

Ch3. 변수와 입출력

- 변수 만들기

- 파이썬에서는 다음 그림과 같은 형식으로 코드를 입력하여 변수를 만듦

할당

X = 10

↑ ↑

변수 이름 값

- 변수 만들기

- $x = 10$ 이라고 입력하면 10이 들어있는 변수 x 가 만들어짐
- 즉, 변수이름 = 값 형식임
- 변수가 생성되는 동시에 값이 할당(저장)됨
- 변수 이름은 원하는 대로 지으면 되지만 다음과 같은 규칙을 지켜야 함

- 영문 문자와 숫자를 사용할 수 있습니다.
- 대소문자를 구분합니다.
- 문자부터 시작해야 하며 숫자부터 시작하면 안 됩니다.
- _(밑줄 문자)로 시작할 수 있습니다.
- 특수 문자(+, -, *, /, \$, @, &, % 등)는 사용할 수 없습니다.
- 파이썬의 키워드(if, for, while, and, or 등)는 사용할 수 없습니다.

- 변수 만들기

- 파이썬 셸에서 변수를 만들어보자
- >>>에 다음 코드를 입력

```
>>> x = 10
>>> x
10
```

- 변수 x를 만들면서 10을 할당함
- 파이썬 셸에서는 변수를 입력한 뒤 엔터 키를 누르면 변수에 저장된 값이 출력
- 변수에는 숫자 뿐만 아니라 문자열도 넣을 수 있음

```
>>> y = 'Hello, world!'
>>> y
'Hello, world!'
```

- ' '(작은따옴표)로 묶은 문자열 Hello, world!를 변수 y에 할당함

- 변수의 자료형 알아내기

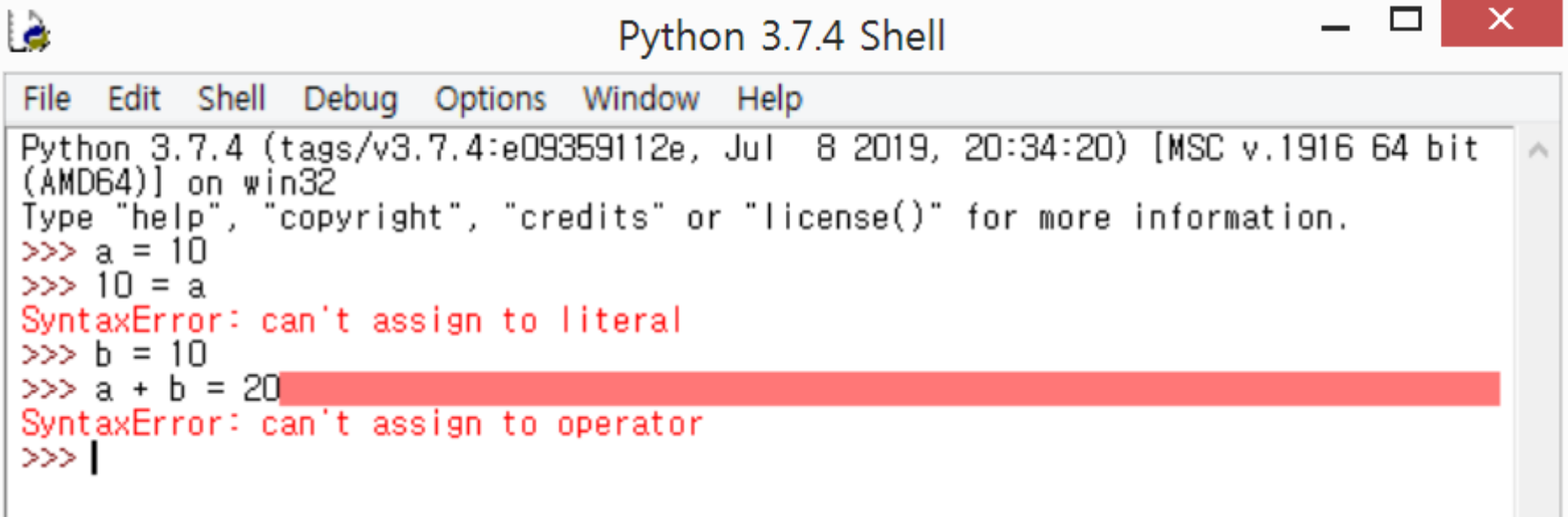
- 파이썬에서는 변수의 자료형이 중요함
- type에 변수를 넣으면 변수(객체)의 자료형이 나옴

- type(변수)

```
>>> type(x)
<class 'int'>
>>> type(y)
<class 'str'>
```

- x에는 정수 10이 들어있으므로 int, y에는 문자열 Hello, world!가 들어있으므로 str이라고 나옴(int는 정수(integer), str은 문자열(string)에서 따옴)
- 변수의 자료형은 변수에 들어가는 값에 따라 달라짐
- 파이썬에서 변수를 사용하다 보면 자료형이 맞지 않아 발생하는 문제를 자주 접하게 됨
- type으로 자료형을 알아보면 문제를 쉽게 해결할 수 있음

- cf) =기호는 같다는 뜻 아닌가요?
 - 수학에서는 =(등호) 기호는 양 변이 같다는 뜻?
 - 프로그래밍 언어에서 =는 변수에 값을 할당(**assignment**)한다는 의미
 - 수학의 등호와 같은 역할을 하는 연산자는 ==임

A screenshot of a Python 3.7.4 Shell window. The window has a title bar with the text 'Python 3.7.4 Shell' and standard window controls (minimize, maximize, close). Below the title bar is a menu bar with 'File', 'Edit', 'Shell', 'Debug', 'Options', 'Window', and 'Help'. The main area of the window contains a text-based interface for the Python interpreter. It shows the version and platform information: 'Python 3.7.4 (tags/v3.7.4:e09359112e, Jul 8 2019, 20:34:20) [MSC v.1916 64 bit (AMD64)] on win32'. It also displays instructions: 'Type "help", "copyright", "credits" or "license()" for more information.' The user has entered three lines of code: '>>> a = 10', '>>> 10 = a', and '>>> a + b = 20'. The first line is successful. The second line results in a red error message: 'SyntaxError: can't assign to literal'. The third line also results in a red error message: 'SyntaxError: can't assign to operator'. The prompt '>>>' is followed by a vertical bar cursor on the last line.

```
Python 3.7.4 Shell
File Edit Shell Debug Options Window Help
Python 3.7.4 (tags/v3.7.4:e09359112e, Jul 8 2019, 20:34:20) [MSC v.1916 64 bit
(AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>> a = 10
>>> 10 = a
SyntaxError: can't assign to literal
>>> b = 10
>>> a + b = 20
SyntaxError: can't assign to operator
>>> |
```

- 변수 여러 개를 한 번에 만들기

- 파이썬에서는 변수 여러 개를 한 번에 만들 수도 있음

```
>>> x, y, z = 10, 20, 30
>>> x
10
>>> y
20
>>> z
30
```

- 변수이름1, 변수이름2, 변수이름3 = 값1, 값2, 값3 형식으로 변수를 ,(콤마)로 구분한 뒤 각 변수에 할당될 값을 지정
- 변수와 값의 개수는 동일하게 맞춰주어야 하며 나열된 순서대로 값이 할당
- 변수와 값의 개수가 맞지 않으면 이렇게 에러가 발생

```
>>> x, y, z = 10, 20
Traceback (most recent call last):
  File "<pyshe11#3>", line 1, in <module>
    x, y, z = 10, 20
ValueError: not enough values to unpack (expected 3, got 2)
```

- 변수 여러 개를 한 번에 만들기

- 변수 여러 개를 만들 때 값이 모두 같아도 된다면 다음과 같은 방식도 사용가능

```
>>> x = y = z = 10
>>> x
10
>>> y
10
>>> z
10
>>>
```

- 변수 3개를 만들면서 모두 같은 값을 할당
- 변수1 = 변수2 = 변수3 = 값 형식으로 변수 여러 개를 =로 연결하고
마지막에 값을 할당해주면 같은 값을 가진 변수 3개가 만들어짐

- 변수 여러 개를 한 번에 만들기

- 두 변수의 값을 바꾸려면 어떻게 해야 할까?
- 변수를 할당할 때 서로 자리를 바꿔주면 됨

```
>>> x, y = 10, 20
>>> x, y = y, x
>>> x
20
>>> y
10
```

- x는 20, y는 10이 나왔음
- 변수1, 변수2 = 변수2, 변수1 형식으로 두 변수의 값을 바꿀 수 있음

- cf) 변수 삭제하기

- 변수 삭제는 del을 사용

del 변수

```
>>> x = 10
>>> del x
>>> x
Traceback (most recent call last):
  File "<pyshell#2>", line 1, in <module>
    x
NameError: name 'x' is not defined
>>>
```

- 변수 x를 삭제하여 변수가 없어졌으므로 x가 정의되지 않았다는 메시지와 함께 NameError가 발생
 - 지금은 변수 삭제가 큰 의미가 없지만 나중에 리스트를 사용할 때 del이 유용하게 쓰임

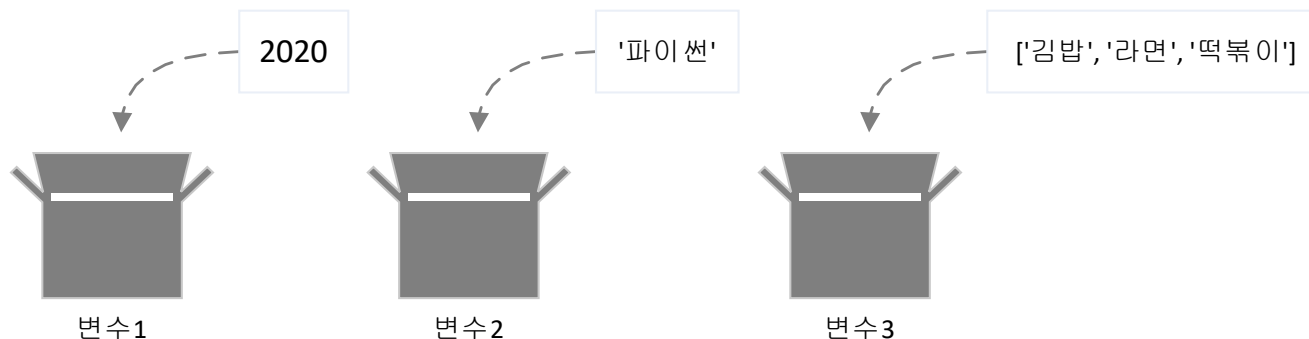
- cf) 빈 변수 만들기

- 변수를 만들 때 $x = 10$ 과 같이 할당할 값을 지정해줌
- 값이 들어있지 않는 변수는 만들 수 있을까?
- 값이 들어있지 않은 빈 변수를 만들 때는 **None**을 할당해주면 됨

```
>>> x = None
>>> print(x)
None
>>> x
>>> (아무것도 출력되지 않음)
```

- print로 변수 x의 값을 출력해보면 **None**이 나옴
- 파이썬에서 **None**은 아무것도 없는 상태를 나타내는 자료형
- 보통 다른 언어에서는 널(null)이라고 표현함

- cf) 동적 타이핑 변수
- 정적 타이핑(Static Typing)
 - C, Java 등의 컴파일러 방식의 언어에서 사용
 - 저장할 상자(변수)를 먼저 준비(선언)한 후에 내용(데이터)을 저장함
 - 컴파일러는 상자의 크기에 맞게 메모리 공간(바이트)을 배정함
 - 선언된 변수는 사용이 끝날 때까지 계속 같은 메모리 주소를 갖게 됨
 - 상자에 맞지 않는 내용(혹은 크기)을 저장하려 할 경우는 에러 발생
 - 잘못된 코드를 사전에 예방할 수 있어 안정도가 높음
 - 변수 사용시 먼저 선언을 해야 해서 프로그래밍이 번거로움



- 정적 타이핑(Static Typing)

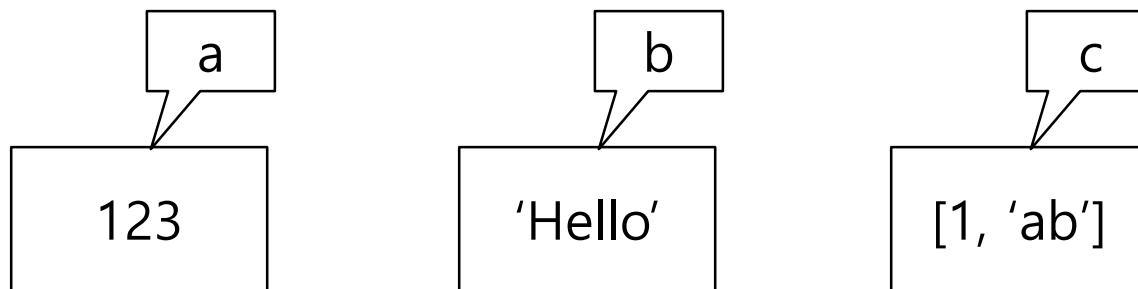
- 변수의 이름이 다른 경우, 내용은 같아도 다른 메모리 주소를 갖게 됨
- 한 상자(a)에 지우개가 있고, 다른 상자(b)에는 연필이 있을 경우, 내용을 서로 바꾸기(swap) 위해서는 임시로 보관할 상자가 하나(c) 더 있어야 함
- 두 변수의 값을 바꾸기 위해서는, 먼저 한 변수 값을 임시로 저장한 후 다시 그 값을 가져와서 저장해야 함
- 왼쪽 프로그램은 실행 후 a=20, b=20 으로 모두 20 값을 갖게 됨
- 오른쪽 프로그램 실행 후 a=20, b=10 으로 변경된 값이 됨

```
int a, b;  
a = 10;  
b = 20;  
  
// swap  
a = b;  
b = a;
```

```
int a, b, c;  
a = 10;  
b = 20;  
  
// swap  
c = a;  
a = b;  
b = c;
```

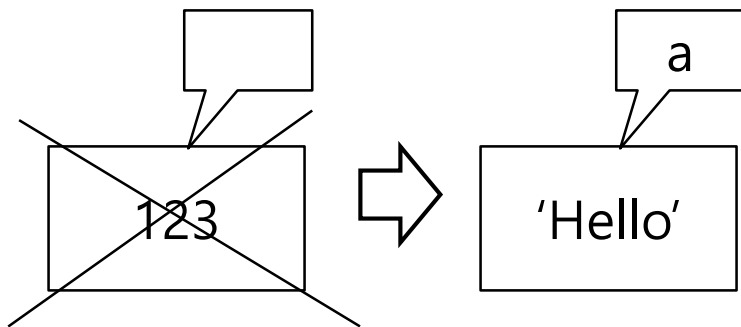
- 동적 타이핑(Dynamic Typing)

- Python, R, Javascript 등의 스크립트 방식의 언어에서 사용
- 선언할 필요 없이 임의 타입의 데이터를 변수에 저장함
- 변수에 데이터를 저장, 변경하는데 제한이 없음
- 상자 보다는 상자에 붙인 태그라고 생각할 수 있음
- 먼저 데이터를 비어 있는 메모리 공간에 저장한 후 태그(변수명)를 붙임
- 실행시 변수의 타입에 따라 연산에러가 발생할 수 있어 안정도가 떨어짐
- 선언을 할 필요가 없어 프로그래밍이 편하고 간결해짐



- 동적 타이핑(Dynamic Typing)

- a 내용이 변경되는 경우, 새로운 내용을 저장하고 그 태그를 a 라고 함
- 예, a = 123 실행 후에, a = 'Hello' 실행하는 경우
 - 새로운 공간(주소)에 'Hello' 를 저장한 후 그 태그를 a 라고 함
 - 앞서 123 저장했던 공간은 더 이상 사용할 수 없게 됨 (Garbage)
- Garbage가 많아지면 메모리 부족이 발생하여, 주기적으로 메모리를 복구하는 처리(Garbage Collection, gc)를 해야 함
- gc는 프로그램에서 중요한 부분으로, 파이썬에서는 자동으로 처리됨
- id(a)는 a 변수의 주소를 보여줌 (C언어의 포인터와 유사. 중요하지 않음)



```
>>> a=123
>>> id(a)
8770073100352
>>> a='Hello'
>>> id(a)
1054514103984
```

- 동적 타이핑(Dynamic Typing)

- 두 변수의 내용이 같을 경우, 같은 메모리 번지를 갖게 됨

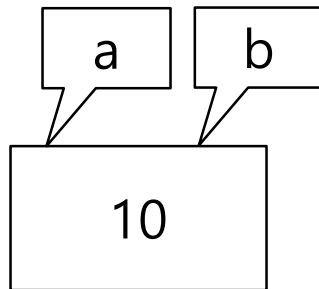
- $a = 10$ 혹은 $a = 10$

- $b = 10$ $b = a$

- 할당(=)이 실행되면,

- 정적 타이핑의 경우 미리 정해진 주소에 입력해서 실행시간이 짧고

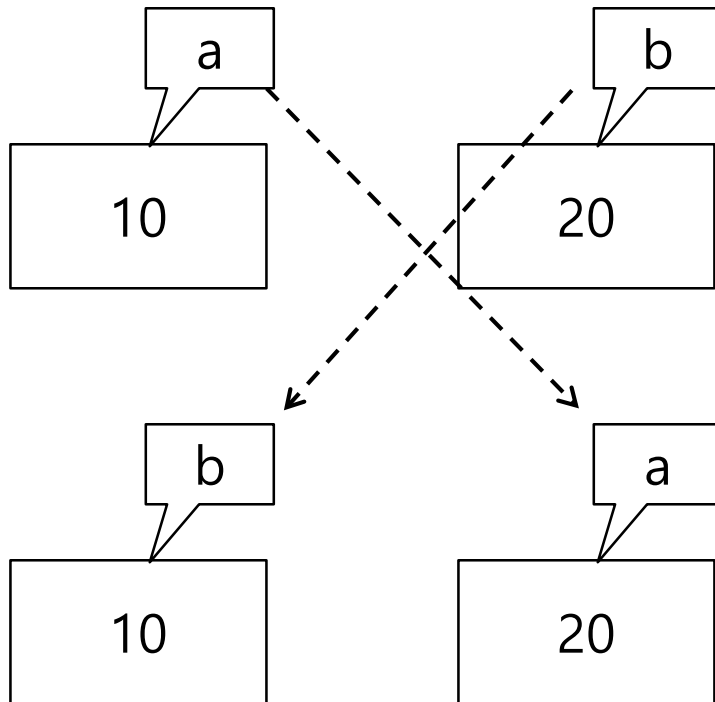
- 동적 타이핑의 경우 그 값이 있는지 확인한 후 없으면 적당한 공간을 찾아 처리해서 실행시간이 느려짐



```
>>> a=10
>>> b=10
>>> id(a)
8770069754400
>>> id(b)
8770069754400
```


- 동적 타이핑(Dynamic Typing)

- a, b 값을 바꾸는(swap) 경우 태그를 바꿔 붙임
- 임시 변수 c 가 필요 없음 => a, b = b, a
- 즉, 10, 20 저장되어 있는 메모리 주소는 그대로이고, 태그만 옮겨 붙임



```
>>> a=10
>>> b=20
>>> id(a)
8770073096736
>>> id(b)
8770073097056
>>> a, b = b, a
>>> a
20
>>> b
10
>>> id(a)
8770073097056
>>> id(b)
8770073096736
```

- 변수로 계산하기

- 변수를 활용하여 계산을 해보자

```
>>> a = 10
>>> b = 20
>>> c = a + b
>>> c
30
```

- 변수 a, b에 숫자를 할당한 뒤에 a와 b의 값을 더해서 변수 c에 할당함
- 변수는 변수끼리 계산할 수 있고, 계산 결과를 다른 변수에 할당할 수 있음

- 산술 연산 후 할당 연산자 사용하기

- 변수 a 의 값을 20 증가시키려면 어떻게 해야 할까?
- $a + 20$ 처럼 20을 더하면 30이 나오지만 a 의 값을 다시 출력해보면 10이 나옴

```
>>> a = 10
>>> a + 20
30
>>> a
10
```

- $a + 20$ 은 a 에 20을 더하기만 할 뿐 계산 결과를 유지하지 않음
- 변수 한 개에서 값의 변화를 계속 유지하려면 계산 결과를 다시 변수에 저장

```
>>> a = 10
>>> a = a + 20    # a와 20을 더한 후 결과를 다시 a에 저장
>>> a
30
```

- 산술 연산 후 할당 연산자 사용하기

- $a = a + 20$ 과 같이 a 에 20을 더한 값을 다시 a 에 할당해주면 계산 결과가 유지
- 파이썬에서는 변수를 두 번 입력하지 않도록 산술 연산 후 할당 연산자를 제공

```
>>> a = 10
>>> a += 20    # a와 20을 더한 후 결과를 다시 a에 저장
>>> a
30
```

- a 에는 10이 들어있고 $a += 20$ 을 수행하면 a 에는 10과 20을 더한 결과인 30 저장
- $+=$ 처럼 산술 연산자 앞에 $=$ (할당 연산자)를 붙이면 연산 결과를 변수에 저장
($+ =$ 처럼 두 연산자를 공백으로 띄우면 안 됨)
- 뺄셈($-=$), 곱셈($*=$), 나눗셈($/=$, $//=$), 나머지($\%=$)도 같은 방식
- 똑같이 연산($-$, $*$, $/$, $//$) 후 할당($=$) 한다는 뜻

```
>>> a=10
>>> a+=20
>>> a
30
>>> a+ = 10
SyntaxError: invalid syntax
>>> |
```

- 산술 연산 후 할당 연산자 사용하기

- 산술 연산 후 할당 연산자를 사용할 때는 주의할 점이 있음
- 만들지 않은 변수 d에 10을 더한 후 다시 할당하면 에러가 발생

```
>>> d += 10    # d는 만들지 않은 변수
Traceback (most recent call last):
  File "<pyshell#2>", line 1, in <module>
    d += 10
NameError: name 'd' is not defined
```

- 계산 결과를 d에 할당하기 전에 d와 10을 더해야 하는데 d라는 변수가 없어서 덧셈이 안 되기 때문임
- 연산 후 할당을 하려면 값이 들어있는 변수를 사용해야 함

- cf) 부호 붙이기

- 계산을 하다 보면 부호를 붙여야 하는 경우도 생김
- 값이나 변수 앞에 양수, 음수 부호를 붙이면 됨

```
>>> x = -10
>>> +x
-10
>>> -x
10
```

- 양수, 음수 부호 붙이기는 수학 시간에 배운 내용과 같음
- -10에 + 부호를 붙이면 부호의 변화가 없고, -10에 - 부호를 붙이면 양수 10이 됨

- **입력 값을 변수에 저장하기**
 - 지금까지 변수를 만들 때 10, 'Hello, world!' 등의 값을 직접 할당
 - 이렇게 하면 고정된 값만 사용할 수 있음
 - 매번 다른 값을 변수에 할당하려면 어떻게 해야 할까?

- **input 함수 사용하기**

- 이때는 input 함수를 사용하면 됨
- >>>에 input()을 입력한 뒤 엔터 키를 누르면 다음 줄로 넘어감
- 이 상태에서 Hello, world!를 입력한 뒤 엔터 키를 누름

```
>>> input()
Hello, world! (입력)
'Hello, world!'
```

- 입력한 문자열이 그대로 출력됨
- input 함수는 사용자가 입력한 값을 가져오는 함수

- **input 함수의 결과를 변수에 할당하기**

- input 함수의 결과를 변수에 할당해보자 • 변수 = input()
- input 함수의 결과를 변수 x에 할당함
- Hello, world!를 입력한 뒤 엔터 키를 누름

```
>>> x = input()
Hello, world! (입력)
>>>
```

- 변수 x에 입력한 문자열이 저장됨
- x의 값을 출력해보면 방금 입력한 'Hello, world!'가 나옴

```
>>> x
'Hello, world!'
```

- 여기서 한 가지 불편한 점은 input 함수가 실행된 다음에는 아무 내용이 없어서 입력을 받는 상태인지 출력이 없는 상태인지 알 수가 없음

- **input 함수의 결과를 변수에 할당하기**

- input의 괄호 안에 문자열을 지정해주면 됨

- 변수 = input('문자열')

```
>>> x = input('문자열을 입력하세요: ')
문자열을 입력하세요: Hello, world! (입력)
>>> x
'Hello, world!'
```

- 실행을 해보면 '문자열을 입력하세요:'처럼 안내 문구가 먼저 나옴
- 문자열을 입력한 뒤 엔터 키를 누르면 입력한 그대로 출력
- 문자열은 스크립트 파일 사용자에게 입력 받는 값의 용도를 미리 알려줄 때 사용
- 다른 말로는 프롬프트(prompt)라고도 부릅니다(파이썬 프롬프트 >>>와 같은 의미)

- 두 숫자의 합 구하기

- 조금 응용해서 숫자 두 개를 입력 받은 뒤에 두 숫자의 합을 구해보자
- IDLE의 소스 코드 편집 창에 입력

input_integer.py

```
a = input('첫 번째 숫자를 입력하세요: ')\nb = input('두 번째 숫자를 입력하세요: ')\n\nprint(a + b)
```

- 소스 코드를 실행하면 '첫 번째 숫자를 입력하세요: '가 출력
- 10을 입력하고 엔터 키를 누름
- '두 번째 숫자를 입력하세요: '가 출력되면 20를 입력하고 엔터 키를 누름

실행 결과

```
첫 번째 숫자를 입력하세요: 10 (입력)\n두 번째 숫자를 입력하세요: 20 (입력)\n1020
```

- 두 숫자의 합 구하기

- 10 + 20의 결과는 30이 나와야 하는데 1020이 나옴
- 이런 결과는 input에서 입력 받은 값은 항상 문자열 형태이기 때문임
- 10과 20은 겉보기에는 숫자이지만 실제로는 문자열이므로 10과 20을 +로 연결하여 1020이 나오게 됨
- input의 결과를 변수에 저장한 뒤 type을 사용해보면 input의 결과가 문자열(str)이라는 것을 알 수 있음

실행 결과

```
첫 번째 숫자를 입력하세요: 10 (입력)  
두 번째 숫자를 입력하세요: 20 (입력)  
1020
```

```
>>> a = input()  
10 (입력)  
>>> type(a)  
<class 'str'>
```

- **입력 값을 정수로 변환하기**

- 10 + 20의 결과가 30이 나오게 하려면 input에서 입력 받은 문자열을 숫자(정수)로 만들어주어야 함

- 변수 = int(input())
- 변수 = int(input('문자열'))

input_integer.py

```
a = int(input('첫 번째 숫자를 입력하세요: '))    # int를 사용하여 입력 값을 정수로 변환
b = int(input('두 번째 숫자를 입력하세요: '))    # int를 사용하여 입력 값을 정수로 변환

print(a + b)
```

실행 결과

```
첫 번째 숫자를 입력하세요: 10 (입력)
두 번째 숫자를 입력하세요: 20 (입력)
30
```

- 입력 받은 값을 숫자(정수)로 만들려면 int에 input()을 넣음
- int는 정수로 된 문자열도 정수로 만들 수 있으므로 문자열 '10'은 정수 10으로 바뀜
- 3.5와 2.1처럼 실수를 더하려면 int 대신 float에 input()을 넣음
- 각자 소스 코드를 수정하여 실수의 합도 구해보자

- 입력 값을 변수 두 개에 저장하기

- input 한 번에 값을 여러 개 입력 받으려면 어떻게 해야 할까?
- input에서 split을 사용한 변수 여러 개에 저장하면 됨(각 변수는 콤마로 구분해줌)
 - 변수1, 변수2 = input().split()
 - 변수1, 변수2 = input().split('기준문자열')
 - 변수1, 변수2 = input('문자열').split()
 - 변수1, 변수2 = input('문자열').split('기준문자열')
- 문자열 두 개를 입력 받아 보자
- IDLE의 소스 코드 편집 창에 입력

input_split_string.py

```
a, b = input('문자열 두 개를 입력하세요: ').split()    # 입력받은 값을 공백을 기준으로 분리

print(a)
print(b)
```

- **입력 값을 변수 두 개에 저장하기**

- 소스 코드를 실행하면 '문자열 두 개를 입력하세요: '가 출력됨
- Hello Python을 입력하고 엔터 키를 누름

실행 결과

```
문자열 두 개를 입력하세요: Hello Python (입력)
Hello
Python
```

- **input에 split을 사용하면 입력받은 값을 공백을 기준으로 분리하여 변수에 차례대로 저장**
(split은 분리하다, 나누다라는 뜻)
- 문자열 'Hello Python'을 공백을 기준으로 분리하여 'Hello'는 첫 번째 변수 a에 'Python'은 두 번째 변수 b에 저장

```
a , b = input('문자열 두 개를 입력하세요: ').split()
```

- 두 숫자의 합 구하기

- 숫자 두 개를 입력 받아서 두 숫자의 합을 구해보자

input_split_int.py

```
a, b = input('숫자 두 개를 입력하세요: ').split()    # 입력받은 값을 공백을 기준으로 분리
print(a + b)
```

실행 결과

```
숫자 두 개를 입력하세요: 10 20 (입력)
1020
```

- 30이 나와야 하는데 1020이 나옴
- input에서 입력 받은 값은 문자열이고, 이 문자열은 split으로 분리해도 문자열이기 때문임
- 문자열 '10 20'을 공백을 기준으로 분리하여 a에는 '10', b에는 '20'이 저장되므로 +로 연결하면 '1020'이 나옴

```
a , b = input('숫자 두 개를 입력하세요: ').split()
```


- **입력 값을 정수로 변환하기**

- 10 + 20의 결과가 30이 나오게 하려면 변수 a와 b를 정수로 변환해주어야 함

input_split_int.py

```
a, b = input('숫자 두 개를 입력하세요: ').split()    # 입력받은 값을 공백을 기준으로 분리
a = int(a)      # 변수를 정수로 변환한 뒤 다시 저장
b = int(b)      # 변수를 정수로 변환한 뒤 다시 저장

print(a + b)
```

실행 결과

```
숫자 두 개를 입력하세요: 10 20 (입력)
30
```

- a = int(a)와 같이 int에 변수를 넣은 뒤 다시 변수에 저장해주면 변수가 정수 자료형으로 변환됨
- int(a)처럼 int만 사용하고 결과를 변수에 저장하지 않으면 정수로 변환되지 않음
- print 안에서 int로 변수를 변환하고 바로 더해도 상관없음

```
print(int(a) + int(b))
```

- **map을 사용하여 정수로 변환하기**

- map에 int와 input().split()을 넣으면 split의 결과를 모두 int로 변환해줌
(실수로 변환할 때는 int 대신 float를 넣음)

- 변수1, 변수2 = map(int, input().split())
- 변수1, 변수2 = map(int, input().split('기준문자열'))
- 변수1, 변수2 = map(int, input('문자열').split())
- 변수1, 변수2 = map(int, input('문자열').split('기준문자열'))

- IDLE의 소스 코드 편집 창에 입력

map_input_split.py

```
a, b = map(int, input('숫자 두 개를 입력하세요: ').split())  
  
print(a + b)
```

실행 결과

```
숫자 두 개를 입력하세요: 10 20 (입력)  
30
```

- input().split()을 사용할 때 map을 사용하면 코드를 짧게 줄일 수 있음

- **입력 받은 값을逗를 기준으로 분리하기**

- split에 기준 문자열을 지정하여 공백이 아닌 다른 문자로 분리해보자

map_input_split_comma.py

```
a, b = map(int, input('숫자 두 개를 입력하세요: ').split(',')) # 입력받은 값을逗를 기준으로 분리  
  
print(a + b)
```

- 코드를 실행한 뒤 '숫자 두 개를 입력하세요: '가 출력되면 10,20을 입력하고 엔터

실행 결과

```
숫자 두 개를 입력하세요: 10,20 (입력)  
30
```

- split(',')과 같이 분리할 기준 문자열을逗로 지정하였으므로 '10,20'에서 10은 a, 20은 b에 저장
- 변수는 값이나 계산 결과를 저장할 때 사용한다는 점, 변수를 만드는 방법, 변수 이름을 짓는 방법만 기억하면 됨
- **input과 split의 결과가 문자열라는 점이 중요**
- 숫자 계산을 한다면 int, float를 사용해서 결과를 숫자로 변환해주어야 한다는 점
- split의 결과를 모두 int, float로 변환할 때는 map을 사용하면 편리함

- 값을 여러 개 출력하기

- print에는 변수나 값 여러 개를 ,(кома)로 구분하여 넣을 수 있음

- print(값1, 값2, 값3)
- print(변수1, 변수2, 변수3)

```
>>> print(1, 2, 3)
1 2 3
>>> print('Hello', 'Python')
Hello Python
```

- print에 변수나 값을 콤마로 구분해서 넣으면 값이 공백으로 띄워져서 한 줄로 출력
- 값을 여러 개 출력할 때 print 함수를 여러 번 쓰지 않아도 됨

- **sep로 값 사이에 문자 넣기**

- 값 사이에 공백이 아닌 다른 문자를 넣고 싶을 때 print의 sep에 문자 또는 문자열을 지정해주면 됨(sep는 구분자라는 뜻의 separator에서 따왔음)

- `print(값1, 값2, sep='문자 또는 문자열')`
- `print(변수1, 변수2, sep='문자 또는 문자열')`

```
>>> print(1, 2, 3, sep=', ')    # sep에 콤마와 공백을 지정
1, 2, 3
>>> print(4, 5, 6, sep=',')    # sep에 콤마만 지정
4,5,6
>>> print('Hello', 'Python', sep='')    # sep에 빈 문자열을 지정
HelloPython
>>> print(1920, 1080, sep='x')    # sep에 x를 지정
1920x1080
```

- `sep=', '`처럼 콤마와 공백을 넣어주면 1, 2, 3과 같은 형태로 출력
- 공백 없이 콤마만 지정해도 됨
- `sep=""`처럼 빈 문자열을 지정하면 각각의 값은 서로 붙어서 출력
- sep에는 'x'와 같은 일반적인 문자도 넣을 수 있음

- **줄바꿈 활용하기**

- 줄바꿈(개행)을 활용해보자
- print에 값을 여러 개 지정하면 한 줄에 모든 값이 출력

```
>>> print(1, 2, 3)
1 2 3
```

- print 한 번으로 값을 여러 줄에 출력할 수는 없을까?
- print의 sep에 개행 문자(\n)라는 특별한 문자를 지정하면 값을 한 줄에 하나씩 출력할 수 있음

```
>>> print(1, 2, 3, sep='\n')
1
2
3
```

- **줄바꿈 활용하기**

- `\n`은 값을 다음 줄에 출력하게 만드는 제어 문자
- `sep`에 `\n`을 지정하면 1 2 3 사이에 `\n`이 들어가므로 1을 출력한 뒤 다음 줄에 2를 출력하고 다시 다음 줄에 3을 출력하게 됨
- 단, `\n` 자체는 제어 문자이므로 화면에 출력되지 않음

```
1\n2 \n3
```

- 줄바꿈 활용하기

- 참고로 \n도 문자이므로 print에 바로 넣어서 사용할 수도 있음
- print(1, 2, 3, sep='\n')와 결과가 같음

```
>>> print('1\n2\n3')    # 문자열 안에 \n을 사용하여 줄바꿈
1
2
3
```

- '1\n2\n3'과 같이 \n은 옆에 다른 문자나 숫자와 붙여서 씀
- \n 양 옆에 공백을 넣어버리면 공백이 그대로 출력되므로 주의해야 함

- **cf) 제어 문자**

- 제어 문자는 화면에 출력되지는 않지만 출력 결과를 제어한다고 해서 제어 문자라 부름
- 제어 문자는 ₩로 시작하는 이스케이프 시퀀스
- **₩n**: 다음 줄로 이동하며 개행이라고도 부름
- **₩t**: 탭 문자, 키보드의 Tab 키와 같으며 여러 칸을 띄움
- **₩₩**: ₩ 문자 자체를 출력할 때는 ₩를 두 번 써야 함

- **end 사용하기**

- print는 기본적으로 출력하는 값 끝에 \n을 붙임
- print를 여러 번 사용하면 값이 여러 줄에 출력됨
- IDLE의 소스 코드 편집 창에 입력한 뒤 실행

print_multiple.py

```
print(1)
print(2)
print(3)
```

실행 결과

```
1
2
3
```

- **end 사용하기**

- print를 여러 번 사용해서 print(1, 2, 3)처럼 한 줄에 여러 개의 값을 출력할 수는 없을까?
- print의 end에 빈 문자열을 지정해주면 됨

- `print(값, end='문자 또는 문자열')`
- `print(변수, end='문자 또는 문자열')`

print_multiple_end.py

```
print(1, end='') # end에 빈 문자열을 지정하면 다음 번 출력이 바로 뒤에 오게 됨
print(2, end='')
print(3)
```

실행 결과

123

- end 사용하기

- end="와 같이 end에 빈 문자열을 지정하면 1, 2, 3이 세 줄로 출력되지 않고 한 줄로 붙어서 출력됨
- 기본적으로 print의 end에 \n이 지정된 상태인데 빈 문자열을 지정하면 강제로 \n을 지워주기 때문임
- end는 현재 print가 끝난 뒤 그 다음에 오는 print 함수에 영향을 줌
- 1 2 3 사이를 띄워주고 싶다면 end에 공백 한 칸을 지정하면 됨

```
print(1, end=' ')    # end에 공백 한 칸 지정
print(2, end=' ')
print(3)
```

실행 결과

1 2 3

- print의 sep, end에 제어 문자, 공백 문자 등을 조합하면 다양한 형태로 값을 출력할 수 있음
- 자신의 상황에 맞게 선택해서 사용하면 됨

- 문자열 사용하기

- 문자열에 대해 좀 더 자세히 알아보자
- 먼저 간단하게 파이썬 프롬프트에서 문자열 'Hello, world!'를 출력해보자

```
>>> hello = 'Hello, world!'
>>> hello
'Hello, world!'
```

- Hello, world!를 ' '(작은따옴표)로 묶어서 문자열로 만들었음
- 문자열은 영문 문자열뿐만 아니라 한글 문자열도 사용할 수 있음

```
>>> hello = '안녕하세요'
>>> hello
'안녕하세요'
```

- 문자열 사용하기

- 파이썬에서는 작은따옴표로 묶는 방법 이외에도 문자열을 만드는 여러 가지 방법이 있음
- " "(큰따옴표)로 묶는 방법

```
>>> hello = "Hello, world!"  
>>> hello  
'Hello, world!'
```

- '''(작은따옴표 3개)로 묶거나 """(큰따옴표 3개)로 묶을 수도 있음

```
>>> hello = '''Hello, Python!'''  
>>> hello  
'Hello, Python!'  
>>> python = """Python Programming"""  
>>> python  
'Python Programming'
```

- 여러 줄로 된 문자열 사용하기

- 여러 줄로 된 문자열(multiline string)을 사용해보자
- '''(작은따옴표 3개)로 시작하고 Hello, world!를 입력한 다음에 엔터 키를 누르면 다음 줄로 이동함
- 문자열을 계속 입력하고 마지막 줄에서 '''로 닫은 뒤 엔터 키를 누르면
>>> 프롬프트로 돌아옴

```
>>> hello = '''Hello, world!  
안녕하세요.  
Python입니다.'''  
>>> print(hello)  
Hello, world!  
안녕하세요.  
Python입니다.
```

- 여러 줄로 된 문자열 사용하기

- print 함수로 hello 의 내용을 출력해보면 입력한 문자열 3 줄이 출력됨
- 사실 파이썬 셸에서는 여러 줄로 된 문자열을 사용할 일이 많지 않음
- 여러 줄로 된 문자열은 주로 .py 스크립트 파일에서 사용
- 여러 줄로 된 문자열은 '''(작은따옴표 3개)로 시작하여 '''로 끝남
- """(큰따옴표 3개)로 시작하여 """로 끝내도 됨
- 문자열을 표현할 때 작은따옴표와 큰따옴표 중 한 가지로 통일하지 않고 여러 가지 방식을 사용하는지 알아보자

- 문자열 안에 작은따옴표나 큰따옴표 포함하기

- 문자열을 사용하다 보면 문자열 안에 작은따옴표나 큰따옴표를 넣어야 할 경우가 생김
- 작은따옴표와 큰따옴표를 사용하는 규칙이 달라짐
- 문자열 안에 '(작은따옴표)를 넣고 싶다면 문자열을 "(큰따옴표)로 묶어줌
- 문자열 안에 '를 그대로 사용할 수 있음

```
>>> s = "Python isn't difficult"
>>> s
"Python isn't difficult"
```

- 반대로 문자열 안에 "(큰따옴표)를 넣고 싶다면 문자열을 '(작은따옴표)로

```
>>> s = 'He said "Python is easy"'
>>> s
'He said "Python is easy"'
```

- 문자열 안에 작은따옴표나 큰따옴표 포함하기

- 문자열을 사용하다 보면 문자열 안에 작은따옴표나 큰따옴표를 넣어야 할 경우가 생김
- 작은따옴표와 큰따옴표를 사용하는 규칙이 달라짐
- 문자열 안에 '(작은따옴표)를 넣고 싶다면 문자열을 "(큰따옴표)로 묶어줌
- 문자열 안에 '를 그대로 사용할 수 있음

```
>>> s = "Python isn't difficult"
>>> s
"Python isn't difficult"
```

- 반대로 문자열 안에 "(큰따옴표)를 넣고 싶다면 문자열을 '(작은따옴표)로

```
>>> s = 'He said "Python is easy"'
>>> s
'He said "Python is easy"'
```

- 문자열 안에 작은따옴표나 큰따옴표 포함하기

- 작은따옴표 안에 작은따옴표를 넣거나 큰따옴표 안에 큰따옴표를 넣을 수는 없음

```
>>> s = 'Python isn't difficult'
SyntaxError: invalid syntax
>>> s = "He said "Python is easy""
SyntaxError: invalid syntax
```

- 실행을 해보면 구문 에러(SyntaxError)가 발생
- 여러 줄로 된 문자열은 작은따옴표 안에 작은따옴표와 큰따옴표를 둘 다 넣을 수 있음
- 큰따옴표 안에도 작은따옴표와 큰따옴표를 넣을 수 있음

- 문자열 안에 작은따옴표나 큰따옴표 포함하기

string_multiline_quote.py

```
single_quote = '''"안녕하세요."  
'파이썬'입니다.'''  
  
double_quote1 = """Hello"  
'Python'"""  
  
double_quote2 = """Hello, 'Python'"""    # 한 줄로 작성  
  
print(single_quote)  
print(double_quote1)  
print(double_quote2)
```

실행 결과

```
"안녕하세요."  
'파이썬'입니다.  
"Hello"  
'Python'  
Hello, 'Python'
```

- 문자열은 '(작은따옴표)로 묶어도 되고 "(큰따옴표)로 묶어도 된다는 점과
여러 줄로 된 문자열은 '''(작은따옴표 3개) 또는 """(큰따옴표 3개)를 사용

- cf) 문자열에 따옴표를 포함하는 다른 방법

- 작은따옴표 안에 작은따옴표를 넣을 수는 있는 방법
- 작은따옴표 앞에 w(역슬래시)를 붙이면 됨

```
>>> 'Python isn\'t difficult'
"Python isn't difficult"
```

- 큰따옴표도 "He said w"Python is easyw"처럼 큰따옴표 앞에 w를 붙이면 됨
- 문자열 안에 ' , " 등의 특수 문자를 포함하기 위해 앞에 w를 붙이는 방법을 이스케이프(escape)라고 부름

- cf) 따옴표 세 개로 묶지 않고 여러 줄로 된 문자열 사용하기
 - 문자열 안에 개행 문자(\n)을 넣으면 따옴표 세 개로 묶지 않아도 여러 줄로 된 문자열을 사용할 수 있음

```
>>> print('Hello\nPython')
Hello
Python
```

- 따옴표 세 개로 묶어서 여러 줄로 된 문자열을 만들면 줄바꿈이 되는 부분에 \n이 들어있음
- print 없이 출력해보면 \n이 그대로 나옴

```
>>> '''Hello
Python'''
'Hello\nPython'
```

- cf) 파이썬 셸과 스크립트 파일의 결과가 다른데요?

- 파이썬 셸의 >>>에서 문자열을 그대로 출력하면 작은따옴표도 함께 출력(변수에 넣은 뒤 변수로 출력해도 마찬가지)

```
>>> 'Hello, world!'  
'Hello, world!'
```

- 파이썬 셸에서는 문자열이나 변수를 그대로 입력하면 출력 결과가 문자열이라는 것을 정확하게 표현하기 위해 작은따옴표로 묶인 문자열이 출력
- 스크립트 파일에서는 문자열이나 변수만으로 출력을 할 수 없으므로 print를 사용

hello.py

```
print('Hello, world!')
```

실행 결과

```
Hello, world!
```

```
>>> print('Hello, world!')  
Hello, world!
```

- **불과 비교, 논리 연산자 알아보기**

- 참(True), 거짓(False)을 나타내는 불(boolean)을 알아보자
- 두 값의 관계를 판단하는 비교 연산자와 두 값의 논리값을 판단하는 `논리 연산자도 함께 알아보자
- 비교, 논리 연산자는 프로그래밍에서 매우 광범위하게 쓰임
- if, while 구문을 작성할 때 비교, 논리 연산자를 자주 사용함

- **불과 비교 연산자 사용하기**

- 불은 True, False로 표현하며 1, 3.6, 'Python'처럼 값의 일종

```
>>> True
True
>>> False
False
```

- **비교 연산자의 판단 결과**

- 파이썬에서는 비교 연산자와 논리 연산자의 판단 결과로 True, False를 사용
- 비교 결과가 맞으면 True, 아니면 False

```
>>> 3 > 1
True
```

- 부등호 >로 두 숫자를 비교함
- 3이 1보다 크니까 결과는 참이고 True가 나옴

- 숫자가 같은지 다른지 비교하기

- 이제 두 숫자가 같은 지 또는 다른 지 비교해보자
- 두 숫자가 같은 지 비교할 때는 `==(equal)`,
다른 지 비교할 때는 `!=(not equal)`을 사용

```
>>> 10 == 10    # 10과 10이 같은지 비교
True
>>> 10 != 5     # 10과 5가 다른지 비교
True
```

- 10과 10은 같으므로 True, 10과 5는 다르므로 True가 나옴
- 파이썬에서 두 값이 같은 지 비교할 때는 `=`이 아닌 `==`을 사용
- `=`은 할당 연산자로 이미 사용되고 있기 때문임

- 문자열이 같은 지 다른 지 비교하기
 - 숫자뿐만 아니라 문자열도 ==와 != 연산자로 비교할 수 있음
 - 문자열은 비교할 때 대소문자를 구분
 - 단어가 같아도 대소문자가 다르면 다른 문자열로 판단

```
>>> 'Python' == 'Python'
True
>>> 'Python' == 'python'
False
>>> 'Python' != 'python'
True
```

- 부등호 사용하기

- 부등호는 수학 시간에 배운 내용과 같음
- 큰지, 작은지, 크거나 같은지, 작거나 같은지를 판단

```
>>> 10 > 20    # 10이 20보다 큰지 비교
False
>>> 10 < 20    # 10이 20보다 작은지 비교
True
>>> 10 >= 10   # 10이 10보다 크거나 같은지 비교
True
>>> 10 <= 10   # 10이 10보다 작거나 같은지 비교
True
```

- 비교 기준은 첫 번째 값
- 첫 번째 값보다 큰지, 작은지 읽음
- 부등호를 말로 설명할 때 >은 초과, <은 미만, >=은 이상, <=은 이하라고도 함
- >, <은 비교할 값과 같으면 무조건 거짓
- >=, <=은 비교할 값과 같으면 참
- 이상, 이하는 비교할 값도 포함된다는 점이 중요함

- 객체가 같은지 다른지 비교하기

- 이번에는 is와 is not
- 같다는 ==, 다르다는 !=이 이미 있는데 왜 is, is not을 만들었을까?
- is, is not도 같다, 다르다지만 ==, !=는 값 자체를 비교하고, is, is not은 객체(object)를 비교

```
>>> 1 == 1.0
True
>>> 1 is 1.0
False
>>> 1 is not 1.0
True
```

- 1과 1.0은 정수와 실수라는 차이점이 있지만 값은 같음
- ==로 비교해보면 True가 나옴
- 1과 1.0을 is로 비교해보면 False가 나옴
- 1은 정수 객체, 1.0은 실수 객체이므로 두 객체는 서로 다르기 때문임

- cf) 정수 객체와 실수 객체가 서로 다른 것은 어떻게 확인하나요?
 - 정수 객체와 실수 객체가 서로 다른지 확인하려면 id 함수를 사용하면 됨
 - id는 객체의 고유한 값(메모리 주소)을 구함
(이 값은 파이썬을 실행하는 동안에는 계속 유지되며 다시 실행하면 달라짐)

```
>>> id(1)
1714767504
>>> id(1.0)
55320032
```

- 두 객체의 고유한 값이 다르므로 서로 다른 객체임
- 1과 1.0을 is로 비교하면 False가 나옴
- is, is not : 클래스로 객체를 만든 뒤에 객체가 서로 같은지 비교할 때 주로 사용
- 여기에 나오는 객체의 고유한 값(메모리 주소)에 대해서는 신경 쓸 필요 없음
- ==, !=와 is, is not의 동작 방식이 다르다는 정도만 알아 두면 됨

- cf) 값 비교에 is를 쓰지 않기
 - 값을 비교할 때는 is를 사용하면 안 됨
 - 변수 a에 -5를 할당한 뒤 a is -5를 실행하면 True가 나오지만 다시 -6을 할당한 뒤 a is -6을 실행하면 False가 나옴

```
>>> a = -5
>>> a is -5
True
>>> a = -6
>>> a is -6
False
```

- 변수 a가 있는 상태에서 다른 값을 할당하면 메모리 주소가 달라질 수 있기 때문
- 다른 객체가 되므로 값이 같더라도 is로 비교하면 False가 나옴
- 값(숫자)를 비교할 때는 is가 아닌 비교 연산자를 사용

- 논리 연산자 사용하기

- a and b

```
>>> True and True
True
>>> True and False
False
>>> False and True
False
>>> False and False
False
>>>
```

- and는 두 값이 모두 True라야 True임
 - 하나라도 False이면 False가 나옴

- 논리 연산자 사용하기

- a or b

```
>>> True or True
True
>>> True or False
True
>>> False or True
True
>>> False or False
False
```

- or는 두 값 중 하나라도 True이면 True임
 - 두 값이 모두 False라야 False가 됨

- 논리 연산자 사용하기

- not x

```
>>> not True
False
>>> not False
True
```

- not은 논릿값을 뒤집음
 - not True는 False가 되고, not False는 True가 됨
 - and, or, not 논리 연산자가 식 하나에 들어있으면 not, and, or 순으로 판단함

```
>>> not True and False or not False
True
```

- 논리 연산자 사용하기

- not True와 not False를 판단하여 False and False or True가 됨
- False and False를 판단하여 False가 나와서 False or True가 되므로 최종 결과는 True가 됨

```
not True and False or not False
False and False or True
False or True
True
```

- 이 식을 괄호로 표현하면 다음과 같은 모양이 됨

```
>>> ((not True) and False) or (not False)
True
```

- 순서가 헷갈릴 때는 괄호로 판단 순서를 명확히 나타내 주는 것이 좋음

- 논리 연산자와 비교 연산자를 함께 사용하기

- 조금 응용해서 논리 연산자와 비교 연산자를 함께 사용해보자

```
>>> 10 == 10 and 10 != 5    # True and True
True
>>> 10 > 5 or 10 < 3        # True or False
True
>>> not 10 > 5              # not True
False
>>> not 1 is 1.0            # not False
True
```

- 비교 연산자로 비교한 결과를 논리 연산자로 다시 판단함
- 비교 연산자(is, is not, ==, !=, <, >, <=, >=)를 먼저 판단하고
논리 연산자(not, and, or)를 판단하게 됨
- 파이썬은 영어 문장과 흡사한 구조를 가지고 있어서 코드가 읽기 쉬운 것이 장점

- cf) 정수, 실수, 문자열을 불로 만들기
 - 정수, 실수, 문자열을 불로 만들 때는 bool을 사용하면 됨
 - 정수 1은 True, 0은 False임
 - 만약 문자열의 내용이 'False'라도 불로 만들면 True임
 - 문자열의 내용 자체는 판단하지 않으며 값이 있으면 True임

bool(값)

```
>>> bool(1)
True
>>> bool(0)
False
>>> bool(1.5)
True
>>> bool('False')
True
```

- 즉, 정수 0, 실수 0.0이외의 모든 숫자는 True임
- 빈 문자열 "", ""를 제외한 모든 문자열은 True임

- cf) 단락 평가

- 논리 연산에서 중요한 부분이 단락 평가(short-circuit evaluation)임
- 단락 평가는 첫 번째 값만으로 결과가 확실할 때 두 번째 값은 확인(평가)하지 않는 방법을 말함
- and 연산자는 두 값이 모두 참이라야 참이므로 첫 번째 값이 거짓이면 두 번째 값은 확인하지 않고 바로 거짓으로 결정함

```
# 첫 번째 값이 거짓이므로 두 번째 값은 확인하지 않고 거짓으로 결정
print(False and True)    # False
print(False and False)   # False
```

- or 연산자는 두 값 중 하나만 참이라도 참이므로 첫 번째 값이 참이면 두 번째 값

```
# 첫 번째 값이 참이므로 두 번째 값은 확인하지 않고 참으로 결정
print(True or True)      # True
print(True or False)     # True
```

- cf) 단락 평가 (cont')

- 파이썬에서 논리 연산자는 이 단락 평가에 따라 반환하는 값이 결정됨
- True, False를 논리 연산자로 확인하면 True, False가 나왔는데, True and 'Python'의 결과는 무엇이 나올까?

```
>>> True and 'Python'  
'Python'
```

- 문자열 'Python'도 불로 따지면 True라서 True and True가 되어 True가 나올 것 같지만 'Python'이 나옴
- 파이썬에서 논리 연산자는 마지막으로 단락 평가를 실시한 값을 그대로 반환하기 때문임
- 논리 연산자는 무조건 불을 반환하지 않음
- 다음과 같이 마지막으로 단락 평가를 실시한 값이 불이면 불을 반환하게 됨

```
>>> 'Python' and True  
True  
>>> 'Python' and False  
False
```

- cf) 단락 평가 (cont')

- 문자열 'Python'을 True로 쳐서 and 연산자가 두 번째 값까지 확인하므로 두 번째 값이 반환됨
- and 연산자 앞에 False나 False로 치는 값이 와서 첫 번째 값 만으로 결과가 결정나는 경우에는 첫 번째 값이 반환됨

```
>>> False and 'Python'
False
>>> 0 and 'Python'    # 0은 False이므로 and 연산자는 두 번째 값을 평가하지 않음
0
```

- or 연산자도 마찬가지로 마지막으로 단락 평가를 실시한 값이 반환됨
- or 연산자에서 첫 번째 값만으로 결과가 결정되므로 첫 번째 값이 반환됨

```
>>> True or 'Python'
True
>>> 'Python' or True
'Python'
```


- cf) 단락 평가 (cont')
 - 두 번째 값까지 판단해야 한다면 두 번째 값이 반환됨

```
>>> False or 'Python'
'Python'
>>> 0 or False
False
```