
avg = sum_data / 3
print("avg : {0:0.4f}".format(avg))
```



```
[['math', 89], ['english']]
[['math', 89], ['english', 95]]
[['math', 89], ['english', 95], ['korean']]
[['math', 89], ['english', 95], ['korean', 79]]
263
avg : 87.6667
```

- 리스트 **type** 변환

```
mystr = "python"
myconv_tuple = tuple(mystr)
print(myconv_tuple)
mylist = list(myconv_tuple)
print(mylist)
myconv_str = "".join(mylist)
print(myconv_str)
```

- 리스트 활용 예제

```
myscorelist = [ ["math", 89] , ["english"]]
print(myscorelist)

myscorelist[1].append(95)
print(myscorelist)

myscorelist.append(["korean"])
print(myscorelist)

myscorelist[2].append(79)
print(myscorelist)

sum_data = 0
for data in myscorelist:
        sum_data += data[1]

print(sum_data)
avg = sum_data / 3
print("avg : {0:0.4f}".format(avg))
```

- 리스트 **join, split** 활용 예제

```
mylist = [ 3.24, 2.22, 0.12 ]
print(mylist)
print(id(mylist))
mystr = '/'.join([str(i) for i in mylist])
print(mystr)
mylist = mystr.split('/')
print(mylist)
print(id(mylist))
```

- 리스트 복사 활용 예제

```
import copy
mylist_test1 = [ [1,2,3], 3, 5]
#mylist_test2 = copy.deepcopy(mylist_test1)
mylist_test2 = copy.copy(mylist_test1)

print(mylist_test1)
print(id(mylist_test1[0]))
mylist_test1[0].append(99)

print("mylist_test1 : ", mylist_test1)
print("mylist_test2 : ", mylist_test2)
print(id(mylist_test2[0]))
```

깊은 복사 :
mutable한 객체
list 의 복사본 생성

얕은 복사 :
참조 **id**를 복사

튜플

- 튜플 활용 예제

```
def func1(mylist_arg):
        sum_data = 0
        for x in mylist_arg:
                sum_data += x
        avg = sum_data / len(mylist_arg)
        return sum_data, avg


mylist  = [90, 71, 83, 63, 85, 63, 89]
total_avg = func1(mylist)

print(type(total_avg))
print(total_avg)
```

- 튜플 **format** 출력

```
mylist = [123, 567, 89]
mytuple = ("127.0.0.1", 66578)
print("client{} 가  접속".format(mytuple))
print("{} 전송 데이터".format(mylist))
```

- 튜플 덧셈 연산

```
mytupledata = (1, 2, 3)
print(mytupledata)
print(id(mytupledata))
mytupledata = mytupledata + (5,)
print(mytupledata)
print(id(mytupledata))
```

- 리스트와 달리 **immutable** 특징으로**append, extend, insert** 메소드 없음
- 덧셈 연산으로 새로운 객체 생성

사전

- 사전 내용 추가

```
mydict_test = {}

mydict_test["A"] = 0
mydict_test["B"] = 1
mydict_test["C"] = 2

print(mydict_test)
```

Key 값과 **value**를 이용해 사전 내용 추가 가능

≠

- **list** 내용 추가

```
mylist = []
mylist[0] = 5
mylist[1] = 7
print(mylist)
```

IndexError: list assignment index out of range

→

```
mylist = []
mylist.append(1)
mylist.append(2)
print(mylist)
```

- **list**는 **append, expend, insert** 메소드를 활용해 내용 추가

```
a = 77
b = [1,2,3]
c = "python"

dict = { a:b, tuple(b):c, c:a }
print(dict)
```

→ 사전의 **key** 값은 변경 불가능한 **immutable** 객체여야 함

사전

- 사전 내용 삭제

```
dict_data = { 'A':90, 'B':80, 'C':90 }
res = dict_data.pop('B')
print(dict_data)
print(res)
```

```
dict_data = { 'A':90, 'B':80, 'C':90 }
del dict_data['B']
print(dict_data)
```

- 사전 내용 추가 및 변경

```
people_dict = { 'china':13567, 'india':123456, 'america':98345 }
print(people_dict['china'])
people_dict['china'] = 56728
print(people_dict['china'])
people_dict['korea'] = 3267892
print(people_dict)
```

- 사전 **update** 메소드 **//** 동일한 키가 존재하면 항목 수정**,** 동일한 키가 없다면 항목 추가

```
dict_test = { 'first':7, 'second':77, 'third':777 }
modify_dict = { 'first':0, 'end':100 }
for my_key in modify_dict:
        if my_key not in dict_test:
                dict_test.update( { my_key:modify_dict[my_key] } )
print(dict_test)
```

set(셋)

HANCOM 한컴MDS 아카데미

```
mylist = [9,5,3,7,2,1,55, 32, 21, 99,3,9,6,7,6,7,4,1]

myset = set(mylist)
print(myset)

mylist_mod = list(myset)
print(mylist_mod)
mylist_sort = sorted(mylist_mod)
print(mylist_sort)
```

→ 리스트를 **set**으로 중복제거

→ **set**을 리스트로

→ 리스트 항목 정렬

- 세 문자열에서 공통된 문자만 출력

```
mystr1 = "python is simple"
mystr2 = "apple is delicious"
mystr3 = "programming"

myset1 = set(mystr1)
myset2 = set(mystr2)
myset3 = set(mystr3)

myset_res = myset1 & myset2 & myset3
print(myset_res)
```

조건 표현식

```
a = 50; b=70
Max_val =  a if a>b else b
print(Max_val)
```

if else 문을 한 줄로 간결히 표현
A표현식 **if** 조건식 **else B**표현식
조건식이 참이면 **A**표현식 수행
조건식이 거짓이면 **B**표현식 수행

- 리스트 내포

```
mylisttest = [ x  for x in range(1, 10) ]
print(mylisttest)
```

리스트 내포
[표현식 **for x in** 반복가능객체 **]**
표현식을 리스트 항목으로 해서 리스트 생성

- 조건 표현식 활용한 **list** 내포 확장

```
mylist = [ "짝" if x%2 == 0 else "홀" for x in range(1,10) ]
print(mylist)
```

리스트 내포의 표현식을 조건 표현식으로 구현
반복객체의 항목이 홀수이면 "홀"로
반복객체의 항목이 짝수이면 "짝"으로 생성

조건 표현식

- **list** 내포 확장

```
mylist3 = [ 3, 6, 5, 9, 8, 2 , 1]
mylist4 = [ "짝" if x%2 == 0 else "홀" for x in mylist3 ]
print(mylist4)
```

리스트 내포의 표현식을 조건 표현식으로 구현
반복객체의 항목이 홀수이면 "홀"로
반복객체의 항목이 짝수이면 "짝"으로 생성

- 리스트 여과기

```
mylist2 = [ x for x in range(1,10) if x%2 == 0 ]
print(mylist2)
```

리스트 내포에 조건문을 추가하여 여과기 기능
반복객체 중 짝수인 값만 리스트 항목 추가
단, **else** 추가시 **syntax** 오류

반복문

- **1~10**까지 홀수의 합 계산

```
total = 0
for x in range(1,10):
        if x%2 == 0:
                continue
        total += x
print("1~1000 까지 홀수 합 : ", total)
```

- **"python"** 문자열을 **for**문을 활용해 대문자로 변환 하되 **'y'** 문자만 소문자 그대로 표현

```
for ch in "python":
        if ch == 'y':
                print("{}".format(ch))
        else:
                print("{}".format( chr(ord(ch)-32) ) )
```

- 사전 **{ }** 내용을 **key** 와 **value** 구분 하여 출력

```
mydic = { "a":1, "b":2, "c":3, "d":4 }

for dickey in mydic:
        print("{", "{} : {}".format(dickey, mydic[dickey]), "}")
```

반복문

- 반지름을 입력 받아 원의 넓이 및 둘레를 계산, **"end"** 입력시 프로그램 종료

```
while True:
        minput = input("반지름 입력 : ")
        if minput == "end":
                break
        else:
                myradius = float(minput)
        print("myradius : ", myradius)
        circle_area = myradius * myradius * 3.14
        circle_len = 2 * 3.14 * myradius
        print("area : ", circle_area)
        print("len : ", circle_len)
```

- **list** 내포를 활용 **1~100**까지 숫자 중 **2**와 **3**의 공배수 중 **4**의 배가가 아닌 수의 리스트 생성

```
mylistdata = [x for x in range(1,101) if ((x%2==0 and x%3==0) and (x%4 != 0)) ]
print(mylistdata)
```

반복문

- 문자열 데이터 중 숫자문자 나 특수 문자 제거한 영문대(소) 문자만 추출

```
mystrdata = "#s45cD!K2ab@"
mylistdata = []
for ch in mystrdata:
        if ch >= 'a' and ch <= 'z':
                mylistdata.append(ch)
        elif ch >='A' and ch <= 'Z':
                mylistdata.append(ch)

print(mylistdata)
mystr_cov = "".join(mylistdata)
print(mystr_cov)
```

- list 내포를 활용 1~100까지 숫자 중 2와 3의 공배수 중 4의 배가가 아닌 수의 리스트 생성

```
mylistdata = [x for x in range(1,101) if ((x%2==0 and x%3==0) and (x%4 != 0)) ]
print(mylistdata)
```

- list 내포를 활용 range(1,20) 을 순회하여 3으로 나누었을 때 나머지가 1 이면 'A'
  나머지가 2이면 'B', 나머지가 0이면 'C'로 리스트 생성

```
mylistdata = [ 'A' if x%3 == 1 else 'B' if x%3 == 2 else 'C' for x in range(1,20) ]
print(mylistdata)
```

Thank you

㈜한컴MDS www.hancommds.com

본사 13493 경기도 성남시 분당구 대왕판교로 644번길 49 한컴타워 3,4층 031-627-3000
연구소 13487 경기도 성남시 분당구 판교로 228번길 17 판교세븐벤처밸리 2단지 1동 9층 031-600-5000