

CSED101 Programming and Problem Solving

Lecture note about CSED101

Hyunseong Kong, POSTECH 24

Last updated: March 6, 2025

Contents

1. Fundamental Words - 꼭 알아야 하는 용어들	2
2. Introduction - 들어가며	5
2.1 Computational Thinking	5
2.2 Programming Language & OS	5
2.3 Python 특징	5
2.4 Program Structure	6
2.5 Programming Paradigm	6
3. Variables and Types - 변수와 타입	7
3.1 Variables	7
3.2 Data Types	7
3.3 Mutability	8
3.4 Shallow Copy & Deep Copy	8
4. Input and Output - 입력과 출력	9
4.1 Input	9
4.2 Output	9
5. Function and Module - 함수와 모듈	10
5.1 Function	10
5.2 Lambda	11
5.3 Variable Scope	11
5.4 Module	12
6. Operator and Expression - 연산자와 표현식	13
6.1 Operator	13

6.2 Expression	14
7. Control Flow - 흐름의 제어	14
7.1 Conditional Flow	14
7.2 Loop	16
7.2.1 While Loop	16
7.2.2 For Loop	16
7.3 Continue, Break	17
8. Data Structure - 데이터 구조	18
8.1 List	18
8.2 Tuple	20
8.3 Dictionary	21
8.4 String	22
8.5 Set	22
8.6 기타	23
9. File IO - 파일 입출력	24
9.1 File IO	24

※ 1. Fundamental Words

- Array: 한 가지 type의 값만을 넣을 수 있다.
- List: 다른 여러 type의 값들을 넣을 수 있다.
- ASCII: American Standard Code for Information Interchange, 7bit.
- Module: 큰 문제를 기능별 작은 단위로 나눈 것. 독립적으로 수행할 수 있는 프로그램 단위이다.
- Scratch, App Inventor, Entry: MIT, 네이버가 제작한 블록 코딩 플랫폼이다.
- C, C++: OS(Operating System)를 만들기 위해 개발된 언어이다.
- Python, JAVA: 보통 네트워크 상황에서 자주 쓰이며, 분산된 개인 PC를 연결하는 목적으로 개발되었다. OOP & procedural 언어이다.
- DOS(Disk OS): disk에 읽기, 쓰기 등의 명령을 수행하는 프로그램. TUI(Terminal User Interface) 기반이다.
- UNIX: multi user OS. C 언어로 작성되어 이식성이 높다는 특징이 있다. 보통 서버에 자주 이용된다. 기업 및 대규모 기관에서 주로 사용되며, 구현 코드는 비공개.

- UNIX 철학: 단일 목적의 소형 프로그램이 협력하여 작업 수행하는 것이 목표.
- Linux: Unix와 유사한 오픈소스 무료 프로젝트이다. 다양한 배포판이 존재함.
- MacOS: 애플이 개발한 OS. 사용자 친화적이고 보안이 강력하며, UNIX 기반이다.
- Windows: MS가 개발한 OS. 넓은 호환성을 갖는다.
- Git: 분산 버전 관리 시스템. 여러 사람이 공동으로 연구할 때 이메일로만 주고받을 수 없으니, 분산 버전 관리 시스템에 저장하여 사용한다.
- GitHub: MS에서 인수한 후에 관리 중인, GIT 이용하는 소프트웨어 개발 프로젝트를 위한 하나의 플랫폼. 코드 저장소를 호스팅하고 버전 관리를 간편하게 할 수 있게 도와준다.
 - repository: 코드 파일, 수정 이력, 관련 설정 등을 저장하는 하나의 저장소이다.
 - branch: 코드의 변경 사항을 메인 프로젝트에 영향을 주지 않고 실험할 수 있게 해주는 일종의 독립된 작업 공간. 하나의 repository 안에 여러 branch가 있으며, 기본적으로 프로젝트는 main branch에서 관리된다.
 - commit: 저장소의 파일 변경 사항을 기록한다.
 - pull request: 특정 branch를 main branch 병합해달라는 요청이다.
 - issue: 프로젝트의 버그 제보, 기능이나 개선 사항을 제안하는 데 사용한다.
- Tensorflow, Keras, CNTK, PyTorch: 인공지능의 기초, ML을 지원하는 오픈소스 라이브러리이다.
 - Tensorflow: tensor라고 하는 다차원 배열을 데이터 구조로 하는 라이브러리이다. 구글에서 관리 중.
 - Keras: Tensorflow에 통합된 고수준 API. 딥러닝 모델을 쉽고 빠르게 만들게 해준다.
 - CNTK: MS에서 만든 라이브러리. 분산 훈련을 지원하여 GPU 활용한 대규모 학습이 쉽게 가능하다.
 - PyTorch: Facebook에서 만든 라이브러리. pythontic한 코드 스타일, 자동 미분과 같은 기능으로 신경망 훈련을 크게 단순화시킨다.
- AlphaGo: Google DeepMind가 개발한 인공지능 프로그램으로 바둑을 두기 위해 설계되었다. 기계 학습과 신경망을 활용하여 사람보다 뛰어난 평가를 받았으며, 강화학습으로 학습되었다.

- OpenAI: 2015년에 설립된 AI 연구소. AI의 잠재력을 최대한 활용하여 인류에게 이익을 제공하는 것이 목적. 책임감 있고 안전한 AI 개발에 초점을 둔다. 샘 올트만, 일론 머스크 등이 설립하였으며, 일반인공지능(AGI) 존재 위험에 대한 염려가 설립 동기이다.
- transformer 아키텍처: 기존의 RNN, LSTM, CNN과는 달리 attention 매커니즘을 사용하여 sequence를 처리한다. self-attention은 입력 sequence의 각 요소 간의 상대적인 중요성을 계산하여 문맥을 보존하고, 긴 거리의 종속성을 캡처하는데 효과적이다.
- GPT3(Generative Pretrained Transformer): OpenAI에서 제작한 모델이며, transformer 아키텍처를 기반으로 하고 있다.
- SORA: OpenAI에서 제작한 영상 제작 모델. 정적인 노이즈에서 시작하여 여러 단계를 거쳐 노이즈를 제거하여 점점 비디오 변형시키는 diffusion 모델이며, GPT처럼 transformer 아키텍처를 사용하여 뛰어난 확장 성능을 제공한다.
- 디지털 트윈: 컴퓨터에 물리적인 물체를 정확하게 반영하도록 설계된 가상의 모델을 만들고, 현실에서 발생할 수 있는 상황을 컴퓨터로 시뮬레이션하여 결과를 미리 예측하는 기술.
- 메타버스: 가상 혹은 디지털 세계를 의미하는 용어. 현실 세계와 유사한 공간을 생성하고 사용자들이 가상으로 상호작용하는 환경을 지칭한다.
 - 디지털 트윈, 메타버스의 차이점: 디지털 트윈은 보통 현실을 모델링하여 분석하고 개선하는데 그 목적을 두지만, 메타버스는 가상 시스템에서 상호작용하는데 그 목적을 둔다.
- NFT: 블록체인 기술을 이용하여 서로 대체할 수 없는 고유한 특성을 가진 디지털 자산이다.
- 블록체인: 분산 데이터베이스 기술. 연결된 데이터를 블록이라는 단위로 묶어 체인 형태로 관리한다. 각 블록이 이전 블록의 고유한 식별자 포함하고 있고 데이터의 변경이나 위조를 방지한다.
 - 탈중앙화: 데이터를 체인 형태로 관리하여 중앙 서버에 집중되는 것을 막는다.
 - 무결성: 데이터 변경시 이전 블록부터 현재까지 모든 블록을 변경해야 하기에 변경이 사실상 불가능하다.
 - 투명성: 모든 거래 기록을 공개하여 신뢰성 높인다.
- AWS, Google Cloud, Azure: 클라우드 컴퓨팅 서비스를 제공하는 주요 플랫폼.

- AWS: 가장 큰 플랫폼이며 웹 호스팅 등에 주로 이용된다.
- Google Cloud: 구글에서 제공하는 머신 러닝 및 인공지능에 강점을 두는 플랫폼이다.
- Azure: MS에서 만들었고 기업용 솔루션과의 통합이 강점이다.

※ 2. Introduction

2.1 Computational Thinking

문제를 해결하고, 작업을 수행하고, 지식을 표현하며, 컴퓨터를 활용하여 가능한 한 효율적으로 작업하는 능력

- **추상화**^{Abstraction}: 복잡한 문제, 개념 단순화. 핵심 개념, 특성에 집중
- **알고리즘**^{Algorithm}: 문제 해결 위한 명확하고 구체적인 방법과 절차
- **자동화**^{Automation}: 일련의 작업 자동적으로 수행

사람이 직접 문제를 풀려면 시간이 너무 오래 걸리는 일을 컴퓨터에게 시키기 위해서는 문제를 컴퓨터가 풀 수 있는 방식으로 재정의해야 한다. 즉, 추상화된 데이터를 주고 컴퓨터가 문제를 풀어내게 하여 주어진 일을 자동화하는 일련의 과정에 대한 사고 체계가 바로 computational thinking이다.

2.2 Programming Language & OS

- **OS**^{Operating System}: 컴퓨터 하드웨어와 소프트웨어와 같은 컴퓨터 자원을 효율적으로 관리하기 위한 시스템.
- **DOS**^{Disk OS}: 텍스트 기반의 운영체제이다. 원시적인 형태의 문자 인터페이스(CLI)가 존재한다.
- **UNIX, Linux**: Unix는 독점 운영 체제인 반면 Linux는 오픈 소스이다. 둘은 굉장히 유사하며, 둘 모두 다양한 시스템 사이에 이식이 가능하고 멀티 테스킹과 다중 사용자 지원하는 OS이다.

2.3 Python 특징

Python이라고 하는 언어는 코딩하기에 유용한 여러 장점을 가지고 있다. 이 과목에서 프로그래밍을 가르치는데 Python을 선정한 것에는 다음과 같은 이유가 있다.

1. **동적 타이핑**^{dynamic typing}: 변수의 타입이 자동으로 결정된다. 이는 개발자에게 유연성을 제공한다.
2. **OOP**^{Object Oriented Programming}: class, object를 사용한다. 데이터 처리하는 method를 캡슐화하여 코드 재사용성 & 유지 보수성이 향상된다.
3. **코딩 용이**: 쉽게 사용가능한 여러 라이브러리가 존재한다.
4. **가독성, 간결성**: 문법이 쉽고 들여쓰기로 가독성을 높였다.
5. **크로스 플랫폼 지원**: 다양한 운영체제에서 동일한 코드 사용할 수 있다.
6. **확장성**: 다른 언어에서 작성된 모듈 사용하거나, 파이썬에서 작성된 모듈을 다른 언어에서 사용 가능하다. 여기에는 적절한 미들웨어(운영 체제와 응용 소프트웨어의 사이를 조정하는 소프트웨어)가 필요하다.

2.4 Program Structure

일반적으로 프로그램이 문제를 해결하는 과정은 다음의 순서를 따른다.

1. **데이터 입력**: 해결하고자 하는 문제와 관련된 데이터를 모두 입력받는다.
2. **문제해결 과정**: 입력받은 데이터를 출력한다. 이 과정에서 데이터를 처리하는 일련의 과정을 담은 알고리즘이 사용된다.
3. **데이터 출력**: 가공된 데이터를 출력한다. 파일에 값을 저장하거나, 화면에 출력하는 작업 등이 바로 이 과정이 된다.

2.5 Programming Paradigm

패러다임^{paradigm}이란, 문제를 푸는 하나의 방법을 의미한다. 같은 문제가 주어졌을 때, 내가 어떤 시선을 가지고 있는지에 따라 문제를 해결하는 방법에 변화가 생기게 된다. 프로그래밍 또한 이전에 사용되던 문제 풀이 방법의 한계가 발견되면서 점점 발전되어왔다. 아래는 각 패러다임의 이름과 그 설명을 나타내 것이다.

- **절차지향**^{procedual}: 처리해야 할 문제의 해결 과정을 큰 문제를 독립적인 기능별로 나눠서 일련의 순서에 따라서 처리. 함수가 필수적.
- **객체지향**^{object oriented}: 관계 있는 데이터와 함수를 하나로 묶어서 선언하는 클래스 도입. 클래스는 객체를 생성하는 데이터 타입의 역할을 한다. 객체지향 개념(상속, 다형성 등) 활용하여 효율적인 코드 작성.
- **함수지향**^{functional}: 자료 처리를 수학적 함수의 계산으로 취급. 람다 함수(고차원 함수)가 지원. 코드 간결하게 만들어줌. 복잡한 로직에는 일반적인 함수 사용.

※ 3. Variables and Types

3.1 Variables

변수^{variable}란 데이터를 저장할 수 있는 메모리 공간에 붙여진 이름이다. 변수는 식별자^{identifier}로 구분하여 사용된다. 식별자는 곧 변수의 고유한 이름이며 아래와 같은 이름 규칙의 제한을 받는다.

- **이름 규칙:** 문자, 숫자, 밑줄("_")만 사용 가능. 예약어를 쓸 수 없으며 대소문자를 구별한다. 또한, 변수의 이름은 문자로 시작해야 한다.

다른 언어들과 다르게, Python은 변수와 객체의 메모리를 자동으로 관리하기 때문에 프로그래머가 메모리를 덜 신경써도 된다.

3.2 Data Types

Python에서 제공하는 기본 데이터 타입은 다음의 7가지 종류가 있다.

1. **number:** 숫자를 저장하며, int(정수), float(실수)가 해당한다.
2. **bool:** 참, 거짓을 저장한다.
3. **string:** 문자열(문자들의 나열)을 저장한다.
4. **list:** 여러 개의 요소를 순서대로 저장한다.
5. **tuple:** list와 유사하지만 값을 변경하는 것이 불가능하다.
6. **dictionary:** key-value 쌍으로 구성된 데이터를 저장한다.
7. **set:** list와 유사하지만 중복되지 않는 요소로 구성된다.

변수는 객체를 가리키는 아이디를 담고 있는 저장 공간이다. 다음과 같이 입력하면 변수의 아이디를 출력할 수 있다.

```
x=10
id(x)
```

3.3 Mutability

Python의 타입은 **변경 가능한 값**^{mutable value}과 **불가능한 값**^{immutable value} 이렇게 2가지 종류로 구분지을 수 있다. 모든 값은 객체로 처리된다. 이는 나중에 더 자세히 다룰 것이다.

- **mutable**: list, dictionary, set이 속한다.
- **immutable**: number, bool, string, tuple이 속한다.

만약, 변경이 불가능한 값이 할당되어 있는 변수에 새로운 값을 집어넣으려 하면 기존의 값은 지워지고 새 값이 들어가게 된다.

3.4 Shallow Copy & Deep Copy

mutable value는 기본적으로 값이 복사가 되지 않는다. 아래와 같은 예제를 보자.

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

결과: [1, 2, 3, 4]

위의 예제는 a라고 하는 list를 b로 복사한 뒤, b의 뒷부분에 4를 추가하고 있다. 이 때, a를 출력해보면 4가 추가되어 있는 것을 알 수 있다. 즉, b에 복사가 원활하게 이루어지지 않은 것이다.

다행히도, copy라고 하는 module에는 전체 값을 복사하게 해주는 함수 deepcopy()가 있다. 다음은 이 함수를 사용하여 mutable의 일종인 list를 복사하는 예제이다.

```
import copy
a = [1, 2, 3]
b = copy.deepcopy(a)
```

이제, b는 a와는 독립적인 새로운 list가 되었다.

※ 4. Input and Output

4.1 Input

입력은 입력 함수 `input()`를 입력하여 수행할 수 있다. 모든 입력은 문자열이며, 띄어쓰기로는 입력이 끝나지 않고, 엔터키를 눌러서 줄바꿈을 수행해야 입력이 끝난다. 아래는 입력을 받아 변수 `a`에 그 값을 저장하는 예제이다.

```
a = input("값을 입력하세요: ")
print("입력된 값:", a)
```

4.2 Output

출력은 출력 함수 `print()`를 입력하여 수행할 수 있다. 모든 출력은 문자열이며, 특수 기호 `%`를 이용하여 변수 등을 출력할 수 있다. 다음은 그 예시이다.

```
a = 10
print("%d %d" % (a, 100))
```

결과: `10 100`

특수 기호 종류	설명
<code>%d</code>	오른쪽 정렬, 10진수로 출력, 5자리 확보. 5자리 이상이면 그대로 출력.
<code>%x</code>	오른쪽 정렬, 16진수로 출력
<code>%o</code>	오른쪽 정렬, 8진수로 출력
<code>%5d</code>	오른쪽 정렬, 최소 5자리 확보, 5자리 이상이면 그대로 출력.
<code>%05d</code>	오른쪽 정렬, 빈자리는 0으로 채워서 최소 5자리 확보.
<code>%f</code>	실수, 기본적으로 소수점 아래 6자리까지 출력
<code>%7.1f</code>	소수점 포함 최소 7자리 확보, 소수점 아래 1자리에서 반올림
<code>%c</code>	문자 하나 출력. 숫자는 ASCII 문자로 변환하여 출력. 예를 들어, 10진수, 16진수, 8진수로 입력 시 해당 ASCII 문자로 변환
<code>%s</code>	문자열 출력

위의 `%`가 수행했던 대응을 `.format`도 동일하게 수행 가능하다. `format`은 문자열의

매서드이다. 또한, 괄호 {}를 이용하면 특수 기호를 더 세밀하게 제어 가능하다. 괄호의 첫 번째 숫자는 format 안의 매개변수의 위치를 나타낸다. 다음은 이것을 이용한 예제이다.

```
print("{1:05d}, {1:3d}, {0:d}".format(10,12))
```

결과: 00012, 12, 10

위의 예제에서 유의할 점은 2번째 12 앞에 공백이 하나 더 출력되었다는 것이다. 이는 3d가 최소 3자리를 확보하기 때문이다. 이때, 3자리 내에서 오른쪽 정렬되기에 "12" 가 출력된다.

문자열 앞에 f를 붙이면 f-string이 되는데, 이것을 이용해도 된다. f-string에서는 중괄호 안에 표현식이나 변수를 넣어서 사용할 수 있다. 아래는 그 예시이다.

```
i = 10
print(f"i+1={i+1}")
```

결과: i+1=11

f-string에서는 f"Pi: {number:.2f}"와 같이 변수 이름 뒤에 콜론(:)을 붙인 뒤, 위에서 언급한 모든 종류의 format도 이용할 수 있다.

문자열과 관련하여 한 가지 유용한 것이 더 있다. 파이썬은 앞에 r을 붙이면 전체 문자열을 특수 기호가 없는 문자열로 취급한다. 다음은 그것을 이용한 예제이다.

```
print(r"\n \t \\")
```

결과: \n \t \\

※ 5. Function and Module

5.1 Function

함수^{function}는 독립적인 기능을 수행하는 프로그램의 단위이다. 즉, 특정 작업을 수행하는 명령어의 모음에 이름을 붙이는 것이다. 함수는 매개변수를 받고, 반환 값을 돌려줄 수 있다. 여러 값 반환할 경우에, 값이 tuple 자료형으로 반환된다.

- 장점: 읽기 쉽고 이해하기 쉬우며, 코드 중복 막음.

함수를 선언하게 되면 메모리에 함수 객체가 생성되며 그 후, 이를 가리키는 참조가 생긴다.

list같은 mutable 객체는 매개변수로 함수에 넘겨주면 원래 리스트에 영향을 준다.

이때, 매개변수에 새롭게 다른 값을 대입 연산하는 순간 그 매개변수는 새로운 객체로 취급되어서 더 이상 기존 변수에 영향을 주지 않는다.

아래는 매개변수 2개를 받아서 덧셈 연산을 수행하고 이를 반환하는 함수의 예제이다. 반환 값은 `return` 키워드를 통해 돌려줄 수 있다.

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

5.2 Lambda

익명 함수는 말 그대로 이름이 없는 함수이다. 보통 한 줄 정도의 간단한 함수의 작성에 이용한다.

`lambda` 키워드를 이용하면 이름 없이 parameter와 expression만으로 구성된다. 익명 함수는 함수를 인자로 받는 함수(고차 함수) 등에 이용된다. `sort`가 이러한 고차 함수의 대표적인 예시로 어떠한 List의 정렬을 수행한다고 할 때, 두 원소를 비교하는 방법을 익명 함수로 제공하여 하나의 정렬 함수인 `sort`를 더 다양하게 쓸 수 있도록 한다. 즉, 이러한 익명 함수는 함수의 다형성을 높이는데 기여한다.

- 익명 함수를 변수에 넣어서 마치 함수처럼 사용하는 예시

```
sum_func = lambda a, b: a + b
sum_func(1, 2)
```

- 정렬하는 예시

```
sorted_list = sorted(list, key=lambda x: len(x))
```

5.3 Variable Scope

스코프 *scope*는 변수에 접근 가능한 범위를 의미한다. 다음은 스코프의 종류에 따라 구분한 것이다.

- 지역변수 *local variable*: 함수 내에서만 사용
- 전역변수 *global variable*: 프로그램 전체에서 사용

함수 내에 선언된 변수는 기본적으로 지역 변수이다. 그러나, 함수 내에 `global` 키워드를 사용하면, 함수 내에서도 전역 변수를 만들 수 있다. 대신 이 전역 변수를 다른

함수의 내부에서 사용하려면 똑같이 global 키워드로 이를 명시해주어야 한다. 파이썬 인터프리터는 global을 붙이지 않으면 local 변수인지 체크하고, 그 후에 해당하는 지역 변수가 없으면 오류를 반환한다.

아래는 함수 내에 global 키워드를 붙이고 그것을 다른 함수에서 사용하는 예시이다.

```
def fun1():
    global x
    x = 10

def fun2():
    global x
    print(x)

fun1()
fun2()
```

결과: 10

위의 예시에서 볼 수 있는 것처럼, 함수 내에서 global 키워드를 사용해야 변수 x의 값을 공유하는 것을 확인할 수 있다.

5.4 Module

모듈^{module}은 함수나 변수, 클래스 등을 모아 놓은 하나의 파일이다. 모듈 내부에는 파이썬 문법에 맞는 명령어들이 저장된다. 다음은 이미 정의되어 있는 여러 모듈의 예시이다.

- keyword module: python의 예약어가 저장되어 있는 모듈.

함수 이름	설명
iskeyword(x)	x가 예약어인지 판별함
kwlist	모든 예약어 리스트

- random module: 무작위 값들과 관련한 함수가 저장되어 있는 모듈.

함수 이름	설명
randint(a,b)	a부터 b까지 무작위 수
random()	0부터 1까지 무작위 수

함수 이름	설명
shuffle(x)	list x를 섞음
choice(x)	list x에서 원소 하나를 선택

※ 6. Operator and Expression

6.1 Operator

연산자 *Operator*는 Python에서 연산을 수행할 수 있도록 정의된 기호나 키워드를 의미한다. 연산자는 받는 값의 개수에 따라 단항 연산자, 이항 연산자로 구분할 수 있다. 파이썬에서 연산자의 종류는 아래와 같다.

- **산술** *Arithmetic*: 기본적으로 수의 연산을 다루는 연산자.
- **관계** *Relational*: 양 변의 관계를 다루는 연산자. 조건에 부합하면 `true`, 부합하지 않으면 `false`를 반환한다.
- **논리** *Logical*: 논리식을 다룬다. 논리식이 참이면 `true`, 참이 아니면 `false`를 반환한다.
- **대입** *Assignment*: 연산자의 오른쪽 항을 먼저 계산 후에, 그 결과를 왼쪽에 넣는 연산자.
- **단축 대입**: 산술 연산자와 대입 연산자를 합쳐놓은 연산자.

연산자 형태	종류
산술	<code>+, -, *(곱하기), /(나누기), **(거듭제곱)</code> <code>//(정수 나누기 몫), %(정수 나누기 나머지)</code>
관계	<code>==(양 변이 같은지 판별), !=(양 변이 다른지 판별)</code> <code><(우변이 큰지 판별), <=(우변이 크거나 같은지 판별)</code> <code>>(좌변이 큰지 판별), >=(좌변이 크거나 같은지 판별)</code>
논리	<code>and(양 변이 둘 다 참인지 판별)</code> <code>or(양 변 중 하나라도 참인지 판별)</code> <code>not(거짓인지 판별)</code>
대입	<code>=</code>
단축 대입	<code>+=, -=, *=, /=, //=%, **=</code>

논리 연산자에서 `not`은 단항 연산자임을 유의해야 한다.

관계 연산자는 아래처럼 여러 연산자를 동시에 중복하여 쓸 수 있다. 각각의 경우에 중복된 하나의 식이 어떻게 해석되는지는 아래와 같다.

- ' $a < b < c$ '는 ' $a < b$ ' and ' $b < c$ '로 해석된다.
- ' $1 > 0 == 0$ '는 ' $1 > 0$ ' and ' $0 == 0$ '으로 해석된다.

6.2 Expression

표현식 *Expression*은 값을 산출하는 하나의 코드 조각의 단위이다. 표현식은 하나 이상의 값으로 표현된다. 즉, 표현식은 평가할 수 있는 구문이다.

일상생활에서 이용하는 사칙연산 구문이 대표적인 예시이다. ' $1+2+3$ '과 같은 것은 결과적으로 6이라고 하는 하나의 값으로 표현이 가능하다. 프로그래밍에서는 함수 호출, 변수가 포함된 식, 배열 할당 연산 등도 모두 표현식에 포함된다.

중요한 점은, 표현식은 결과적으로 하나의 '값'으로 줄어든다는 것이다. 함수 호출의 경우, 함수의 반환값으로 줄어들며,

반면에, **구문** *Statement*은 '실행가능한' 프로그램의 단위로, 특정 작업을 수행하는 독립적인 코드 구문이다. 프로그래밍을 하면서 인터프리터가 이해하고 실행할 수 있는 모든 문자열은 다 구문이다. 문법적으로는 파이썬의 코드 한 줄이나 한 블록이 전부 구문이라고 할 수 있다.

할당, 조건, 반복, 함수 정의, 반환, 모듈 `import`가 구문에 해당되며, 구문은 흔히 한 개 이상의 표현식을 포함하는 경우가 많다. 즉, 구문은 표현식을 포함하는 더 큰 개념이다.

※ 7. Control Flow

흐름을 제어하는 구문은 조건문, 반복문 2개가 있다.

7.1 Conditional Flow

특정한 조건을 만족했을 때, 주어진 코드를 실행하는 구문을 조건문이라고 부른다.

- **if**: 주어진 조건식이 참이면 내부 코드가 처리되고, 반대로 거짓이면 코드가 무시되는 종류의 제어문.
- **if-else**: 주어진 조건식이 참이었을 때 내부 코드가 처리되는 것은 동일하나, 거짓일 때 또 다른 종류의 코드를 실행할 수 있게 하는 종류의 제어문.

if-else 문은 아래와 같은 예제를 통해 쉽게 익힐 수 있다.

```
x = 10
if x > 11:
    print("x is bigger than 11")
else:
    print("x is smaller or equal than 11")
```

결과: x is smaller or equal than 11

위에서 보이는 것처럼 if 뒤에 조건식인 $x > 11$ 을 넣어서 평가하도록 하였다.

if-else 구문은 여러 번 중첩해서 사용할 수도 있는데, 이럴 때는 중간에 elif라는 명령어를 넣는다. 아래는 그 예시이다.

```
def grade(x):
    if x > 95:
        print("A+!!!")
    elif 95 >= x > 90:
        print("A0!!!")
    elif 90 >= x > 85:
        print("A-!!!")
    else:
        print("Not an A...")
```

```
grade(97)
grade(89)
grade(10)
grade(-1)
```

결과:
A+!!!
A-!!!
Not an A...
Not an A...

모든 파이썬 파일(모듈)은 `__name__`이라는 내장 변수를 가진다. 그 값은 파일이 주 파일로서 실행될 때, `__main__`로 설정된다. 반면, 만약 파일이 import 되면, `__name__`은 모듈 이름으로 설정된다. 이것을 이용하여 아래와 같이 써서 이 파일이 주 파일로서 실행될 때만 코드를 실행하게 할 수 있다.

```
if __name__ == "__main__":
    pass # Do something else
```

7.2 Loop

반복문은 특정 조건이 만족하는 상황에서, 주어진 코드를 반복적으로 실행하는 구문이다. 반복문은 다시 for, while 2개로 종류를 구분할 수 있다.

- **while문**: 주어진 조건식이 참이면 내부 코드가 반복적으로 수행함.
- **for문**: 리스트와 같은 시퀀스형 데이터를 순회하며 각 요소에 대해 작업을 수행하는 것에 주로 이용함.

7.2.1 While Loop

while 문 같은 경우에는 아래와 같은 예시가 존재한다.

```
i = 0
while i<10:
    print(f"{i}회 실행했습니다.")
    i += 1
```

결과:

```
0회 실행했습니다.
1회 실행했습니다.
...
9회 실행했습니다.
```

위의 예시에서는 변수 i에 0을 대입한 뒤, i가 10보다 작으면 내부 코드를 계속 순회하게 한다. 반복문은 코드를 잘못 작성하는 경우에, 반복이 끝나지 않고 영원히 돌아가는 오류가 발생할 수 있으므로, 조건을 철저하게 작성해야 한다.

7.2.2 For Loop

for 문 같은 경우에는 list, tuple, set과 같이 원소 여러 개가 묶여있는 자료형을 처리하는데 주로 사용한다.

```
list_a = [1, 4, 2]
for data_a in list_a:
    print(f"{data_a}")
```

결과:

1
4
2

혹은, 아래와 같은 함수를 사용하여 의도적으로 리스트를 만들고 그것을 응용할 수 있다.

- range(x): 0부터 x-1까지 리스트 [0,1,2,...,x-2,x-1]를 반환.
- range(a,b,c): a부터 b-1까지 c만큼 더해가며 리스트 [a,a+c,a+2c,...]를 반환. 만약, c를 계속 더하다가 b-1을 넘어가면 b-1이 들어가지 않고 끝남.

```
for i in range(3,10,2):
    print(f"출력: {i}")
```

결과:

출력: 3
출력: 5
출력: 7
출력: 9

range 함수는 잘만 사용하면 여러 동작을 매끄럽게 처리할 수 있는 강력한 도구이기 때문에 활용을 해보려 노력하는 것이 좋다.

7.3 Continue, Break

파이썬은 반복문의 중간에서 반복문을 나가거나 조작할 수 있는 구문을 제공한다. 조건문은 아래의 키워드를 모두 무시하며, 반복문 안의 조건문 안에 해당 키워드가 있으면 바깥쪽 반복문에 영향을 끼친다.

- continue: 반복문의 현재 단계를 건너뛴다.
- break: 반복문을 탈출한다.

아래는 for문에서 continue를 사용하는 예제이다.

```

for i in range(5):
    if i % 2 != 0: # 홀수라면
        continue # 다음 반복으로 건너뜀
    print(i)

```

결과:

0
2
4

※ 8. Data Structure

8.1 List

리스트 *List*는 여러 개의 값을 저장할 수 있는 자료형이다. 여러 개의 숫자나 문자들을 하나의 변수에 담고 싶을 때 리스트를 사용할 수 있다. 리스트에 저장된 값은 순서가 있으며, 값을 변경할 수 있다. 또한, 여러 데이터 타입을 하나의 리스트에 저장할 수 있다.

파이썬에서 리스트는 대괄호("[]")를 이용하여로 생성할 수 있다. 다음은 리스트를 만드는 예제이다.

```

my_list = [1, 2, 3]
empty_list = []

```

리스트에서 특정 위치의 값에 접근하거나 변경하려면 인덱스 *Index*를 사용해야 한다. 인덱스는 0부터 시작하기 때문에 이를 유의해야 한다. 다음은 리스트의 5번째 값을 가져오고 4번째 값을 -1로 수정하는 예제이다.

```

fifth_value = my_list[4]
my_list[3] = -1

```

다음은 여러 리스트 관련 기능을 분류한 것이다. 각 분류는 이름을 포함하여 알아두는 것이 좋다.

- 슬라이싱 *Slicing*: `list[a,b]`와 같이 입력하여 인덱스 a부터 b-1까지의 원소를 자르고 반환할 수 있다. 비슷하게, `list[a:]`는 인덱스 a부터 끝 원소까지, `list[:a]`는 처음부터 인덱스 a-1 원소까지를 반환한다.

- **합치기** *Concatenate*: 더하기 +를 이용하여 두 리스트를 합친다.

```
x = [1, 2]
y = [3, 4, 5]
print(x + y) # [1, 2, 3, 4, 5] 출력
```

- **반복** *Multiply*: 곱하기 *를 이용하여 list를 반복하여 더할 수 있다.

```
x = [1, 2, 3]
print(x * 3) # [1, 2, 3, 1, 2, 3, 1, 2, 3] 출력
```

- **삭제** *Delete*: 특정 인덱스 범위의 값 전체를 삭제하기 위해서는 list[a:b]=[]와 같이 입력한다. 이 코드는 리스트의 인덱스 a부터 인덱스 b-1까지 원소를 삭제한다.

```
x = [1, 2, 3, 4, 5]
x[2:4] = []
print(x) # [1, 2, 5] 출력
```

- **언팩킹** *Unpacking*: list를 분해하여 각 원소를 새로운 변수에 담는다.

```
p = [1, 2]
a, b = p # a, b에 각각 1, 2가 담기게 된다.
```

- **zipping**: 2개 이상의 list를, 각 list의 같은 index 원소끼리 묶어서 tuple로 반환한다. zip 함수를 이용한다.

```
a = [1, 2]
b = [3, 4]
print(zip(a, b)) # [(1, 3), (2, 4)] 출력
```

아래의 연산자는 if 문이나, while 문의 조건으로 사용될 수 있는 연산자이다.

- **in, not in** 연산자: list에 element가 포함되었는지 체크하는 연산

```
x = ['a', 'b', 'c']
if 'b' in x:
    print("x has b")
```

결과: x has b

위에서 설명한 `in`, `not in` 연산자는 `tuple`, `set`, `dictionary`에서도 사용할 수 있는 범용적인 연산자이며, 다양한 상황에서 응용되니 기억해두어야 한다.

아래는 리스트 관련 각종 함수를 나타낸 것이다.

함수 이름	설명
<code>list.append(x)</code>	<code>list</code> 뒤에 <code>x</code> 추가
<code>list.pop()</code>	맨 뒤 원소 반환하고 삭제
<code>list.sort()</code>	정렬
<code>list.reverse()</code>	순서 역순
<code>list.index(x)</code>	<code>x</code> 찾아서 반환
<code>list.insert(loc, x)</code>	<code>loc</code> 에 <code>x</code> 삽입
<code>list.remove(x)</code>	<code>x</code> 제거. 여러 개 있으면 첫 번째만
<code>list1.extend(list2)</code>	<code>list1</code> 뒤에 <code>list2</code> 추가하여 확장. +와 동일한 역할
<code>list.count(x)</code>	<code>list</code> 내부에 찾을 값 <code>x</code> 의 개수
<code>del(list[pos])</code>	<code>list</code> 에서 위치 <code>pos</code> 의 항목 삭제
<code>len(list)</code>	<code>list</code> 전체 항목 개수
<code>zip(list1, list2)</code>	<code>list1</code> 과 <code>list2</code> 를 같은 index끼리 묶기

8.2 Tuple

튜플 *Tuple*은 리스트와 유사한 형태의 자료형으로 기본적으로 내부의 원소가 변하지 않는다는 특징이 있다. 리스트와 유사하게 인덱스가 존재하여 인덱스로 원소를 추출할 수 있다. 그러나, 인덱스를 사용해 내부 원소를 바꾸는 것은 불가능하다.

튜플은 소괄호 ()를 이용해 생성하며, 튜플 간의 더하기가 가능하다. 튜플은 변경 불가능한 값이기 때문에 튜플 사이의 더하기 연산을 수행하면 아예 새로운 튜플이 튀어나오게 된다. 아래는 그 예시이다.

```
x = (1, 2, 3)
y = x * 2
print(x + y)
```

결과: `(1, 2, 3, 1, 2, 3, 1, 2, 3)`

튜플은 리스트와 마찬가지로 `concatenate`, `multiply`, `unpacking`, `zipping`이 모두 가능하다. `delete` 연산은 튜플이 수정불가능하다는 특징으로 인해 튜플 전체를 삭제하는 것만 허용된다.

아래는 튜플 관련 함수를 나타낸 것이다.

함수 이름	설명
tuple.index(x)	x 찾아서 반환
tuple.count(x)	찾을 값 x의 개수
del(tuple)	tuple 삭제
len(tuple)	tuple의 전체 항목 개수
zip(tuple1, tuple2)	두 튜플 tuple1, tuple2를 같은 index끼리 묶기

8.3 Dictionary

딕셔너리 *Dictionary*는 key:value로 구성되어 있는 가변적인 자료형이다. 딕셔너리의 내부의 요소를 추가, 삭제, 변경할 수 있다. 또한, 다양한 데이터 타입의 키와 값을 가질 수 있는 유연한 구조이다. 각 키는 고유하며, 순서는 정해져 있지 않다. (OrderedDict을 써서 순서를 정할 수도 있다.)

딕셔너리는 중괄호 {}를 이용해 생성한다. 보통 딕셔너리는 키를 입력하면 값이 나오기 때문에 데이터 검색에 효율적이다. 앞서 설명했듯이 각 키는 고유하기 때문에 이미 있는 키에 대입 연산하면 기존의 값을 변경할 수 있다. 다음은 딕셔너리를 생성하는 예제이다.

```
d = {'a': 3, 2: 7}
print(d['a'], d[2]) # 출력 순서는 실행 시마다 다를 수 있음
```

결과: [3, 7]

아래는 딕셔너리 관련 함수를 나타낸 것이다.

함수 이름	설명
dict.get(key)	key로 데이터 접근. dict[key]와 같이 간단하게 쓸 수 있음.
dict.keys()	모든 키 반환. dict_keys([1,2,'a'])와 같은 식으로 출력됨.
	list(dict.keys())와 같이 list로 캐스팅하여 dict_keys를 지울 수 있음.
dict.values()	모든 값 반환. dict_values([1,2,3])와 같은 식으로 출력됨.
dict.items()	key-value 쌍이 리스트로 묶인, 리스트가 나옴. dict_values([('a', 1), ...])과 같은 식으로 출력됨.
dict in key	key 있으면 true 반환.
dict.update(dict2)	병합
pop(key)	key 제거하고 값을 반환. 없으면 None.
del(dict[key])	원소 삭제.

함수 이름	설명
clear()	모든 요소 제거.

8.4 String

문자열 *String*은 문자들의 나열로 구성된 데이터 타입이다. 즉, 글자 *Character*의 리스트라고 볼 수 있다. 따라서, 리스트에서 했던 것처럼 인덱스를 활용해 글자 하나 하나에 접근이 가능하다. 주요 기능은 다음과 같다.

- **인덱싱** *Indexing*: 특정 글자를 얻는다.
- **슬라이싱** *Slicing*: 특정 범위의 요소를 선택한다.
- **합치기** *Concatenate*: 문자열을 합친다. +를 이용해 간단하게 수행할 수 있다.
- **반복** *Repetition*: 문자열을 반복한다. *를 이용해 간단하게 수행할 수 있다.
- **길이** *Length*: len()로 길이를 반환한다.

아래는 문자열 관련 함수를 나타낸 것이다.

함수 이름	설명
str.upper()	모든 문자를 대문자로 만듦.
str.lower()	모든 문자를 소문자로 만듦.
str.swapcase()	모든 문자의 대/소문자를 뒤바꿈.
str.title()	모든 단어의 첫글자만 대문자로 만듦.
ord(ch)	ch를 대응하는 고유한 숫자로 변환. 한글도 변환 가능하다.
chr(ch)	ord와 반대로 ch에 해당하는 문자 반환.
ch.islower()	소문자인지 판별.
ch.isupper()	대문자인지 판별.

8.5 Set

집합 *Set*은 중복되지 않는 요소들의 모임으로, 수학의 집합과 유사한 자료형이다.

집합은 중복을 허용하지 않고, 순서가 주어지지 않으며, 변경 가능한 자료형이기 때문에 요소의 추가나 삭제가 가능하다. 집합은 딕셔너리처럼 중괄호 {}를 이용해 생성한다. 아래는 집합을 생성하는 예제이다.

```
a = {1, 2, 3, 4, 5}
print(a) # 출력 순서는 실행 시마다 다를 수 있음
```

결과: {1, 2, 3, 4, 5}

다음은 집합의 주요한 연산을 나타낸다.

함수 이름	설명
set.add(x)	set에 단일 요소 x를 추가.
set.update(i)	다른 i(리스트, 집합, 튜플 등)의 여러 요소를 추가.
set.remove(x)	set에서 x 제거하고 없으면 KeyError 발생.
set.discard(x)	set에서 x 있으면 제거. 없어도 에러가 발생하지 않음.
set.pop()	임의의 요소 제거하고 반환.
set.clear()	set의 모든 요소 제거.
set1.union(set2)	두 집합 set1, set2의 합집합.
set1.intersection(set2)	두 집합 set1, set2의 교집합.
set1.difference(set2)	두 집합 set1, set2의 차집합.
set1.symmetric_difference(set2)	두 집합 set1, set2의 대칭 차집합(서로 공통되지 않은 요소).

아래의 4개 함수 union(|), intersection(&), difference(-), symmetric_difference(^)은 괄호 안의 연산자로 대체 가능하다.

집합은 리스트와 같이 변경 가능한 자료형을 요소로 가질 수 없다. 반면에, 변경 불가능한 자료형인 튜플은 요소로 가질 수 있다. 또한, 집합은 순서를 보장하지 않기 때문에 인덱스를 활용한 접근이 불가능하다. 인덱스를 활용한 접근이 필요하다면 list(set)을 이용하여 자료형을 리스트로 변환하여 사용할 수 있다.

8.6 기타

각 자료형은 서로의 자료형으로 변환이 가능하다. 함수 list(), set(), tuple(), dict()를 이용하면 된다.

아래는 리스트를 집합으로 변환하는 한가지 예제이다.

```
my_list = [1, 2, 2, 3, 4, 4, 5]
my_set = set(my_list)
print(my_set)
```

결과: {1, 2, 3, 4, 5}

※ 9. File IO

9.1 File IO

이 단원에서는 파일의 입출력에 대해 다룬다. 참고적으로, IO는 Input and Output의 줄임말이다.

앞서 다루었던 `input()`과 `print()`는 키보드 입력 및 모니터 출력을 다루는 함수였다. 파일 입출력은 비슷한 일을 파일을 통해 해결한다. 입력을 키보드로 수행하는 것이 아닌, 파일 내부의 값을 통해 수행하고, 출력도 마찬가지로 파일에 쓰기 연산을 통해 모니터에 출력하지 않고 수행한다.

콘솔 *Console*은 입출력을 위한 물리적인 장치로 키보드, 모니터 등이 여기에 포함된다. 보통 입력 받기 위한 인터페이스가 존재하게 되며 이러한 인터페이스 중에서도 CLI를 터미널 *Terminal*이라고 부른다.

기본적으로 `open` 함수를 이용해서 파일을 열 수 있다. 여는 방식(모드)에 따라 파일을 읽거나 쓰는 등 다양한 동작을 수행할 수 있다. 아래는 파일을 읽는 예시이다.

```
f = open('test.txt', 'r')
print(f)
```

결과: 10

아래는 가능한 읽기 형식 모드를 나타낸 것이다.

읽기 형식	설명
r (reading)	읽기, 기본값
w (writing)	덮어쓰기, 파일 없으면 생성
a (append)	새로운 텍스트를 추가
b (binary)	바이너리를 읽고 씀

아래는 파일 입출력 관련 함수를 나타낸 것이다.

함수 이름	설명
<code>read()</code>	모든 파일의 데이터를 한 번에 읽음.
<code>readline()</code>	파일을 한 줄씩 읽음.
<code>readlines()</code>	파일을 전부 읽어서 리스트를 반환함. 이때, 파일이 줄바꿈으로 구분되어 있었다면 리스트에 \n도 추가되어 있음.
<code>write(x)</code>	파일에 문자열 데이터를 씀.

함수 이름	설명
writelines(list)	파일에 문자열 리스트를 한번에 씀. 자동 줄 바꿈이 없기 때문에, 문자열 리스트 각 원소의 끝 부분마다 수동으로 \n 넣어주어야 줄 바꿈이 수행됨.
close()	파일을 닫음.

항상 프로그램이 종료되기 전에 close() 함수를 이용하여 파일을 닫아주어야 한다. 그렇지 않으면 파일이 손상되는 등의 문제가 발생할 수도 있다.

- with 구문을 이용하면 자동으로 열고 닫는 작업을 수행할 수 있다. 아래는 그 예시이다.

```
with open('test.txt', 'r') as file:
    print(file.read())
```

이 코드는 test.txt 파일을 r 모드로 열어서 읽고, 그것을 화면에 출력한 다음 파일을 닫는 동작을 수행한다.

References

- [1] Youn Eun Young. *CSED101 Lecture Presentation Slides*. POSTECH, 2024.
- [2] John V. Guttag. *Introduction to Computation and Programming Using Python, Third Edition*. MIT Press, 2021.