Python 및 정보과학 입문 교재

Python 3.7 및 기본적인 알고리즘에 대한 이해

교재 제작 v3.1

이재호 2019년 6월 28일

차 례

1 Python 소개 및 기본 요소

1.1 강의 계획

본 교재는 Python 3.7의 기본적인 문법을 익힐 수 있는 내용과 예제들이 수록되어 있습니다. 난이도는 프로그래밍 언어를 처음 접하더라도 익힐 수 있도록 구성되어 있으며, 상황에 맞추어 유동적으로 진행할 예정입니다. 본 수업은 algorithm알고리즘[1.1]을 배우는 것보다는 Python 3 언어의 기본적인 문법 숙지와 더불어 자주 사용되는 패턴에 익숙해지는 것에 초점을 맞추고 있습니다. 또한, 정보과학적인 사고(순차적으로 논리적인 비약이 없이 문제 해결을 할 수 있는 능력)를 배우고 간단한 알고리즈믹 문제의 해결법을 배우는 것이 목적입니다. 아래의 진도표는 회차 당 세 시간 수업을 기준으로 한 것으로, 가변적일 수 있습니다.

1회차 Python 소개 및 기본 요소

2회차 Functions함수와 Conditionals조건문

3회차 Boolean Functions불리언 함수와 Loops반복문 기본

4회차 Lists리스트, Strings문자열, Counters카운터

5회차 Quantifiers한정자와 While 문

6회차 Loops반복문 응용과 파일 입출력

~~7회차~~ ~~Toy Robot~~

7회차 재귀법Recursion과 Python의 다양한 객체

8회차 람다Lambda 함수

기본적으로 알고리즘보다는 언어 자체의 문법과 활용에 초점을 맞춘 커리큘럼이므로, 알고리즘을 본격적으로 다루기 위해서는 자료 구조와 알고리즘에 대해서 깊게 다루는 서적을 읽어보시는 것을 추천드립니다. Python과 같이 언어를 익히고 난 다음에는 알고리즘을 배우고 실제로 구현할 수 있는 준비가 되어 있을 것입니다.

알고리즘에 대해서 더 배우고 싶으시다면 흔히 CLRS라고 불리우는 *Introduction to Algorithms 3/e* (*MIT Press*, *2009*)를 추천드립니다. 혹시 더 깊은 이해를 원하신다면, 저도 아직 읽어보지는 못했지만 커누스 교수의 TAOCP로 불리우는 *The Art of Computer Programming* 시리즈[1.2]를 읽어보시면 됩니다.

[1.1] 간단히 말해서, 어떤 값들을 입력받아 값들을 출력하는 잘 정의된 과정을 말합니다.

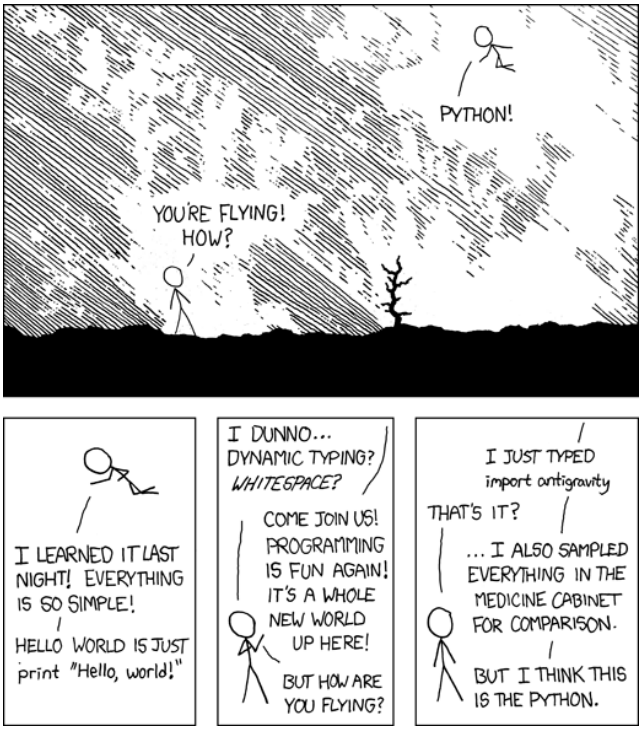
[1.2] 커누스 교수가 1968년부터 집필을 시작해 현재는 4권의 일부분까지 완성되어 있으며, 현재 7권까지 계획이 되어 있습니다. 빌 게이츠는 본인이 훌륭한 프로그래머라고 생각하면 TAOCP를 읽어보고, 다 읽을 수 있다면 자신에게 이력서를 보내라고 하였습니다. 여담으로, 커누스 교수는 이 책의 조판을 위해서 본 교재를 위해서 사용된 TEX을 개발하였습니다.

1.2 들어가기 전에

Python은 문법이 단순하면서도 활용 가능성이 무궁무진한 언어입니다. 단순히 '정보과학'만을 위해서 Python을 배운다기보다는, 평생 활용할 수 있는 도구를 배우는 것입니다. Python은 R이나 Matlab 등과 함께 학문적인 용도로도 쓰임이 많은 언어입니다. 특히 Matlab과 다르게 Python은 오픈 소스에 무료인데다가, 단순히 데이터 처리만을 위한 언어가 아닙니다. NumPy와 Matplotlib 등의 패키지를 활용한다면 Matlab이나 Mathematica와 같은 상용 프로그래밍 언어가 할 수 있는 다양한 작업들을 대등하게 할 수 있기도 합니다.

실제로 저는 학업 중 물리학, 천문학, 화학, 생물학 등의 분야에서 데이터 분석 및 차트 제작을 위해 Python을 활용하거나, Raspberry Pi와 연동하여 다양한 센서를 사용해 프로젝트를 진행하는데에도 사용했습니다. 또한 Django 등의 프레임워크를 사용한다면 웹서버를 구축할 수도 있는데, YouTube, Dropbox, Facebook, Netflix, Google, Instagram, Spotify 등의 유명 사이트들이 Python을 활용하여 서비스를 제공하고 있습니다. 이처럼 Python은 배워두면 무궁무진한 방면에서 활용할 수 있는 가능성을 가지고 있는 언어입니다.

1.3 Python이란?



흔히 Python을 "batteries included"(배터리가 들어간) 언어라고 합니다.

Python은 1991년에 Guido van Rossum이 발표한 프로그래밍 언어로, 문법이 굉장히 쉬우면서도 높은 생산성을 가지고 있는 언어입니다.[1,3] 특히 pseudocode의사코드와 유사하기 때문에 비교적 배우기 쉬워, 많은 학교에서 프로그래밍 입문 수업 언어로 Python을 채택하고 있습니다.

[1,3]그가 1989년 크리스마스 연휴에 취미로 Python을 제작하였던 것이 그 시작입니다.

또, **프로그래밍 언어로 할 수 있을 법한 거의 모든 기능들이 이미 Python을 위한 패키지로 구현되어 있습니다.** 이 때문에 Python을 흔히 "batteries included"라고 부릅니다. 따라서 속도가 중요한 작업이 아니라면 C/C++보다 Python을 쓰는 것이 효율적입니다. (연산이 아니라 업무의 효율을 말하는 것입니다.)

프로그래밍에 익숙치 않더라도 컴퓨터에 관심이 많다면 C, C++, Java 등의 언어와 함께 Python도 한 번 쯤 들어보았을 정도로, Python은 점유율 상위 다섯 언어 안에 드는 주류 언어입니다. Python은 1995년에 등장한 Java보다도 오래 전에 만들어진 언어로, 긴 역사를 가지고 있습니다. 그 만큼 버전의 숫자도 큰데, 이 글을 작성하는 현 시점에서 Python 2의 최신 버전은 2.7.16, Python 3의 최신 버전은 3.7.3입니다. 한동안은 Python 2와 Python 3가 동시에 개발되기도 하였고, Python 3보다는 Python 2를 사용하는 것이 낫다고 말하던 때가 있었습니다.[1.4] 그러나 현재는 Python 3를 배우고 사용하는 것이 권장됩니다. Python 2는 2.7이 최종 버전이고, 이후에는 bugfix 릴리즈만 있을 예정인데다가 2020년까지만 지원을 할 예정입니다. 나아가 최신 버전의 Ubuntu(Linux 배포판의 한 버전)는 Python 3를 기본 Python 버전으로 설정하고 있습니다.

Python은 interpreter인터프리터 언어입니다. Interpreter 언어란 소스 코드를 한 줄씩 기계어[1.5]로 번역하는 방식의 언어입니다. 조금 더 가독성을 높인, 기계어와 일대일 대응이 되는 (마찬가지로 저급 언어인) 어셈블리어가 있기도 합니다. 고급 언어를 기계어로 변환시키는 방법은 크게 두 가지가 있습니다. Compiler컴파일러와 interpreter입니다. C와 같은 언어는 compiler 언어로, 코드 전체를 한꺼번에 기계어로 변환시킵니다. 이 때문에 실행 속도는 빠르다는 장점이 있지만, 코드를 수정하기 위해서는 다시 한 번 전체를 기계어로 변환시키는 과정이 필요합니다. 반면 Python은 한 줄씩 기계어로 번역하기 때문에 실행 속도는 다소 느리지만, debug디버그[1.6]에 유리합니다. 한 줄씩 실행하기 때문에 애초에 compile을 거부하는 compiler 언어와는 다르게 버그가 있는 해당 줄까지 코드를 실행시켜주기 때문입니다. 이러한 장단점 때문에 프로그램의 골격을 Python으로 만들고, 빠른 연산이 필요한 부분만 C로 만드는 것도 가능합니다.

마지막으로, Python이 얼마나 쉽고 직관적인 언어인지 알아봅시다. 만약 A+가 F, C+, B0, A-, A+에 있으면 "A+가 있습니다."를 실행해주는 Python 코드를 봅시다.

```
if "A+" in ["F", "C+", "B0", "A-", "A+"]: print("A+가 있습니다.")
```

프로그래밍을 할 줄 모르더라도 저 코드를 이해할 수 있을 것입니다. 이처럼 Python은 인간이 사고하는 방식을 그대로 옮겨 놓았다고 해도 과언이 아닐만큼이나 직관적입니다. 익숙해진다면 자신이 하고 싶은 일을 코드로 옮기려고 끙끙댈 필요 없이, 생각하는 그대로 코드를 작성할 수 있을 것입니다. 참고로, 이와 같은 일을 하려면 C언어에서는 다음과 같이 해야 합니다.

```
1 #include <stdio.h>
2 #include <string.h>
```

[1.4] 1, 2년 전까지만해도 Python 2 vs Python 3의 논쟁이 비일비재하게 일어났었습니다.

[1.5] 기계가 바로 이해할 수 있는 저급 언어로, 0과 1의 이진수로만 구성되어 있는 언어로, 모든 CPU를 구동시키기 위해서는 C와 같은 고급 언어를 저급 언어로 변환시키는 과정이 필요합니다.

[1.6] 코드에서 bug버그, 즉 오류를 제거하는 것을 의미합니다.

```c
int main(void) {
    char grades[][3] = {"F", "C+", "B0", "A-", "A+"};
    for (int i = 0; i < 5; ++i) {
        if (strncmp(grades[i], "A+", 2) == 0) {
            printf("A+가 있습니다.\n");
        }
    }
    return 0;
}
```

단 한 줄의 Python 코드로 될 일을 C 언어로는 10줄 이상으로 작성해야 하는 것입니다.

1.4 Python의 기본 요소

Hello, World

```
#include <stdio.h>                    include information about standard library

main()                                define a function named main
                                      that receives no argument values
{                                     statements of main are enclosed in braces
    printf("hello, world\n");         main calls library function printf
}                                     to print this sequence of characters;
                                      \n represents the newline character
```

The first C program.

K&R *The C Programming Language*의 첫 프로그램

1978년에 씌여진 K&R이라고 불리우는 *The C Programming Language*에서 예시로 사용된 이후 첫 프로그램은 보통 `Hello, World!`를 출력하는 것으로 시작됩니다. 먼저, IDE에 다음과 같은 코드를 입력해봅시다:

```python
print("Hello, World!")
```

실행시켜보면 `Hello, World!`가 출력될 것입니다. C 코드에서는 `include` 등 이것저것 따로 입력해야 할 것이 있는데, Python에서는 사뭇 간단한 것을 볼 수 있습니다. 이제 Python의 문법에 대해서 본격적으로 알아봅시다.

Values값과 Variables변수

윈도우를 쓴다면 명령 프롬프트나 powershell, 맥이나 리눅스와 같은 유닉스 계열을 쓴다면 터미널, 혹은 그냥 IDE에 내장된 쉘shell에 다음을 입력해 봅시다. Interpreter 언어이기 때문에,

명령을 한 줄씩 입력하여 명령을 수행할 수 있습니다. (>>>는 직접 입력하는 것이 아닙니다.)

```
1 >>> a = (1+2+3+4)/2
2 >>> b = a - 1
3 >>> c = 3
4 >>> print(b + c)
5 7
6 >>> print(b, c)
7 4  3
8 >>> d = c
9 >>> a = d
10 >>> print(a)
11 3
```

`a = (1+2+3+4)/2`는 등호 왼쪽에 있는 `a`에 등호 오른쪽에 있는 $\frac{1+2+3+4}{2}$를 배정한다는 뜻입니다. 즉, 수학에서 말하는 등호와는 의미가 다른 것이지요. 줄 2의 `b = a - 1`은 현재 `a`에 배정된 $\frac{1+2+3+4}{2} = 5$의 값에 1을 뺀 4를 `b`에 배정한다는 뜻입니다. 이 때, 좌변에 있는 `a`, `b` , `c`를 variables변수[1.7], 우측의 `(1+2+3+4)/2`를 expression표현이라고 합니다. 또한 이러한 변수에 배정되는 것을 value값이라고 합니다. 다시 위의 예시로 돌아가서, 줄 2에 있는 좌변의 `b`는 variable, 우변의 `a - 1`는 value인 것이죠. 영어 문장으로 "Let {variable} be {value}."가 말이 되는지 대입하여 보면 쉽게 알 수 있습니다. Variable과 value는 문자인지 숫자인지의 여부가 아니라 대입이 되는 대상인지 대입이 되어지는 대상인지의 여부가 결정짓습니다. 물론, variable이 value 그 자체가 될 수 있습니다. 줄 8의 예시와 같은 경우, 우변의 `c`는 variable이면서 `d`라는 variable에 대입되는 value입니다. 그렇다면, 아래와 같은 표현을 어떨까요?

```
>>> 10 = a
```

지금까지 잘 따라왔다면, value가 위치해야 할 좌변에 value인 literal[1.8]이 위치해 잘못된 syntax 구문이라는 것을 알 수 있습니다. `SyntaxError: can't assign to literal`이라는 오류 log를 볼 수 있을 것입니다. (여담이지만, 앞으로 수 없이 많은 오류 log를 볼 텐데, 이를 꼼꼼히 읽어보는 습관을 들입시다. 오류 log를 무시하고 디버깅하는 것은 마치 백사장에서 바늘을 찾는 것과도 같습니다.) 혹은, "let 10 be a"라는 표현이 말이 되지 않는다는 것을 통해서도 직관적으로 잘못되었음을 파악할 수 있습니다.

[1.7] 필요에 따라 영문의 'variable', 국문의 '변수'를 사용하여 서술하겠습니다. 이외의 용어도 처음에는 영문과 국문을 병기한 후, 필요에 따라 둘 중 하나의 표현만 사용하겠습니다.

[1.8] 어떤 객체에 value를 부여할 수 있는 것입니다. `0`은 `int` literal, `"python"`은 `str` literal입니다. `int`, `str`이 무엇인지는 조금 뒤에 type형이라는 것을 배우면 알 수 있습니다.

이러한 변수는 여러 가지 종류가 있습니다. 이를 data type자료형, 혹은 간단히 type형이라고 합니다. `-1`, `0`, `76` 등의 값은 `int`[1.9] type, `3.14159`, `0.`, `6.626e-34`, `6.022E23` 등의 값은 `float`[1.10] type, `"Hello, world!"`, `'python'`, `""` 등의 값은 `str`[1.11] type입니다. Type에 대해서는 조금 뒤에 더 자세히 살펴봅시다.

변수는 어떤 값이 배정된 것이라고 했는데, 이것을 assign되었다고 하며 값이 written되었다고도 표현할 수 있습니다. 이렇게 변수에 저장된 값은 read읽을 수 있는데, 첫 예시의 줄 2처럼 `a`에 저장된 값 `5`를 불러오는 것이 이에 해당합니다. 또한 줄 4의 `print`를 통해서도 값을 읽어올 수 있습니다.

변수는 서로 다른 값이 배정될 수 있습니다. 첫 예시에서 줄 9를 보면, `5`가 저장되었던 `a`에 `3`이 저장된 `d`의 value가 다시 `a`에 쓰여지는 것을 볼 수 있습니다. 줄 10에서 이 사실을 재확인할 수 있고요. 이와 같이 변수는 재활용될 수 있고, 이는 개수를 세거나(counting) 특정 사건을 기록하기 위해 flag 등으로 사용하는데 도움이 됩니다.

```
1 >>> summ = 0
2 >>> summ = summ + 1
3 >>> summ = summ + 1
4 >>> print(summ)
5 2
```

더 자세한 활용은 차차 Python을 익혀가면서 알아봅시다.

이 뿐만이 아니라, Python은 여러 변수에 여러 값을 한 번에 배정하는 것(multiple assignment)을 허용합니다. 나아가 **변수의 swapping을 매우 손쉽게 할 수 있습니다.** 이 둘을 다음 예시를 통해 함께 확인합시다.

```
1 >>> a, b, c = 2, 7, 12
2 >>> a, b, c = b, c, a
3 >>> print(a, b, c)
4 7 12 2
5 >>> a, b = b % a, a
6 >>> print(a, b)
7 5 7
```

Multiple assignment를 허용하지 않는 대다수의 언어에서는 임시 변수를 도입해야 합니다. 예컨대 줄 5의 경우 아래와 같은 방법을 사용해야 합니다.

[1.9] 정수라는 뜻의 integer에서 따온 명칭입니다. 단, 수학에서 3.0은 정수이지만 `3.0`은 `float` type입니다.

[1.10] 부동 소수점 혹은 떠돌이 소수점이라는 뜻의 floating point에서 따온 명칭입니다. 실수가 아니라 근삿값이라는 의미에 더 가깝습니다.

[1.11] 나열이라는 뜻의 string에서 따온 명칭입니다.

```
>>> tmp = a
>>> a = b % tmp
>>> b = tmp
```

Python에서는 마치 하노이의 탑을 연상시키는 이러한 과정을 시행하지 않아도 됩니다.

마지막으로, 변수의 이름으로 정할 수 없는 특정 문자열이 있습니다. Python이 내부적으로 사용하는 `int`, `str`, `if`, `else`, `for`, `range`, …등이 이에 해당합니다.[1.12] 그리고 변수명은 영어 대소문자, 숫자, 그리고 _로만 이뤄져 있어야 합니다. (Python 3부터는 한글로도 이름을 명명할 수 있습니다.) 나아가 숫자로 시작할 수 없는데, `1st_name`과 같은 문자열을 변수명으로 지정할 수 없는 것입니다.

Expressions표현

Expression은 variable, value, 그리고 operator들의 조합입니다. 우리가 흔히 부르는 사칙 연산 `+`, `-`, `*`, `/`와, 나머지를 구해주는 `%`, 지수를 뜻하는 `**` 등이 이에 해당합니다. **주의할 점은, `^`이 지수를 뜻하는 것이 아니라 `**`이라 것입니다.** 또한 `//`는 몫을 구해줍니다. 아래의 예시를 봅시다.

```
>>> 12 + 5
17
>>> 12 - 5
7
>>> 12 * 5
60
>>> 12 / 5
2.4
>>> 12 // 5
2
>>> 12 % 5
2
>>> 12 ** 5
248832
>>> 12 ^ 5
9
```

`^`는 bitwise XOR의 연산자로서, $12 = 1100_2$, $5 = (0)101_2$이므로 digit이 다른 $2^3, 2^2, 2^1$ 자릿수만 1을 취한 $1001_2 = 9$가 `12 ^ 5`의 값입니다. `^`은 지수의 연산이 아닙니다! 연산의 순서는 기본적으로 괄호(`()`), unary 연산(`+x`, `-x`), 지수(`**`), 곱셈/나눗셈/나머지 연산(`*`, `/`, `%`), 덧셈/뺄셈(`+`,

[1.12] `int` 등 type 명은 배정이 가능하지만, 형 변환을 위한 함수로 사용되므로 사용하면 안됩니다. 위에 나왔던 코드 중 `summ`이라고 썼던 변수명도 `sum`으로 쓰지 않은 이유는 `sum`에 해당하는 내장 함수가 있기 때문입니다.

-)의 순서입니다. 헷갈리는 경우나 혼동을 불러올 수 있는 경우에는 ()를 사용하여 순서를 명시할 수 있습니다. 또는 논리적인 블럭이 되는 경우 괄호를 사용하여 묶어주는 것이 권장됩니다.

참고로, Python 2에서는 나눗셈을 할 때 정수끼리 행하면 몫만이 구해집니다. `12 / 5 = 2`와 같이 말입니다. 반면 `12.0 / 5 = 2.0`와 같이 제수나 피제수 중 하나라도 `float` 형이면 결과도 실제 `float`의 나눗셈의 결과로 나옵니다. 위와 같은 결과를 인터넷이나 서적에서 보신다면 Python 2 코드이므로 유의하시기 바랍니다.[1.13]

코드를 작성할 때에는 특별한 경우를 제외하고는 가독성이 중요합니다. 예컨대, 중복되는 값이나 의미가 있는 값은 특정 변수에 저장하여 해당 변수를 통해 식을 표현하는 것이 바람직합니다. 아래의 예시를 봅시다.

```
>>> S = ((3 + 4 + 5) * (-3 + 4 + 5) * (3 - 4 + 5) * (3 + 4 - 5))**0.5
>>> a, b, c = 3, 4, 5
>>> s = (a + b + c) / 2
>>> S = (s * (s - a) * (s - b) * (s - c))**0.5
```

비록 줄 수는 늘어났지만, 줄 1의 표현보다는 줄 4의 표현이 가독성이 높을 뿐만이 아니라 더 일반적이어서 값을 바꾸기 위해서는 줄 2의 숫자 부분만 변경을 하면 된다. 반면 줄 1의 표현의 경우 +와 -의 부호 구분에서 실수를 할 수 있고 다른 값을 대입하기 위해서는 12 부분에 수정을 가해야 한다.

마지막으로 소개할 문법은 위에서 잠시 언급한 개수 세기 등에서 유용하게 쓸 수 있습니다. 변수 뒤에 산술 연산자(+, -, *, /, %, **) 뒤에 바로 =를 붙인 후 수를 쓰는 syntax입니다.[1.14] `x += 1`과 같이 말입니다. 이는 해당 변수에 저장된 값에 등호 뒤에 쓰인 값을 더한다는 의미로, `x = x + 1`과 동일한 의미를 가지고 있습니다. 아래와 같이 응용할 수 있습니다.

```
>>> x = 4
>>> x += 2
>>> x
6
>>> x -= 1
>>> x
5
>>> x *= 2
>>> x
10
>>> x /= 5
>>> x
2
```

[1.13] 또한 이 교재도 원래 Python 2 기준으로 쓰여져 있었는데, 이와 관련해 미처 수정되지 않은 부분이 있다면 알려주시기 바랍니다.

[1.14] `int`나 `float`형에서는 모든 산술 연산자를 쓸 수 있고, `str`형에 대해서는 정의가 되어 있는 +만 사용할 수 있습니다.

```
>>> x %= 3
>>> x
2
>>> x **= 3
>>> x
8
```

참고로, C/C++이나 Java와 같은 언어에는 ++, --와 같이 쉽게 1을 더하거나 뺄 수 있는 연산자가 있습니다. a가 3의 값을 가지고 있었을 때 a++를 하면 4가 되는 것이지요. C++의 ++이 해당 연산자에서 따온 것입니다. 그러나 a++과 ++a에 따라서 연산 후 결과는 같지만 실제 코드에서 사용되는 위치에 따라 수행 결과가 달라지는 등 버그의 원인이 되기 때문에 Python이나 모던 언어에서는 해당 연산자를 문법에 넣지 않는 추세입니다.

Types형

위에서 간단히 소개한 바 있는데, variable의 종류를 type형이라고 합니다. 현재로서는 지금까지 언급한 int(정수형), float(실수형), str(문자열) 세 가지 type만 알아두시면 됩니다. 지금까지는 숫자가 실수형임을 명시할 때 3.과 같이 온점을 찍어 표현하였는데, type conversion형 변환이라는 것을 사용하여도 됩니다. **형 변환은 정수형과 실수형 간에서 자유롭게 가능하고, 문자열의 경우에는 그 자체가 수일 경우에만 변환이 가능합니다.**

```
>>> x = 76
>>> x
76
>>> float(x)
76.0
>>> str(x)
'76'
>>> pi = 3.14
>>> pi
3.14
>>> int(pi)
3
>>> str(pi)
'3.14'
>>> s = "1"
>>> s
'1'
>>> int(s)
1
```

```
>>> float(s)
1.0
```

위 예시를 통해 필요한 모든 경우를 파악하셨을 것입니다. 또한, type(·)를 통해 직접 type을 확인할 수 있습니다.

```
>>> print(type(76))
<class 'int'>
>>> print(type(76.))
<class 'float'>
>>> print(type("76"))
<class 'str'>
```

Input/Output입출력

지금까지는 Python shell에서만 명령을 실행했습니다. 그렇기 때문에 a에 담긴 값을 알기 위해서는 굳이 print(a)가 아니라 a를 치는 것 만으로도 충분했습니다. 하지만 여러 줄의 코드를 한꺼번에 작성하여 실행할 때에는 이런 방식의 접근이 불가능합니다. 또, 코드를 실행 중일 때 어떤 입력을 받기 위해서는 지금까지와는 다른 방법이 필요합니다. Shell과는 다르게 한 줄씩 직접 입력하는 방식이 아니기 때문입니다. 값을 출력하는 것은 지금까지 해왔던 것처럼 print를 사용하면 되는데, 아래에서 print에 대해 좀 더 알아보겠습니다. Shell이 아니라 파일을 만들어서 실행시킵니다.

```
today = "Monday"

print("Today is", today)
print("Today is " + today)

print("\nprintf style:")
print("Today is %s" % today)
print("Today is %(day)s" % {"day": today})

print("\nPython 3, back-ported to Python 2:")
print("Today is {}".format(today))
print("Today is {day}".format(day=today))

print("\nPython 3.6+, Formatted String Literals:")
print(f"Today is {today}")
```

위 예시는 Python에서 `Today is Monday`를 출력하는 여러가지 방법을 나열한 것입니다. 첫 번째(줄 3)는 `,`를 사용하여 `print` 함수 내의 여러 인자들을 출력하는 방식입니다. 자동으로 띄어쓰기가 들어간다는 것에 유의합니다. 두 번째(줄 4)는 문자열의 덧셈을 통해 출력한 것으로, 띄어쓰기는 직접 앞 `Today is`에 추가하였습니다. 다음 예시부터는 문자열 포맷팅에 관한 내용입니다. 먼저 줄 7과 8의 예시는 과거 사용되었던 방식으로, 현재에도 사용할 수 있는 방식입니다. C 언어의 `printf`와 유사한 방식입니다. `%s`는 뒤 `%` 뒤의 변수를 문자열 형식으로 넣으라는 뜻입니다. 만약 이름을 붙여서 지정하고 싶다면 줄 8과 같이 사용하면 됩니다. Python 3에서 시작되어 현재는 Python 2로 백포트된 문자열 포맷팅 방식은 줄 11과 12에 나와 있습니다. `{}`로 지정된 부분에 뒤 `.format()` 메소드 내부의 인자가 대입되는 것을 확인할 수 있습니다. 마지막 15 줄의 방식이 가장 최근에 도입된 Formatted String Literals라는 것입니다. 문자열 앞에 `f`를 붙인 후 단순히 중괄호 안에 원하는 변수 이름을 넣으면 대입이 됩니다. 만약 Python 3.6 이상 버전을 쓰고 있다면 사용이 가능하므로 되도록이면 해당 방식을 사용하는 것이 권장됩니다.

이제는 값을 입력하는 법에 대해 알아봅시다. `input(·)` 함수를 사용하면 됩니다. 다음과 같은 예시를 살펴봅시다.

```
1 today = input("What day is it today? ")
2 print("Today is", today)
3
4 s = input("Enter the number you want to know the square root of: ")
5 n = float(s)
6 print(n**.5)
```

위 코드를 실행시키면 창에 `What day is it today?` 가 출력된 후 입력이 될 때까지 기다립니다. 키보드로 값을 입력한 후–`Thursday`를 입력했다고 합시다–리턴 키를 치면 값이 입력되고, `Today is Thursday.`가 출력될 것입니다. 또한, 수를 입력 받을시 `int(·)`나 `float(·)`로 형 변환을 수행해줘야 합니다. **`input(·)`이 넘겨주는 값은 항상 `str` 형이기 때문입니다.**

1.5 예제

1. 1부터 n까지 자연수의 제곱의 합과 세제곱의 합의 차이를 구하는 코드를 작성하세요. 단, 아직 배우지 않은 `for` 문 없이 다음의 공식을 사용하세요:

$$\sum_{k=1}^{n} k^2 = \frac{n(n+1)(2n+1)}{6}$$
$$\sum_{k=1}^{n} k^3 = \left(\frac{n(n+1)}{2}\right)^2$$

```
1 n = int(input("Enter n: "))
2 # Add here!
```

2. 원탁 주위에 a, b, c, d, e가 앉아 있습니다. 각자는 자신이 좋아하는 정수를 종이에 적은 후, 양 옆에 앉은 사람의 정수를 더합니다. 최종적으로 각자가 얻게된 수를 출력하도록 다음 코드를 완성하세요:

```
a = int(input("Enter a: "))
b = int(input("Enter b: "))
c = int(input("Enter c: "))
d = int(input("Enter d: "))
e = int(input("Enter e: "))
print("Favorite integers: ", a, b, c, d, e)
# Add here!
print("Final integers: ", a, b, c, d, e)
```

3. 이차방정식 $ax^2 + bx + c = 0$의 해를 구하는 코드를 작성하세요. 단, 판별식 $b^2 - 4ac > 0$이라고 가정합니다.

```
a = int(input("Enter a: "))
b = int(input("Enter b: "))
c = int(input("Enter c: "))
# Add here!
# x1 = ...
# x2 = ...
print("Solutions for the quadratic equation are ", x1, " and ", x2)
```

4. 다음 표현의 값을 예상하세요.

```
>>> 3*2**6+12
???
>>> 32/7+2**3/3
???
>>> 13/3%3+2.0
???
>>> 13//2/4.
???
>>> int(2**(1/2))+1
???
```

5. Python 3에서 가능한 변수명을 모두 고르세요.

- `if`
- `sum`
- `max`
- `1st_var`
- `var_1`
- `this+that`
- `_self`
- **변수**
- `r&e`

6. 다음 코드는 여섯 개의 실수 `x1`, `x2`, `x3`, `y1`, `y2`, `y3`을 입력받습니다. `i`가 `1`, `2`, `3`을 취할 때 (`xi`, `yi`)가 좌표평면에서 서로 다른 세 점을 나타내는 좌표라고 가정합니다. 이때 세 점이 이루는 삼각형의 면적을 계산하는 코드를 작성하세요.
   참고로,
   - 세 변의 길이가 a, b, c인 삼각형의 면적은 $s = \frac{a+b+c}{2}$일 때 $\sqrt{s(s-a)(s-b)(s-c)}$입니다.
   - 또한, 어떤 두 벡터 $\mathbf{u}$와 $\mathbf{v}$가 이루는 삼각형의 면적은 $\frac{1}{2}\|\mathbf{u} \times \mathbf{v}\| = \frac{1}{2}\|\mathbf{u}\|\|\mathbf{v}\| \cos\theta$입니다.
   - 이때, $\cos\theta = \frac{\mathbf{u}\cdot\mathbf{v}}{\|\mathbf{v}\|\|\mathbf{u}\|}$ 입니다.

```python
x1 = int(input("Enter x1: "))
y1 = int(input("Enter y1: "))
x2 = int(input("Enter x2: "))
y2 = int(input("Enter y2: "))
x3 = int(input("Enter x3: "))
y3 = int(input("Enter y3: "))
# Add here!
# area = ...
print("Area of the triangle is", area)
```

7. `a`를 `b`로 나눈 나머지를 `%` 없이 사칙연산과 `//` 만으로 구하세요.

```python
a = int(input("Enter a:"))
b = int(input("Enter b:"))
# Add here!
# r = ...
print("Remainder is", r)
```

참고. 다음 코드는 양의 실수에 대해 올림 함수 $\lceil x \rceil = \min\{n \in \mathbb{Z} \mid n \geq x\}$를 계산합니다. `int(·)`, `+`, `-`만을 사용하여 구현하세요.[1.15]

- 힌트: `int(·)` 함수의 그래프를 그려보세요.

```
x = float(input("Enter x:"))
# Add here!
# ceil_x = ...
print("Ceil(x) =", ceil_x)
```

2 Functions함수와 Conditionals조건문

2.1 Functions함수

오늘날 프로그래밍의 기본 중 기본은 functions함수라고 말할 수 있습니다. 함수를 전혀 사용하지 않고 프로그래밍을 하는 것이 가능은 하지만, 함수를 쓰면 다양한 이점이 있습니다. 일단 프로그램을 역할에 따라 여러 부분으로 나누어 구조적 프로그래밍[2.1]이 가능해집니다. 또한 함수를 사용하면 같은 코드를 반복하지 않음으로써 프로그램의 용량을 줄이고 범용성을 늘릴 수 있습니다. 지난 시간에 자주 반복하여 사용하는 식을 하나의 변수에 대입하여 사용하면 효율적이라고 했던 것과 같은 맥락입니다. 나중에 객체 지향 프로그래밍 패러다임을 배우면 더 자세히 배우겠지만, 함수를 사용하여도 encapsulation캡슐화가 가능해집니다. 캡슐화란 함수를 한 번 구현해두면 내용을 모르더라도 그 기능을 사용할 수 있게 하는 것이라고 생각하면 됩니다.

한 번 Python에서 함수를 어떻게 정의하고 사용하는지, 아래의 예시를 통해서 살펴봅시다.

```
def gamma(v):
    c = 299792458
    b = v / c
    return 1 / (1 - b**2)**0.5

def t_rel(x, t, v):
    c = 299792458
    return gamma(v) * (t - v * x / c**2)

pos = 10
time = 13
```

[1.15] 정답: `int(x) - (int(x) - int(x) + 1)`

[2.1] 구조적 프로그래밍은 비구조적 프로그래밍과 대비되는 프로그래밍 패러다임입니다. 비구조적 프로그래밍은 프로그램 전체를 하나의 연속된 코드로 작성하는 방식입니다. 이러한 프로그래밍 방식에서는 GOTO문과 같은 제어문에 의존할 수 밖에 없는데, 이는 디버깅이 어렵고 가독성이 떨어집니다. 특히 구조화되지 않은 코드는 프로그램이 복잡해지면 수행 방향이 복잡하게 얽히게 되는데, 이를 비유적으로 '스파게티 코드'라고 표현하기도 합니다. machine language기계어와 assembly language어셈블리어를 제외한 현대의 대부분 프로그래밍 언어는 구조적 프로그래밍을 지원합니다.

```
12 vel = 3E7
13
14 print(gamma(vel))
15 print(t_rel(pos, time, vel))
```

일단 함수를 사용하기 위해서는 함수를 정의해야 합니다. 함수의 정의는 `def function(arg1, arg2, ...)`의 형태로 시작해서 `return value`으로 끝납니다. 이때 `function`은 원하는 함수명이고, `arg1`, `arg2`, $\cdots$은 함수의 인자에 해당하는 변수명입니다. 함수를 호출(사용)할 때에는 그냥 `function(a, b, c)`의 형태로 사용하시면 됩니다. 또한, 예시처럼 매개 변수의 수는 여러 개가 될 수 있습니다. 수학의 다변수 함수와 유사하게 받아들이면 됩니다. 본격적으로 함수를 어떻게 정의하고 사용해야 하는지 알아봅시다.

Indentation들여쓰기

Python에서는 indentation들여쓰기가 구문을 나누는 중요한 syntax입니다. C, C++ 등의 언어에서는 들여쓰기를 하지 않아도 중괄호`{.}`와 세미콜론 `;`으로 명령을 구분하고, 들여쓰기는 완전히 가독성만을 위한 것입니다. 그러나 중괄호나 세미콜론을 사용하지 않는 Python에서는 들여쓰기가 반드시 필요합니다. 띄여쓰기를 4회 하여 들여쓰기를 하던가, 탭을 통해서 들여쓰기를 할 수도 있습니다.[2.2] 띄어쓰기 4회를 통해 들여쓰기를 한다고 스페이스 키를 네 번 누를 필요는 없습니다. IDE나 텍스트 에디터 설정에서 탭 키를 soft tab (띄어쓰기 여러 개를 사용하여 들여쓰기를 하는 것)으로 설정하던지 hard tab (탭을 사용하여 들여쓰기를 하는 것)으로 설정하면 됩니다. Python의 코딩 스타일 가이드인 PEP-8에서는 soft tab이 권장됩니다.

위에서 언급하였듯이, Python에서 들여쓰기는 필수적인 문법입니다. 함수뿐만이 아니라 이번 시간에 배울 조건문, 나중에 배울 반복문 등에서 들여쓰기는 구문을 구별하고 포함 관계를 나타내는 역할을 합니다. 위의 예시처럼 함수에서는 `def` 다음 줄부터 `return`의 해당 줄까지 한 수준 들여쓰기를 해야 합니다. **들여쓰기를 실수로 의도한 바와 다르게 한다면 코드의 수행 결과가 완전히 달라질 수 있습니다. 나아가 문법적으로는 맞을 수 있어서 에러 메시지는 뜨지 않고 결과만 차이가 생기는 경우도 빈번하니 유의해야 합니다.** 따라서 들여쓰기를 명시적으로 나타내주는 IDE의 기능을 활용하면 생산성을 높일 수 있습니다. IDE의 기능에서 기본 중의 기본이니 대부분의 IDE에서 들여쓰기를 도와주는 기능이 있을 것입니다. Wing IDE에서는 Preferences > Editor > Indentation으로 들어가면 Show Indent Guides 항목에 체크를 하면 됩니다. PyCharm에서는 기본적으로 indentation guide를 보여줄 것입니다.

Built-in Functions내장 함수

사실 이번에 함수를 처음 접하는 것이 아닙니다. 지난 시간에도 여러 함수를 보았는데요, `type(·)`이나 형 변환을 위해 사용한 `int(·)` 등이 바로 함수입니다. 이러한 함수는 사용자가 직접 정의하지 않아도 사용할 수 있는 built-in functions내장 함수입니다. 또한 어떤 내장 함수들은 바로 사용할 수 있는 것이 아니라, '이러이러한 종류의 함수들을 사용할 것이다'라는 것을 코

[2.2] 띄어쓰기를 하여 들여쓰기를 할것인지, 탭을 하여 들여쓰기를 할 것인지는 프로그래머들 사이에서 펼쳐지는 오랜 논쟁입니다.

드에 작성하여야 쓸 수 있는 것들이 있습니다. 바로 `import package`을 (보통) 코드 맨 윗 줄에 적어주는 것입니다. 특히 수식 계산을 위해 자주 사용할 패키지는 `math` 패키지로, 제곱근 함수 `sqrt(·)`이라든지 사인 함수 `sin(·)`와 같은 삼각 함수, 로그 함수 `log(·)`을 사용하기 위해서는 `math` 패키지를 import해야 합니다. 지난 단원의 마지막 문제인 올림 함수는 `math` 패키지의 `ceil(·)`[2.3] 함수를 사용하면 됩니다. 마찬가지로 `int(·)`와 같은 '가짜' 내림 함수가 아니라 진짜 내림 함수인 `floor(·)`가 제공됩니다. Python 2에서는 `math` 패키지에서 제공되던 반올림 함수`round(·)`는 이제 `math` 없이도 제공됩니다. 아래에서 사용 예시를 봅시다.

```
>>> import math
>>> math.sqrt(4)
2.0
>>> math.sin(30 * math.pi / 180)
0.49999999999999994
>>> math.log(math.e)
1.0
>>> math.ceil(-1.5)
-1
>>> math.floor(-1.5)
-2
>>> round(-1.5)
-2
```

만약 `math.(·)`를 반복하는 것이 귀찮다면, `from 패키지 명 import *`로 import를 하면 아래와 같이 계속 `math.`를 앞에 붙일 필요가 없어집니다.

```
>>> from math import *
>>> sqrt(4)
2.0
>>> sin(30. * pi / 180.)
0.49999999999999994
>>> log(e)
1.0
```

단, `sin` 등의 함수를 사용자가 새로 정의하거나, `pi`나 `e`와 같은 값을 새로 지정한다면 값이 덮어씌워져서 실수할 가능성이 커집니다.

```
>>> from math import *
>>> e
```

[2.3] Ceiling function올림 함수의 앞에서 따온 명칭입니다.

```
2.718281828459045
>>> e = 1
>>> e
1
```

또한 같은 함수 명 fn을 지닌 두 개의 패키지 pack1과 pack2에서 from pack1 import *를 한 후 from pack2 import *를 한다면 뒤에서 불러온 pack2의 fn이 먼저 불러온 pack1의 fn을 덮어 씌울 것입니다. 예컨데 numpy라는 패키지를 컴퓨터에 설치한 후, from numpy import *와 from math import *를 한다면 상당수의 함수가 겹치게 되어 문제가 생길 가능성이 높습니다. 따라서 문제가 생기지 않을 것이라고 확신하거나 불가피할 경우에만 from import *를 사용하는 것이 권장됩니다. 나아가 자신이 math 패키지에서 sin(·) 함수만 필요하다는 경우에는 from math import sin을 하여 바로 math. 없이 sin 함수를 사용할 수 있습니다.[2.4]

User-Defined Functions사용자 정의 함수

본인이 원하는 함수가 없다면 직접 정의를 해서 함수를 만들 수 있습니다. 함수는 반환 값이 있는 것과 그렇지 않은 것 두 종류가 있습니다. 아래는 원의 반지름을 받아서 면적을 반환하는 함수 circ_area(·)입니다.

```
import math

def circ_area(radius):
    return math.pi * radius**2
```

return 다음에 반환할 값이 나타나 있습니다. 반면 아래는 원의 면적을 반환하지 않고 단순히 출력을 하는 함수 print_circ_area(·)입니다.

```
import math

def print_circ_area(radius):
    print(math.pi * radius**2)
```

circ_area(·)의 경우에는 원의 면적을 출력하기 위해 print를 사용해야 합니다. 그렇지만 다른 값을 계산하는 등의 작업에서 함수를 표현에 넣을 수 있습니다. print_circ_area(·)는 직접적으로 원의 면적을 출력하지만, 그 값을 다른 표현에서 활용할 수는 없습니다. 반환하는 값이 없기 때문입니다.

[2.4] 보통 import numpy as np로 numpy 패키지를 불러옵니다. 이러한 경우, np.sin과 같은 형식으로 패키지를 사용할 수 있습니다.

함수를 정의하여 사용할 때에는, 사용하기 위해 호출하기 이전에 정의를 하기만 하면 됩니다. 위 조건만 만족한다면 다른 코드들 사이에 끼워두어도 되고, 함수들 정의 순서도 상관이 없습니다. 또한 함수를 정의하기 위해서 다른 함수를 사용할 수 있습니다.

```python
import math

def get_dist(x1, y1, x2, y2):
    dx = x1 - x2
    dy = y1 - y2
    return (dx**2 + dy**2)**0.5

def circ_area(x1, y1, x2, y2):
    r = get_dist(x1, y1, x2, y2)
    return math.pi * r**2

print(circ_area(0, 0, 3, 4))
```

위 코드의 `circ_area(·)` 함수를 정의하기 위해 `get_dist(·)` 함수를 사용하였는데, `get_dist(·)` 함수의 정의와 `circ_area(·)` 함수 정의의 순서는 바뀌어도 상관은 없습니다. 하지만 코드를 읽는 사람이 위부터 읽어 내려갈 때 모르는 함수가 있으면 코드를 이해하는데 어려움이 생길 수 있으니 가독성을 위해서 순서를 적당히 지켜주는 것이 좋을 것 같습니다.

코드의 뼈대 부분을 `main()` 등의 함수로 묶어두고, 함수 정의 밖에는 `main()` 한 번만 호출되게 할 수도 있습니다. 이는 C/C++, Java 등의 언어에서는 기본적으로 채택되는 방식입니다. 예컨대, C에서는 `Hello, world!`를 출력하는 코드가 다음과 같습니다.

```c
#include <stdio.h>

int main()
{
    printf("Hello, world!\n");
    return 0;
}
```

Python 코드에서는 함수 없이 가장 윗 수준에 적힌 코드가 실행되는 반면, C에서는 `main()` 함수에 적힌 내용이 실행됩니다. 하지만 Python에서도 다음과 같이 코드를 구성할 수 있습니다.

```python
def main():
    print("Hello, world!")
```

```
3
4 main()
```

방금 정의한 `main()` 함수는 매개 변수를 가지지 않는데, 이러한 종류의 함수는 매개 변수 없이 항상 동일한 기능을 하게 됩니다.

Parameters매개 변수와 Arguments전달 인자

함수를 정의할 때 사용되는 매개 변수들은 함수 정의 밖에서 호출될 수 없습니다. 일종의 블랙박스로 생각하면 되는데, 함수 안에서 정의되는 local variable지역 변수는 그 안에서만 사용을 할 수 있습니다. 예를 들어 예시에서 본 `get_dist(·)` 함수에서 `dx`나 `dy`는 함수 안에서만 생성되었다가 사라지는 변수라고 생각하면 됩니다. 나아가 함수 밖에서 `dx`라는 이름의 새로운 변수를 만들더라도 이는 함수 안에서 정의되었던 지역 변수 `dx`와는 다른 변수입니다. 반면, 함수 밖에서 먼저 정의가 되었으면서 함수 안에서 다시 정의되지 않은 변수는 global variable전역 변수라고 합니다. 이러한 경우에는 함수 밖에서 정의가 된 값이 그대로 함수 안에서 사용됩니다. 또한 함수의 매개 변수도 함수 밖에서 사용될 수 없는 지역 변수입니다. 매개 변수의 이름과 함수의 밖에서 정의된 어떤 변수의 이름이 같더라도 이 둘은 관련이 없습니다. 물론, 해당 변수에 지정된 값이 같은 이름의 매개 변수로 정의된 함수의 arguments전달 인자로 지정될 수는 있습니다. 매개 변수는 항상 전달 인자로 값이 지정되기 때문에 전역 변수와 이름이 겹친다고 해도 영향을 받지 않습니다. 여기서 전달 인자는 `fn(3)`의 3처럼 단순히 함수에 전달되는 값입니다. 아래의 예시를 통해 좀 더 명확히 이해를 해보시길 바랍니다. 이해를 위해 같은 이름을 가졌더라도 가리키는 값이 다르면 다른 색상으로 표시하였습니다.

```python
def fn1(a, b):
    c = a + b
    d = a - b
    return (c + d) / 2.

a, b = 1, 2
print(fn1(a, b))
print(fn1(1, 2))

c, d = 3, 4

def fn2(a, b):
    d = a - b
    return (c + d) / 2.

print(fn2(c, d))
print(fn2(3, 4))
```

정리하자면, fn1(·)의 매개 변수 a와 b는 아래에서 전달 인자로 사용된 a = 1 및 b = 2와 무관합니다. 줄 7과 줄 8의 출력 값이 같다는 사실을 통해, 줄 7의 fn1(a, b)는 단지 밖에서 정의된 a의 값인 1과 b의 값인 2를 대입한 fn1(1, 2)와 완전히 동일한 역할을 한다는 것을 확인할 수 있습니다. fn2(·)의 경우는 약간 다릅니다. 함수 내부에서 c가 지역 변수로 새로 정의되지 않은 상태로, 줄 14에서 c를 사용하였습니다. 여기에서 쓰인 c는 c = 3로 정의된 전역 변수입니다. 반면 fn2(·) 내부에서 사용된 d는 전역 변수 d가 아니라 새로 a - b의 값을 가지는 지역 변수입니다. 이와 같이 어떤 변수가 유효성을 가지는 영역을 scope변수 영역이라고 합니다. 마지막 예시를 통해 변수 영역의 의미를 확실히 파악할 수 있을 것입니다.

```
1  def mult(a, b):
2      x = a * b
3      return x
4
5  def div(a, b):
6      x = a / float(b)
7      return x
8
9  a, b = 1, 2
10 print(mult(b, a))  # mult(2, 1)
11 print(div(a, b))  # div(1, 2)
12 print(x)  # NameError: name 'x' is not defined
```

함수 mult(·)와 div(·), 그리고 줄 9 이후의 a, b, x는 서로 무관하며, 줄 12에서는 함수 mult(·)와 div(·)의 변수 영역 밖에서 x가 정의된 적이 없으므로 오류가 발생합니다.

2.2 Conditionals조건문

그렇다면, 입력값(전달 인자)에 따라서 반환값이 달라지는 함수를 만들 때에는 어떻게 해야 할까요? 단순히, 절댓값 함수를 구현하려고 해도 다음과 같이 경우가 갈리게 됩니다.

$$|x| = \begin{cases} x & \text{if } x \geq 0 \\ -x & \text{otherwise} \end{cases}$$

이러한 함수를 구현하고 싶을 때 활용해야 하는 것이 바로 conditionals조건문입니다. 내장 함수인 abs(·)를 사용할 수도 있지만, 절댓값 함수는 다음과 같이 구현할 수 있습니다.

```
1  def abs(x):
2      if x >= 0:
3          return x
4      else:
5          return -x
```

Boolean Type불리언 자료형과 Expressions표현

지금까지 소개한 `int`, `float` 등의 자료형 외에도 자주 쓰게 될 자료형은 Boolean type불리언 자료형[2.5]입니다. 불리언 자료형은 어떤 표현의 참/거짓을 나타내는 형으로, `True`와 `False` 두 값을 가집니다. 위의 예시에서 본 표현인 `x >= 0`은 `True`일수도 `False`일수도 있는 불리언 자료형입니다. 불리언 표현을 구성하기 위해서는 대소 관계와 일치 여부 등을 판단하는 관계 연산자와 `and`, `or`, `not` 등의 논리 연산자로 구성될 수 있습니다. `x >= 0 or (y < 0 and z < 0)`이 불리언 자료형을 가지는 표현의 예시입니다.

관계 연산자는 다음의 여섯 가지 종류가 있습니다.

- `x == y`: `x`와 `y`가 일치하면(=) `True`
- `x != y`: `x`와 `y`가 일치하지 않으면(≠) `True`
- `x > y`: `x`가 `y` 초과이면(>) `True`
- `x < y`: `x`가 `y` 미만이면(<) `True`
- `x >= y`: `x`가 `y` 이상이면(≥) `True`
- `x <= y`: `x`가 `y` 이하면(≤) `True`

지금까지 어떤 변수 `x`에 값을 대입할 때에는 `=`를 사용하였습니다. `x = 10`과 같이 말입니다. 수학에서도 "let $x = 10$"처럼 =를 사용하는 것과 같은 맥락입니다. 나아가 수학에서는 보통 두 값을 비교할 때, "if $x = 10$, "과 같은 표현을 사용하는데, Python에서는 두 값을 비교할 때 단순히 등호 하나가 아니라 두 개를 연달아 쓴 `==`를 사용합니다. `if x == 10`과 같이 말입니다.

논리 연산자는 다음 표의 세 가지 종류가 있습니다.

p	q	p `and` q	p `or` q	`not` p
`True`	`True`	`True`	`True`	`False`
`True`	`False`	`False`	`True`	`False`
`False`	`True`	`False`	`True`	`True`
`False`	`False`	`False`	`False`	`True`

위 표에는 각 표현에 대해서 어떤 불리언 값을 가지는지 진리표가 제시되어 있습니다. `and`, `or`, `not`은 각각 수학의 $\wedge$, $\vee$, 그리고 $\neg$에 대응됩니다.

이제 여러가지 관계 연산자와 논리 연산자에 대해서 알아보았으니, 기존에 알았던 연산의 순서에 새로 배운 연산자들을 넣어봅시다. 연산의 순서는 우선 순위에 따라 다음과 같이 나열할 수 있습니다.

1. $(\cdot)$
2. `**`

[2.5] 영국의 수학자이자 논리학자인 George Boole의 이름을 따온 것입니다.

3. *, /, %
4. +, -
5. ==, !=, >, <, >=, <=
6. not
7. and
8. or
9. =

모든 관계 연산자들은 논리 연산자들보다는 순위가 높고, 산술 연산자들보다는 낮습니다. 이전과 같이 우선 순위가 겹친다면 좌에서 우로 순서가 결정됩니다.

if-else Conditionals조건문

if-else statements문의 기본적인 형태는 아래와 같습니다.

```
1 if expression:
2     ...
3 else:
4     ...
```

위에서 expression에는 x%2 == 0 or x%3 == 0 등의 임의의 불리언 표현이 올 수 있습니다. 만약 제시된 불리언 표현이 True라면 줄 2의 내용이 실행되고, 그렇지 않다면 줄 4의 내용이 실행됩니다. 제시된 불리언 표현이 False일 때 실행할 내용이 없다면 else 부분은 빠뜨릴 수 있습니다. 다음과 같은 예시를 통해 확인해 봅시다.

```
1 x = 76
2 if x % 3 == 0 and x % 2 == 0:
3     print(x, "is divisible by 6")
4 else:
5     print(x, "is indivisible by 6")
```

실제로 실행해보면 76은 6의 배수가 아니기에 76 is indivisible by 6이 출력되는 것을 볼 수 있습니다.

만약 6의 배수가 아니라면, 적어도 3의 배수인지, 혹은 2의 배수인지까지 판별하고 싶다면 어떻게 해야 할까요? 일단 배운 것만으로는 다음과 같이 할 수 있을 것입니다.

```
1 x = 76
2 if x % 3 == 0 and x % 2 == 0:
```

```
    print(x, "is divisible by 6")
else:
    if x % 3 == 0:
        print(x, "is divisible by 3")
        else:
            if x % 2 == 0:
                print(x, "is divisible by 2")
            else:
                print(x, "is neither divisible by 3 nor 2")
```

6의 배수인지 체크한 후, 그렇지 않다면 다시 3의 배수인지 체크한 후, 그렇지 않다면 2의 배수인지 체크하는 코드입니다. 분명 작동은 잘 하겠지만, 가독성도 좋지 않고 깔끔하지도 않습니다. 이런 상황에서 쓸 수 있는 것이 `elif`[2.6]입니다. 세 가지 이상의 경우가 있을 때, `if`로 시작하여 두 번째부터 마지막 바로 직전 경우까지는 `elif`를, 마지막 예외의 경우에는 `else`로 마무리하는 방식입니다. 마찬가지로 모든 경우에 해당하지 않을 때 시행할 것이 없으면 마지막 `else`는 생략할 수 있습니다. `elif`를 활용하면 위의 코드를 아래와 같이 간결하게 나타낼 수 있습니다.

```
x = 76
if x % 3 == 0 and x % 2 == 0:
    print(x, "is divisible by 6")
elif x % 3 == 0:
    print(x, "is divisible by 3")
elif x % 2 == 0:
    print(x, "is divisible by 2")
else:
    print(x, "is neither divisible by 3 nor 2")
```

만약 `if-elif-...-elif-else`를 훑는 중간에 참인 것이 나오면, 해당 조건에서 실행되는 명령을 수행한 후 이후의 경우는 모두 건너뛰게 됩니다.

또한 흥미로운 사실은, `if-else` 문에서 불리언 표현 대신 `int` 등의 자료형을 가진 변수를 넣어도 된다는 것입니다. `0`은 거짓, `0` 이외의 모든 수는 참의 값을 가집니다. 그렇다고 이러한 값들이 정확히 `True` 혹은 `False`의 값을 가지는 것은 아닙니다. 실제로 `0 == False`, `1 == True`의 시행 결과는 `True`가 나오지만, `2 == True`에 대한 결괏값은 `False`이기 때문입니다. 또한 문자열의 경우 `""` 외의 하나 이상의 문자를 담고 있는 문자열은 참입니다. `""`는 거짓에 해당합니다. 아직은 배우지 않았지만, 문자열과 마찬가지로 list리스트 또한 하나 이상의 값을 담고 있는 경우 참, 그렇지 않은 `[]`은 거짓입니다. 이는 사실 `bool`형이 `int`형의 부분이기 때문인데, `True`는 정확히 `1`과, `False`는 정확히 `0`과 동일합니다.[2.7] 따라서, 다음과 같은 결과가 나옵니다.

[2.6] `else if`에서 `else`의 `el`만 따온 것입니다. C 언어에서는 그대로 `else if`를 사용합니다.

[2.7] 사실 Python 2에서는 `True`와 `False`가 키워드로 지정이 되어 있지 않아 일반 변수명으로 사용하여 아무 값으로나

```
>>> True * 3
3
>>> True * False
0
>>> (6 % 2 == 0) + False
1
```

그리고 이 사실을 활용하여 다음과 같은 (유의미한) 코드를 쓸 수 있습니다.

```
money = 100
if money:
    print("I have money")
else:
    print("I don't have money")
```

단, 여기서 중요한 사실은 불리언 `True`와 참인 표현이 같지 않다는 것입니다. 예컨대 `True != 100`입니다.

Multiple Exit Points 다중 반환점

한 함수에 `if-else` 문을 넣어 경우별로 값을 반환하도록 만들 수 있습니다. **2.2 Conditionals 조건문** 처음에 제시한 `abs(·)` 함수가 그 예시로, `x >= 0`일 경우 `x`를 반환하고 그렇지 않은 경우 `-x`를 반환하였습니다. 이러한 함수에는 여러 개의 `return`을 사용하였기 때문에 multiple exit points다중 반환점[2.8]이 있다고 표현하기도 합니다. 함수에 여러 개의 반환점이 있으면 경우에 따라 실수를 하여 예기치 못한 버그가 생길 수 있기 때문에, 다중 반환점을 쓰는 것이 바람직한가에 대한 논쟁이 있습니다. 하지만, 적절한 상황에서 사용한다면 분명 편리할 것입니다. 만약 `abs(·)` 함수를 하나의 반환점만을 사용한다면 아래와 같게 나타낼 수 있습니다.

```
def abs(x):
    if x >= 0:
        y = x
    else:
        y = -x
    return y
```

재지정할 수 있습니다. 따라서 사용자가 `True = 3`과 같은 식으로 값을 배정할 수 있습니다. 반면 Python 3에서는 `True`와 `False` 모두 키워드라 값을 지정하려고 하면 syntax 에러가 발생하게 됩니다.

[2.8] 한국어로 적절한 번역이 없어서 다중 반환점이라는 표현을 사용하였습니다.

2.2절 처음에서 제시한 코드와 위 코드의 우열을 가릴 수는 없습니다. 자신이 편한 방식을 택하면 됩니다. 다만 다중 반환점을 2.2절 처음에서 제시한 코드와 다르게, `else` 없이 아래와 같이 다시 쓸 수 있습니다.

```
def abs(x):
    if x >= 0:
        return x
    return -x
```

둘 다 다중 반환점을 사용하였지만, 약간의 의미 차이가 있습니다. 위 코드의 경우에는 가능한 경우를 모두 걸러서 마지막 경우에는 `-x`를 반환하라는 의미가 있습니다. 이렇게 다중 반환점을 사용할 경우, 일반적으로는 다음과 같이 함수를 만들 수 있습니다.

```
def abs(x):
    if ...:
        return ...
    if ...:
        return ...
    ...
    if ...:
        return ...
    return ...
```

2.3 예제

1. 정규분포를 표현하기 위해 사용되는 가우시안 함수는

$$g(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2},$$

이며, 이때 μ는 기댓값, σ는 표준편차입니다. 가우시안 함수 `gaussian(mu, sigma, x)`를 작성하세요.

```
import math

def gaussian(mu, sigma, x):
    # Add here!

print(gaussian(0, 1, 0))                  # 0.3989422804014327
print(gaussian(-2, math.sqrt(0.5), 0))    # 0.010333492677046037
```

2. 전하가 각각 q_1과 q_2인 두 물체 사이의 쿨롱힘은 거리 r에 따른 함수

$$F_C(r) = \frac{1}{4\pi\varepsilon_0}\frac{q_1 q_2}{r^2} \text{ where } \varepsilon_0 = 8.854\,187\,817 \times 10^{-12}\ \mathrm{F}\cdot\mathrm{m}^{-1}.$$

로 쓸 수 있습니다. 또한, 질량이 각각 m_1, m_2인 두 물체 사이의 만유인력은 거리 r에 따른 함수

$$F_g(r) = G\frac{m_1 m_2}{r^2} \text{ where } G = 6.674\,08 \times 10^{-11}\ \mathrm{m}^3\cdot\mathrm{kg}^{-1}\mathrm{s}^{-2}.$$

로 표현됩니다. 두 물체가 전하 `q1`과 `q2`, 그리고 질량 `m1`과 `m2`를 가지고, 거리 `r`만큼 떨어져있다고 가정합니다. 각각 쿨롱힘과 만유인력을 구하여 반환하는 함수 `coulomb(q1, q2, r)`과 `gravity(m1, m2, r)`을 작성하세요. 또한 이 둘 모두를 구하여 더하는 `total_force(q1, q2, m1, m2, r)`도 작성하세요.

```
import math

def coulomb(q1, q2, r):
    # Add here!

def gravity(m1, m2, r):
    # Add here!

def total_force(q1, q2, m1, m2, r):
    # Add here!

e_c = -1.6021766208e-19
e_m = 9.10938356e-31
p_c = -e_c
p_m = 1.672621898e-27
a_0 = 5.2917721067e-11

print(coulomb(e_c, p_c, a_0))
print(gravity(e_m, p_m, a_0))
print(total_force(e_c, p_c, e_m, p_m, a_0))
```

3. When $a \neq 0$, a function $f(x) = ax^2 + bx + c$ has an extremum. Write a function `extremum(a, b, c)` that returns the extremum of a function $f(x) = \mathtt{a}x^2 + \mathtt{b}x + c$.

```
def f(a, b, c, x):
    return a*x**2 + b*x + c

```

```
4 def extremum(a, b, c):
5     # Add here!
6
7 print(extremum(1, 5, 10))   # 3.75
8 print(extremum(1, -5, 10))  # 3.75
9 print(extremum(3, 7, 5))    # 0.916666666667
```

4. An integer `n` where $1 \leq$ `n` ≤ 9999 is given. Write a function `reverse(n)` that returns an integer with reversed digits of `n`.

```
1 # Add here!
2
3 print reverse(3702)  # 2073
4 print reverse(370)   # 73
5 print reverse(37)    # 73
6 print reverse(3)     # 3
```

5. 10 이상 990 이하의 10의 배수 `n`이 주어졌을 때, 10, 50, 100, 500원 동전을 최소로 사용해서 `n`원을 만드는데 사용되는 동전의 수를 반환하는 함수 `count_coins(n)`을 작성하세요.

```
1 # Add here!
2
3 print(count_coins(730))  # 6
4 print(count_coins(790))  # 8
5 print(count_coins(260))  # 4
6 print(count_coins(70))   # 3
```

6. Write a function `least(a, b, c)` that returns the least number among three numbers `a`, `b`, and `c`. The type of the numbers can be `int` or `float`.

```
1 # Add here!
2
3 print(least(1, 2, 3))  # 1
4 print(least(2, 1, 3))  # 1
5 print(least(1, 1, 2))  # 1
```

7. Write a function `count(n1, n2, n3, n4)` that counts the number of the multiple of 2 or 3 but not 6. Assume inputs are `int` type.

```
# Add here!

print(count(1, 6, 3, 4))    # 2
print(count(12, 3, 5, 15))  # 2
```

8. Write a function `quadrant(x, y)` that returns:

   - `"First Quadrant"` if $x > 0$ & $y > 0$
   - `"Second Quadrant"` if $x < 0$ & $y > 0$
   - `"Third Quadrant"` if $x < 0$ & $y < 0$
   - `"Fourth Quadrant"` if $x > 0$ & $y < 0$
   - `"On the Boundary"` otherwise

   Assume `x` and `y` are `float` type.

```
# Add here!

print(quadrant(10, 5))   # First Quadrant
print(quadrant(-5, 3))   # Second Quadrant
print(quadrant(-5, -7))  # Third Quadrant
print(quadrant(3, -5))   # Fourth Quadrant
print(quadrant(0, -3))   # On the Boundary
```

9. Write a function `ceiling(x)` that returns $\lceil x \rceil$. Use `if-else` statements.

```
# Add here!

print(ceil(4.3))   # 5
print(ceil(-0.3))  # 0
print(ceil(0.01))  # 1
```

10. Write a function `min_x(a, b, c)` that returns the integer x that minimizes $ax^2 + bx + c$. Assume that `a`, `b`, and `c` are `float` type, where $a > 0$ and $b < 0$. If such x is not unique, return the smallest one.

```
def f(a, b, c, x):
    return a*x**2 + b*x + c

```

```
def min_x(a, b, c):
    # Add here!

print(min_x(1, -9, 2))    # 4
print(min_x(9, -5, 0))    # 0
print(min_x(9, -15, 0))   # 1
print(min_x(7, -13, 3))   # 1
```

11. "S", "R", and "P" represent scissors, rock, and paper, respectively. Write a function rock_paper_scissors(a, b) that returns "a" if a wins, "b" if b wins, or Tie if the game is tied, where a and b are elements of {"S","R","P"}.

```
def rock_paper_scissors(a, b):
    if a == b:
        return "Tie"
    # Add here!

print(rock_paper_scissors("R","R"))  # Tie
print(rock_paper_scissors("R","S"))  # a
print(rock_paper_scissors("R","P"))  # b
print(rock_paper_scissors("S","S"))  # Tie
print(rock_paper_scissors("S","P"))  # a
print(rock_paper_scissors("S","R"))  # b
print(rock_paper_scissors("P","P"))  # Tie
print(rock_paper_scissors("P","R"))  # a
print(rock_paper_scissors("P","S"))  # b
```

12. gold1, silver1, and bronze1 represent the numbers of gold, silver, and bronze medals of the first country, respectively. gold2, silver2, and bronze2 represent the numbers of gold, silver, and bronze medals of the second country, respectively. Write a function better(gold1, silver1, bronze1, gold2, silver2, bronze2) that returns "First" if the first country achieved better than the second, "Second" if the second country achieved better, or "Tie" if they tied. The score is evaluated according to the gold-silver-bronze order, not by the sum of total medals.

```
def better(gold1, silver1, bronze1, gold2, silver2, bronze2):
    if gold1 > gold2:
        return "First"
    # Add here!
```

```
5
6  print(better(10,4,24, 1,35,25))  # First
7  print(better(1,35,25, 10,4,24))  # Second
8  print(better(10,18,0, 10,4,24))  # First
9  print(better(10,4,24, 10,18,0))  # Second
10 print(better(10,20,5, 10,20,4))  # First
11 print(better(10,20,4, 10,20,5))  # Second
12 print(better(10,20,5, 10,20,5))  # Tie
```

13. Write a function `area_triangle(a, b)` that returns the area of a triangle formed by $y = \texttt{a}x + \texttt{b}$, x-axis, and y-axis. Return 0 if no triangle is formed. Assume `a` and `b` are either `int` or `float` type.

```
1 def area_triangle(a, b):
2     # Add here!
```