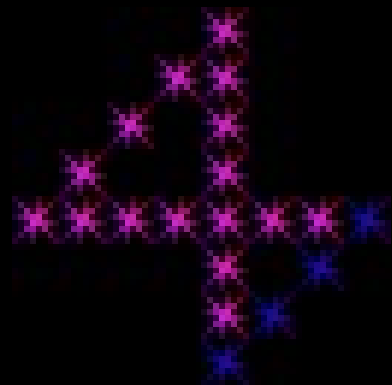


Starlit 4.1

Design Your Style!

Code Design Guidelines



Contents

1. Starlit 시작하기 전에

- 101 Starlit 언어 소개
- 102 Starlit 4.1 소개
- 103 Starlit 4.1 구축하기

2. Hello, World! 출력하기

- 201 간단하게 Hello, World! 출력하기
- 202 Full-Style로 출력하기
- 203 Starlic-Style로 출력하기
- 204 Simple-Style로 출력하기
- 205 Pythonic-Style로 출력하기
- 206 GUE-Style로 출력하기
- 207 Pseudo-GUE-Style로 출력하기
- 208 들여쓰기 규칙과 Hello, World!
- 209 문자열 변수를 사용하는 방법

3. 변수와 연산자

- 301 Starlit에서의 변수와 연산
- 302 Full-Style과 연산 코드
- 303 Starlic/Simple/Pythonic 스타일에서 연산 코드
- 304 GUE 스타일에서 연산 코드

4. OLED에 출력하기

- 401 OLED 글자 색과 위치, 그리고 크기
- 402 Full-Style에서 색깔 입혀서 출력하기
- 403 Starlic, Simple, Pythonic에서 색깔 입혀기
- 404 GUE-Style에서 색깔 입혀기
- 405 OLED 도형 출력
- 406 Full-Style에서 도형 출력하기
- 407 Starlic, Simple, Pythonic에서 도형 출력하기
- 408 GUE에서 도형 출력하기

5. 조건문 - if/when

- 501 조건문 if의 이해
- 502 Full-Style에서 if문 사용하기
- 503 Starlic, Simple, Pythonic에서 if문 사용하기
- 504 GUE에서 if문 사용하기
- 505 조건문 when의 이해
- 506 Full-Style에서 when문 사용하기
- 507 Starlic, Simple, Pythonic에서 when문 사용하기
- 508 GUE에서 when문 사용하기

6. 반복문 - while/for

- 601 반복문 while의 이해
- 602 Full-Style에서 while문 사용하기
- 603 Starlic, Simple, Pythonic에서 while문 사용하기
- 604 GUE에서 while문 사용하기
- 605 반복문 for의 이해
- 606 Full-Style에서 for문 사용하기
- 607 Starlic, Simple, Pythonic에서 for문 사용하기
- 608 GUE에서 for문 사용하기

7. 함수 정의하기

- 701 함수의 매개변수와 출력
- 702 Full-Style에서 매개변수와 출력
- 703 Starlic, Simple, Pythonic에서 매개변수와 출력
- 704 GUE에서 매개변수와 출력

1. Starlit 시작하기 전에

101 Starlit 언어 소개

Starlit 언어는 2019년부터 Arduino에서 LED Matrix를 자유자재로 작동시키는 방법을 찾다가 프로그래밍 언어로 발전해 온 프로젝트이다. Starlit 언어를 제작하기 전부터 LED Matrix 패턴을 MicroSD 메모리에 저장해서 출력해 왔지만, if/while문을 추가한 알고리즘의 형태를 2019년 도입하기 시작하여 2024년까지 안정화 작업을 거쳐 현재에 이르게 되었다.



파일형식	.txt	.dp	.sr	.slc	.slc	.sl4
실행파일	.txt	.dap	.psr	.sbc	.sbc	.sbc
시작		start>>	main	main	main	-
주석		///	/// /** **/	/// /** **/	/// /** **/	/// /** **/
조건문		if elif else	if elif else	if elif else	if elif else	if elif else when
반복문		while	while for	while for	while for 반복연산	while for 반복연산
반복문 탈출		미확인	break continue again	break continue again	break continue again	break continue again
Library				\$import	\$import	\$import

※ Starlit 2까지는 내장함수를 라이브러리에서 정의하지 않고 함수 리스트에서 정의하였음.

※ Starlit 3부터는 내장함수도 라이브러리에서 정의하여 \$import 구문을 필수적으로 사용.

■ Starlit 1 - Dot Project 시절(2019)

- 설계 배경 : LED Matrix를 자유롭게 제어하기 위한 알고리즘을 쉽게 설계하는 방법을 고찰.
- 이전 버전 대비 변경사항 : if문, while문과 LED Matrix를 제어하기 위한 공간(전역변수)을 마련.
- 주요 특징
 - `start>>`에서 시작하여 `end`에서 종료, end 뒤에는 한 칸 띄어쓰기해야 컴파일 오류가 발생하지 않는다.
 - 중괄호를 사용하지 않고 들여쓰기 규칙을 적용한다. 단, start와 end 내부에서는 들여쓰기를 사용하지 않는다.
 - `_a`는 당시 변수 a를 의미했다. 이때, `a`와 `_a`는 같은 의미로 쓰였는데, 변수 이름이 !, %와 같은 특수문자도 가능했기에 이런 경우 앞에 `_`를 꼭 붙여야 했기 때문이다.
 - 띄어쓰기가 반점과 같은 의미로 쓰인다.
 - if문과 while문이 존재했다. 이 밖의 구문을 절대 사용하지 않았다. 컴파일러의 간소화 때문이다.
 - 당시에는 컴파일러가 임베디드 시스템 내부에 내장되어 있었다. 따라서 코드를 그대로 업로드하여 작동시켰다.
 - 변수는 전역변수만 존재했고, 변수 이름이 곧 주소였던 시기여서 a라고 쓰면 주소가 'a'가 된다. 따라서 포인터를 나타낼 때 'a'와 같이 썼다.

● 예제 코드

```
start>>
portout 3 PORT_INPUT
while 1
    portAnalog 3 _a
    _a _a*50/1023
    getRGB _a*8/5 255 255 'R' 'G' 'B'
    twodigits _a (_R<<16)|(_G<<8)|_B 0
    dispimg 32
end
```

● S90(2020)

- Starlit 1의 개선형으로, LED Matrix 제어에 특화된 형태이다. 이때 만들어진 이미지 처리 함수들은 현재까지 유효하다(stuimg 라이브러리에서 사용).

■ Starlit 2 - Starlight 시절(2021~2022)

- 설계 배경 : Starlit 1의 한계를 느끼고, C언어의 형태를 빌려서 안정적이고 효율적으로 사용할 수 있게 설계
- 이전 버전 대비 주요 변경사항 : 지역변수, 함수 정의, C언어와 유사한 형태(세미콜론 선택사항)
- 주요 특징
 - 변수 정의는 #을 사용한다. 이때, 대입도 #을 사용해서 대입한다(#a(0)).
 - 변수를 나타낼 때도 #을 붙여서 표현한다. 단, #을 생략해서 표현해도 상관없다.
 - 변수는 정수형만 존재한다. 정수형으로 소수형을 흉내 내서 쓸 수 있다.
 - if문이나 while문을 사용할 때 반드시 중괄호를 써야 한다.
 - 들여쓰기 규칙에서 중괄호 규칙으로 변경하였다.
 - main 함수 안에 프로그램을 작성한다.
 - 함수의 출력은 존재하지 않아서 함수 안에 다른 함수를 넣을 수 없고, 출력해야 하는 함수는 레퍼런스를 받는다.
 - 변수 n의 레퍼런스는 `_n`으로 표기한다.
- 예제 코드

```
/**
    주사위 0~10의 이미지를 순차적으로 켜는 프로그램입니다.
**/

main(){
    #n(0)///이미지 개수
    #m(0)
    imgopn(0, "img/dice.bmp", _n)
    while(1){
        #m((#m+1)%#n)
        imgprt(#m)
        delay(1000)
    }
}
```

■ Starlit 3(2022~2024)

- 설계 배경 : STM32 프로세서를 본격적으로 사용하면서 정수, 소수형의 변수의 필요성을 느끼면서 개편, OLED를 더 자유자재로 제어할 수 있게 설계
- 이전 버전 대비 주요 변경 사항 : Class 개념 도입(이를 사용하여 소수형을 흉내낼 수 있음), 연산자 오버로딩, 출력이 있는 함수, 자연스러운 대입 연산
- 주요 특징
 - 변수 정의는 #으로 시작한다. 단, Starlit 2와 다르게 대입은 대입 연산을 사용한다. 정의 - #a():int; / 대입 : a = 20;
 - 클래스를 정의할 수 있다. 기본적으로 int 클래스와 float 클래스가 지원된다. 원래는 정수형만 정의되지만, float 클래스를 통해 소수도 사용할 수 있다.
 - 출력이 있는 함수를 정의할 수 있고, 같은 이름의 함수를 여러 번 정의할 수 있다. - f(a:int)와 f(a:float)는 다른 함수 취급한다.
 - 연산자 오버로딩이 가능하다. C++의 방식과 다르게 연산자는 그에 대응하는 함수명이 정의되어 있다.
 - Starlit 2와 다르게 세미콜론을 필수로 붙여야 한다. 또, 중괄호가 필요한 상황에서 중괄호도 반드시 붙여야 한다.
 - Starlit 1, 2와 다르게 Starlit 3에서의 기능은 Starlit 4에서도 대부분 적용된다.

● 예제 코드

```
main(){
    OLED_Clear();
    #result(j^^j):complex_t;
    OLED_cout << Re(j^^j) << "/b+\n" << Im(j^^j) << "\n";
    while(!BUTTON_Read()){}
```


■ Starlit 3.3(2024~2025)

- 설계 배경 : 변수의 정의와 대입 연산의 괴리감이 커서 변수를 좀 더 편하게 사용할 수 있게 설계
- 이전 버전 대비 주요 변경 사항 : 자연스러운 변수 정의 방식(a = int 등), 반복 연산자 추가
- 주요 특징
 - 변수 정의에서도 #을 사용하지 않는다. # 정의법은 옵션이고, 이제는 대입 형태로 변수 정의, 대입을 모두 할 수 있다(현재는 지역변수, 전역변수 모두 사용 가능).
 - OLED의 변화 - OLED 자체를 객체로 보는 방식을 도입해서 OLED_cout 대신 OLED 그 자체를 사용해 출력할 수 있다.

● 예제 코드

```
$import(oled);
$import(stuimg);
$target(apps\test\test.sbc);

main(){
    img = img_t;
    img.data[0:3, 0:7, 0:7] = 0;

    IMAGE_TwoWrite(_img, '2', '0', 0xB0FF00, 0x00FF30);
    MATR_Print(_img);

    while(!BUTTON_Read()){
    }
```

102 Starlit 4.1 소개

Starlit 4.1은 2019년에 처음으로 만들고 난 뒤 3.3 버전의 후속 버전이자 앞자리 수를 올린 버전이다. 앞자리 수를 바꾼 만큼 코드의 전반적인 내용에 변화가 있다. 하지만, 지금까지의 Starlit 버전은 이전 버전이 호환되지 않았지만, 지금은 이전 버전의 코드와 완벽히 호환해서 사용할 수 있고, 앞으로도 이전 버전과 호환되게 설계한다. 다만, 새로운 버전의 코드를 이전 버전의 컴파일러에 적용했을 때는 호환되지 않는다. 파일 형식 역시 s1c에서 s14로 바뀌지만, 호환성에 따라 바이트 코드는 sbc를 그대로 유지한다.

■ Why Starlit 4.1?

Starlit 4.1을 만든 데에는 LED Matrix를 작동하는 코드를 간결하게 나타내는 방법을 고찰하다가 설계한다. Starlit 언어는 높은 자유도로 어떤 문제를 해결하는 데에는 여러 방법이 있다는 언어지만, 간결하게 나타내는 방법도 존재해야 한다는 철학이 담겨 있다. 그러나 중괄호를 사용하면 간결하게 나타내는 데에 한계가 존재한다. 그래서 중괄호를 사용하는 방법을 다시 한번 깨서 Starlit 1과 Starlit 2의 요소도 부활하게 설계하는 것이다. 근본적으로 컴파일러 자체는 Starlit 3.3을 사용하고, 각각의 구문을 Starlit 3.3에서 해석할 수 있게 변환하는 방법을 사용해 이전 버전과 호환되면서 4.1만의 독특한 스타일을 만들어 간다.

■ Changes

이전 버전과 비교했을 때 달라진 점이라면 코드 스타일을 대표적으로 들 수 있다. Starlit 1에서는 들여쓰기 규칙과 괄호 없는 규칙 등이 적용되었고, Starlit 2에서는 세미콜론을 선택적으로 쓸 수 있게 하였다. Starlit 3에서는 세미콜론을 필수적으로 사용하게 했는데, Starlit 4에서는 중괄호를 사용할 때만 세미콜론을 붙이기로 한다. 그러면 중괄호 없이 코딩해도 되냐고? 바로 그것이 변경사항이다. 중괄호를 붙이면 자유롭게 들여쓰기 규칙을 무시해도 되지만 세미콜론을 붙여야 하고, 중괄호를 생략하면 들여쓰기 규칙이 추가되지만, 세미콜론 없는 코드를 사용할 수 있게 한다. 그래서 다중 코딩 스타일 언어를 표방한다.

■ Free Style

Starlit 4가 다중 코딩 스타일 언어인 만큼 사용자는 코드 스타일을 입맛에 맞게 선택해서 쓸 수 있다. 심지어 main 함수 대신 사용자 입맛에 맞는 형태로 코드를 시작해도 좋다. 심지어 함수 없이 구문을 써도 좋다.

■ Starlit4 동작 구성(기본형)

Starlit 4는 사용자가 자유롭게 설계할 수 있는 언어인 특성상 main함수가 아니어도 start함수나 setup...loop함수를 정의해서 써도 된다. 그러나 함수에는 서열이 있어 작동 순서를 확인해 보고 작성해 주면 좋다. 함수를 정의하지 않으면 무시한다. 또, 프로그램이 종료되면 버튼 한 번 누르고 나갈 수 있게 Waitkey가 내장될 수 있다. loop가 정의되어 있으면 무한 반복하여 프로그램이 종료되지 않는다. loop를 정의하지 않으면 프로그램을 종료할 수 있게 설계할 수 있다.

동작 순서	함수	설명
1	전역변수/구문	함수 밖에서 작성된 구문 중 유효한 코드 실행
2	setup	사용자가 직접 설정
3	start	본격적인 프로그램 시작
4	main	프로그램의 핵심 내용
5+	loop	무한 반복해서 실행하는 구문
6	WaitKey	프로그램 종료 전 버튼 누름

103 Starlit 4.1 구축하기

■ 사전에 설치할 프로그램 : HDL Works의 Scriptum

아래 링크에 들어가 [Scriptum 24.0을 다운로드](#)한다. 본인이 사용하던 개발환경은 22버전이었지만, 24에서도 사용할 수 있다. 파일을 실행하여 [agree]를 눌러 설치할 수 있다.

Scriptum은 Verilog와 같은 HDL을 편집할 수 있는 편집기지만, C언어와 같은 일반적인 프로그래밍 언어도 가볍게 편집할 수 있도록 설계되었고, 커스텀 프로그래밍 언어도 추가할 수 있는 특징이 있다. 이미 Scriptum이 설치되어 있다면 설치 과정은 건너뛰어도 상관없다.

■ Starlit 4.1 다운로드 및 압축 풀기

Starlit 4.1을 내려받고, 띄어쓰기가 전혀 없으면서 한글을 포함하지 않는 경로¹⁾ (문서 폴더는 실제 명칭이 Document로 되어 있어 사용 가능)에 압축을 푼다.

■ 편집기 경로 설정하기

압축 푼 폴더에 있는 **Preference.txt** 파일에서 ``STAR3.edit`` 값을 아래와 같이 설정한다. 복사 후 붙여넣자.

```
C:\Program Files\HDL Works\Scriptum24Rev1\bin\pc\scriptum.exe;
```

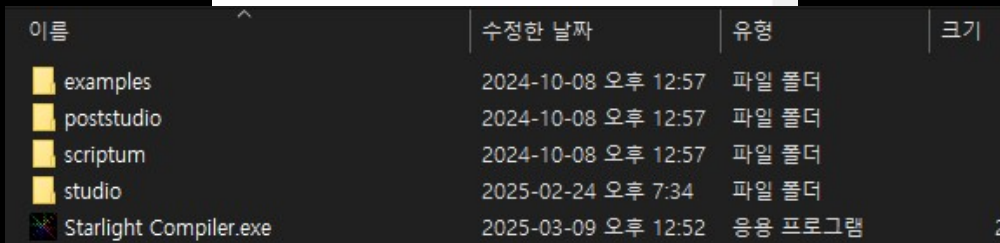
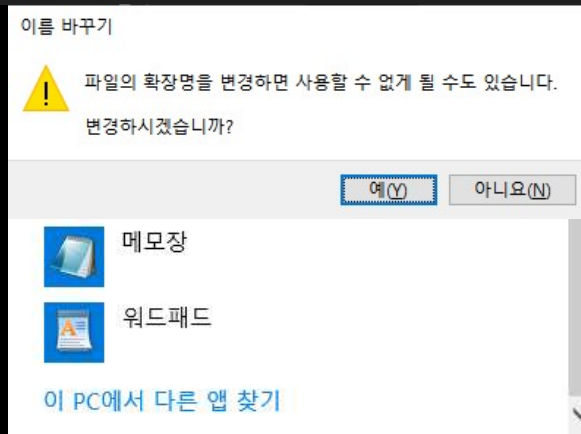
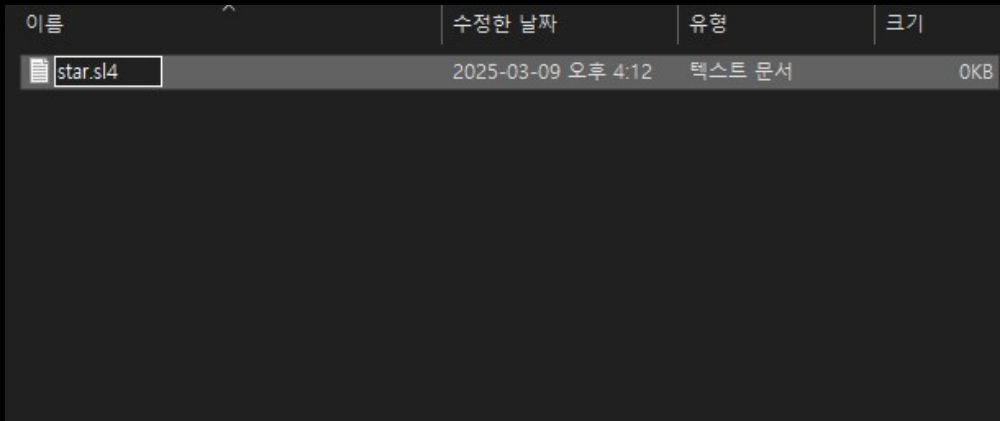
■ 서식 파일 설치하기

Starlight Compiler.exe 파일이 있는 폴더에서 **STAR_Scriptum_Setup.exe** 파일을 관리자 권한으로 실행한다. 그러면 서식 파일이 자동으로 적용된다.

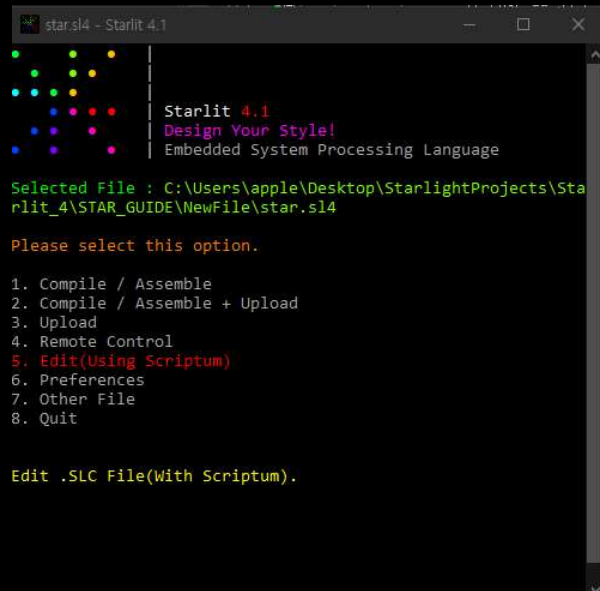
1) 여담이지만, STM32CubeIDE도 비슷한 이슈가 존재하여 한글을 포함하지 않는 경로에 붙일 것을 강조하는데, 그 옆에다가 붙이는 것을 추천한다.

■ Starlit4 파일 만들고 실행하기

적절한 폴더(띄어쓰기가 없으며 한글을 포함하지 않는 경로)에 마우스의 오른쪽을 클릭하고 **새로 만들기 - 텍스트 파일**을 선택한 후 파일 이름을 입력한다. 이때, 확장자는 `.txt`라고 되어 있는 것을 `.sl4`로 고쳐야 한다. 그러면 파일을 실행할 수 없게 되는데, 더블클릭하여 맨 아래로 내려 다른 앱 찾기를 클릭 후 Starlit 설치한 폴더에서 `Starlight Compiler.exe`를 찾는다.



파일을 실행하면 아래와 같은 창이 뜨는데, 1번이 컴파일을 진행한다는 의미고, 5번이 편집기로 편집한다는 의미이다. 우선 파일을 만들어야 하니 편집에 들어가 보겠다. 키보드로 **5번을 누르고 엔터**를 눌러 준다. 이제 편집기 창이 뜨는데, 여기서 코드를 만들면 된다.



2. Hello, World! 출력하기

2장에서는 `Hello, World!`를 출력하는 여러 가지 방법을 알아보겠다. 언제나 프로그램의 첫 시작은 `Hello, World!`를 출력하는 것인데, Starlit 4.1에서는 출력하는 방법이 매우 다양하다. 예제를 보면서 `Hello, World!`를 출력하는 다양한 스타일의 코드를 살펴보겠다.

201 간단하게 `Hello, World!` 출력하기

■ 가장 간단한 `Hello, World!` 출력 코드

`Hello, World!`를 출력하는 코드는 아주 간단하게 아래와 같이 작성해도 좋다. 물론 문자열만 단독으로 쓰는 코드인 만큼 일반적으로 사용하는 코드는 아님에 유의해야 한다.

HelloWorld.sl4	Result
<code>"Hello, World!"</code>	<code>Hello, World!</code>

실행 결과 OLED에 빨간 글자로 `Hello, World!`라고 출력되는 것을 확인할 수 있다. 빨간 글자로 출력되는 이유는 나중에 알게 될 것이다.

■ f-문자열을 사용한 `Hello, World!` 출력

아래의 코드는 f-문자열을 단독으로 사용해서 `Hello, World!`를 출력하는 예제이다. 앞에서의 코드에 f만 붙인 코드이다.

HelloWorld.sl4	Result
<code>f"Hello, World!"</code>	<code>Hello, World!</code>

실행 결과 역시 빨간 글자로 `Hello, World!`를 출력하는 것을 확인할 수 있다. 이 역시 문자열을 단독으로 사용한 것이므로 빨갇게 출력된다.

■ OLED.Print 명령어를 활용한 Hello, World! 출력 코드

다음은 OLED.Print 명령어를 사용해 Hello, World!를 출력하는 예제이다. OLED.Print 뒤에 Hello, World!를 적으면 된다.

HelloWorld.sl4	Result
<code>OLED.Print "Hello, World!"</code>	Hello, World!

실행 결과 앞서와 다르게 흰색 글자로 Hello, World가 출력된 것을 확인할 수 있다. 기본적으로 색상 설정 없이 OLED에 출력하면 흰색으로 문자열이 출력된다. 문자열을 단독으로 적었을 때 빨간 글자를 출력하는 것은 디버그 등을 위해 어디서 실행했는지 쉽게 확인하기 위해 그렇게 설계한 것이다.

■ OLED << 연산자를 활용한 출력

다음은 OLED << 연산자를 사용해서 Hello, World!를 출력하는 예제이다. 가장 일반적으로 문자열을 출력하는 구문으로 아래와 같이 간단히 작성해서 실행하면 된다.

HelloWorld.sl4	Result
<code>OLED << "Hello, World!"</code>	Hello, World!

실행 결과 흰색 글자로 Hello, World!를 출력하는 것을 알 수 있다. 역시 흰색으로 실행한 만큼 정상적인 방법으로 문자열을 출력했다고 볼 수 있다.

■ OLED.Print 함수를 활용한 출력

다음은 함수를 사용한 Hello, World! 출력 예제이다. 명령어와 비슷하지만 약간 다른데, 괄호를 감싼다는 차이점이 있다.)

HelloWorld.sl4	Result
<code>OLED.Print("Hello, World!")</code>	Hello, World!

■ Hello, World! 관련 이스터 에그

Hello, World!를 출력하는 프로그램을 처음 만드는 사람을 배려하기 위해 H만 작성해도 Hello, World!를 자동으로 출력하게 하였다. 다만, 색상은 약간 다르게 출력될 것이다.

HelloWorld.s14	Result
H	Hello, World!

실행 결과 노란색 글자로 Hello, World!가 출력된 것을 알 수 있다. 역시 정상적인 방법이 아닌 비정상적인 방법으로 H만 적으면 Hello, World!가 출력되어서 프로그래밍 초보자들도 쉽게 코딩할 수 있게 배려해 주었다.

■ 정리

지금까지 Hello, World!를 출력하는 방법을 표로 정리하면 다음과 같다. 이중 일부는 정상적으로 Hello, World!를 출력하는 방법은 아니니 유의하자. 앞으로 아래의 3가지 방식을 주로 활용해서 Hello, World!를 출력해 볼 것이다.

Style	Code	Result	Description
H	H	Hello, World!	일종의 이스터 에그로 H만 입력하면 Hello, World!가 출력되게 설계.
문자열 단독	"Hello, World!"	Hello, World!	문자열만 단독으로 입력하면 그 문자열이 빨간색으로 출력.
	f"Hello, World!"		f-문자열을 사용해도 똑같이 작동한다.
연산자	OLED << "Hello, World!"	Hello, World!	OLED에 문자열을 출력하는 연산자
명령어	OLED.Print "Hello, World!"		OLED에 문자열을 출력하는 명령어로 함수와 동치이다.
함수	OLED.Print("Hello, World!")		OLED에 문자열을 출력하는 함수.

202 Full-Style로 출력하기

■ Full-Style

Starlit 3에서 사용했던 형태로, **세미콜론**`;`과 **중괄호**`{...}`를 사용하는 방식이다. 이 방식을 사용하면 들여쓰기 제한을 받지 않고 코드를 구성할 수 있지만, 코드의 면적을 많이 차지하여 개발 속도가 느려지는 단점이 있다. C언어와 비슷하면서도 다른데, C언어는 구문 하나는 중괄호를 생략해도 좋지만(함수 제외), Starlit은 중괄호를 엄격하게 사용해야 한다.

■ Original Full-Style Code

다음은 가장 기본적인 형태로 **Hello, World!**를 출력하는 예제이다. Starlit 3에서 쓰던 방식으로, 중괄호와 세미콜론을 사용했다는 특징이 있다. **main** 함수 안에서 작성하면 된다.

HelloWorld.s14	Result
<pre>main(){ OLED.Print("Hello, World!"); }</pre>	Hello, World!

실행 결과 오른쪽 그림처럼 흰색 글자로 Hello, World가 출력되는 것을 알 수 있다.

■ One-Line Full-Style Code

Full-Style은 중괄호와 세미콜론을 모두 사용하는 특성상 들여쓰기, 줄 바꿈, 띄어쓰기의 제약으로부터 비교적 자유로운 편이다. 따라서 아래와 같이 한 줄로 작성해도 컴파일하는 데 문제없다.

HelloWorld.s14	Result
<pre>main(){OLED.Print("Hello, World!");}</pre>	Hello, World!

실행 결과 오른쪽 그림처럼 흰색 글자로 Hello, World가 출력되는 것을 알 수 있다. 위의 코드와 완전히 동일하기 때문이다.

■ 연산자를 사용하는 경우

Full-Style에서도 연산자를 사용하여 출력할 수 있다. 아래와 같이 `OLED <<`를 사용하여 `Hello, World!`를 출력할 수도 있다.

HelloWorld.s14	Result
<pre>main(){ OLED << "Hello, World!"; }</pre>	Hello, World!

실행 결과 오른쪽 그림과 같이 흰색 글자로 Hello, World가 출력되는 것을 알 수 있다.

■ 명령어를 사용하는 경우

Full-Style에서도 명령어를 사용하여 출력할 수 있다. 아래와 같이 `OLED.Print` 명령어를 사용하여 OLED에 출력하면 된다. 단, [이전 버전\(.s1c\)에서 아래와 같이 작성하면 컴파일 오류가 발생할 수 있으니 유의해야 한다.](#)

HelloWorld.s14	Result
<pre>main(){ OLED.Print "Hello, World!"; }</pre>	Hello, World!

■ 함수를 사용하는 경우

Full-Style에서도 함수를 사용하여 출력할 수 있다. 앞에서의 예제 역시 함수 사용 예시이다. 이번에는 `OLED.Printf`를 사용한 예제인데, 2가지 방법으로 작성할 수 있다.

HelloWorld.s14	Result
<pre>main(){ OLED.Printf("Hello, World!"); }</pre>	Hello, World!
<pre>main(){ OLED.Printf("%s", "Hello, World!"); }</pre>	Hello, World!

실행 결과 둘 다 똑같이 출력된다. 첫 번째 코드는 `format` 문자열에 `Hello, World` 자체를 넣은 것이고, 두 번째 코드는 `format` 문자열에 `%s`를 넣고 그 뒤에 `Hello, World`를 그 자리에 출력한 형태이다.

■ main에서 괄호의 생략

Full-Style에서 `main` 함수를 작성할 때 괄호 안에 아무것도 없으니 괄호를 생략해도 좋다. 따라서 아래와 같이 작성해도 오류 없이 작동한다. 이 코드 역시 Starlit3 컴파일러에서는 제대로 작동하지 않으니 주의해야 한다.

HelloWorld.s14	Result
<pre>main{ OLED.Print("Hello, World!"); }</pre>	Hello, World!

■ main 안에서 문자열을 단독으로 사용한다면?

아래와 같이 `main` 안에서 문자열을 단독으로 사용한다면 어떤 일이 일어날까? 일반적으로 변수나 상수를 단독으로 사용하면 아무 일도 일어나지 않지만, 문자열 리터럴을 단독으로 사용하면 예외적으로 문자열이 빨간색으로 출력된다. 따라서 아래와 같이 작성하면 빨간색으로 `Hello, World!`가 출력된다.

HelloWorld.s14	Result
<pre>main(){ "Hello, World!"; }</pre>	Hello, World!

■ f-문자열의 사용

f-문자열 역시 앞에서 다룬 방법과 똑같이 사용할 수 있다. 다만, 단독으로 사용하면 빨간색으로 출력될 것이다.

HelloWorld.s14	Result
<pre>main(){ OLED.Print(f"Hello, World!"); }</pre>	Hello, World!
<pre>main(){ OLED << f"Hello, World!"; }</pre>	
<pre>main(){ OLED.Print f"Hello, World!"; }</pre>	
<pre>main(){ f"Hello, World!"; }</pre>	Hello, World!

■ 정리

지금까지 배운 내용을 정리하면 다음과 같다.

- **Full-Style의 정의** : 중괄호와 세미콜론을 모두 사용한 스타일로, 들여쓰기의 제약 없이 작성하는 방법.
- **Full-Style의 성질** : Starlit 3의 문법과 거의 동일하게 사용할 수 있다. 단, 명령어 형태로 작성 시 Starlit 3에서는 작동하지 않는다.
- **Full-Style에서 구문 사용** : Full-Style에서는 함수, 명령어, 연산자 형태를 사용해서 Hello, World를 출력할 수 있다. 또한, main 함수의 괄호를 생략해도 상관없다. F-문자열을 사용해도 똑같이 구현할 수 있다.
- **Full-Style에서 단독 구문 사용** : Full-Style에서 문자열 리터럴만 단독으로 쓰고 세미콜론을 붙이면 단독 구문이 되며, 빨간색으로 출력된다.
- **Full-Style에서 Hello, World 코드**

HelloWorld.sl4	Result
<pre>main(){ OLED.Print("Hello, World!"); }</pre>	Hello, World!

203 Starlic-Style로 출력하기

Full-Style은 코드에서 갖춰져야 할 요소인 중괄호와 세미콜론을 모두 사용하는 방식이다. 이제부터는 중괄호와 세미콜론 등을 생략하면서 다른 스타일을 소개하려고 한다.

■ Starlic-Style 소개

Starlic-Style은 **Full-Style**에서 중괄호를 생략한 대신 들여쓰기 규칙을 적용한 방식의 규칙이다. 중괄호를 없애고 들여쓰기 규칙을 강제했기 때문에 **Full-Style**보다 정리되고 간결한 느낌의 코드를 얻을 수 있다. 다만, **Starlic-Style**에서는 세미콜론은 생략하지 않는다.

■ Starlic-Style로 Hello, World 출력하기

아래의 코드는 **Starlic-Style**로 **Hello, World!**를 출력하는 예제이다. 중괄호를 완전히 없애고, 들여쓰기에 신경 써 주면 완성이다.

HelloWorld.s14	Result
<pre>main() OLED.Print("Hello, World!");</pre>	Hello, World!

실행 결과 중괄호를 사용한 **Full-Style**에서의 결과와 완전히 같은 것을 알 수 있다. 중괄호를 없앴을 뿐이지, 실제로는 완전히 같은 코드이기 때문이다.

■ 명령어의 형태 사용하기

아래의 코드는 명령어 형태로 작성한 것이다. **OLED.Print** 명령어를 사용해서 **Hello, World!**를 출력할 수 있다.

HelloWorld.s14	Result
<pre>main() OLED.Print "Hello, World!";</pre>	Hello, World!

■ 연산자 형태 사용하기

다음은 `OLED << 연산자`를 통해 `Hello, World!`를 출력하는 예제이다. 중괄호 없애기와 들여쓰기 규칙만 신경 쓰면 되기 때문에 `Full-Style`과 마찬가지로 문제없이 작동한다.

HelloWorld.s14	Result
<pre>main() OLED << "Hello, World!";</pre>	Hello, World!

■ main 함수에 괄호 생략해도 되나요?

`Starlic-Style` 역시 `Full-Style`과 마찬가지로 `main` 함수에서 괄호를 생략해도 작동하는 데 문제없다.

HelloWorld.s14	Result
<pre>main OLED.Print("Hello, World!");</pre>	Hello, World!

■ 문자열을 단독으로 사용하는 경우

`Starlic-Style`에서도 문자열을 단독으로 사용하면 빨간색 글자로 출력된다. 다만, 주의할 점은 들여쓰기만 잘 지켜주면 된다.

HelloWorld.s14	Result
<pre>main "Hello, World!"</pre>	Hello, World!

■ f-문자열을 사용하는 경우

`Starlic-Style`에서도 `f-문자열`을 그대로 사용할 수 있다. 아래는 `f-문자열`과 함수를 사용해서 `Hello, World!`를 출력하는 예제이다.

HelloWorld.s14	Result
<pre>main OLED.Print(f"Hello, World!");</pre>	Hello, World!

■ Printf 함수를 사용하는 경우

`OLED.Printf` 함수를 사용해서도 출력할 수 있다. 아예 `Hello, World!`를 출력하는 방법이 있고, `%s` 뒤에 문자열을 추가하는 경우도 있다. 후자의 방법을 사용한다면, `f`-문자열 사용이 제한된다.

HelloWorld.sl4	Result
<code>main</code> <code>OLED.Printf("Hello, World!");</code>	Hello, World!
<code>main</code> <code>OLED.Printf("%s", "Hello, World!");</code>	Hello, World!

■ Arduino-Style로 작성하는 경우

`Starlit3`이 `main` 함수만 사용했던 것과는 다르게 `Starlit 4`에서는 `main` 없이 작성하는 것도 가능하고, `main` 함수 안에서 작성할 수도 있다. 또, `Arduino-Style`로 `setup` 함수와 `loop` 함수를 사용해 코드를 만들어도 좋다. 아무 것도 실행하지 않는 함수는 아무것도 작성하지 않아도 된다.

HelloWorld.sl4	Result
<code>setup</code> <code>OLED.Print("Hello, World!");</code> <code>loop</code>	Hello, World!

실행 결과 오른쪽 그림과 같이 흰색 글자로 `Hello, World`가 출력되기는 하지만, `loop`에 의해 프로그램이 종료되지 않고 무한 반복한다. 따라서 프로그램이 종료될 필요가 있다면 `Arduino-Style`은 사용하지 않기로 하자. 물론, `Arduino-Style`이 편할 때도 있으니 적절히 사용할 예정이다.

■ 정리

지금까지 배운 내용을 정리하면 다음과 같다.

- **Starlic-Style의 정의** : Full-Style에서 중괄호를 생략한 대신 들여쓰기 규칙을 적용한 방식.
- **Starlic-Style의 성질** : Starlit 3의 문법에서 중괄호를 모두 없애고, 들여쓰기에 신경 써서 코드를 작성하면 그 코드는 유효하다.
- **Starlic-Style에서 구문 사용** : Starlic-Style에서는 함수, 명령어, 연산자 형태를 사용해서 Hello, World를 출력할 수 있다. 또한, main 함수의 괄호를 생략해도 상관없다. f-문자열도 그대로 사용할 수 있다.
- **단독 구문 사용** : Starlic-Style에서도 문자열 단독 구문을 사용할 수 있다. 다만, 들여쓰기 규칙은 그대로 적용된다. 단독 구문을 사용하는 경우 빨간색으로 출력된다.
- **Arduino-Style에서 구문 사용** : setup과 loop로 프로그램을 만들 수 있지만, loop를 정의하면 프로그램이 종료되지 않고 무한 반복한다.
- **Starlic-Style에서 Hello, World 코드**

HelloWorld.sl4	Result
<pre>main() OLED.Print("Hello, World!");</pre>	Hello, World!

204 Simple-Style로 출력하기

Simple-Style은 최대한 간략한 형태로 코드를 구현하도록 설계된 형태이다. 기본적으로는 Starlic-Style에서 세미콜론만 뺀 형태지만, 더 간략하게 표현해볼 수도 있다.

■ Simple-Style 소개

Simple-Style은 Starlic-Style에서 세미콜론을 모두 없애 버린 형태에서 시작한다. 이번 장에서는 단순히 세미콜론을 없애는 것을 넘어서서 가장 간단하게 Hello, World를 출력하는 방법을 고민해 보기로 한다.

■ Simple-Style로 Hello, World! 출력하기

Simple-Style에서 Hello, World!를 출력하는 코드는 아래와 같이 작성할 수 있다. 아래 코드는 가장 기본적인 형태의 Simple-Style로, 흰색 글자로 Hello, World가 출력된다.

HelloWorld.s14	Result
<pre>main() OLED.Print("Hello, World!")</pre>	Hello, World!

■ 명령어 형태로 출력하기

Simple-Style에서 명령어 형태의 출력은 위 구문에서 괄호만 지우면 끝이다. 물론, 다른 스타일에서도 괄호만 지워서 출력했었다. 괄호를 지울 때 띄어쓰기로 한 칸 띄워주어야 오류가 나지 않는다.

HelloWorld.s14	Result
<pre>main() OLED.Print "Hello, World!"</pre>	Hello, World!

■ main 함수의 괄호 없애기

Simple-Style에서도 `main` 함수 뒤에 있는 괄호를 없애 버릴 수 있다. 따라서 아래와 같이 아주 간단한 코드를 만들 수 있다.

HelloWorld.sl4	Result
<code>main</code> <code>OLED.Print "Hello, World!"</code>	Hello, World!

■ main 함수 없애기

`main` 함수조차 지우고 `OLED.Print`만 써서 코드를 작성해도 실행하는데 문제는 없다. 단, 전역변수가 추가되어 있다면 순서에 유의해야 한다.

HelloWorld.sl4	Result
<code>OLED.Print "Hello, World!"</code>	Hello, World!

■ OLED.Print 없애기

이젠 다 지우고 `Hello, World!`만 남겨두었다. 그래도 실행은 된다. 문제는 글자 색, 단독 구문이므로 빨간색으로 출력될 것이다. Simple해졌지만 약간 출력이 달라진 것을 알 수 있다.

HelloWorld.sl4	Result
<code>"Hello, World!"</code>	Hello, World!

자, 여기서도 흰색으로 출력할 기회는 얼마든지 있다. 글자 색을 흰색으로 강제로 바꾸면 되기 때문이다. 아래와 같이 `/w`를 문자 시작 부분에 추가하자. 그러면 흰색으로 돌아온다. 글자 색 설정 부분은 뒤에서 자세히 다룬다.

HelloWorld.sl4	Result
<code>"/wHello, World!"</code>	Hello, World!

■ f-문자열 사용하기

Simple Style에서도 f-문자열을 사용해 볼 수 있다. 놀랍게도 f-문자열 그 자체가 포매팅을 직관적으로 할 수 있게 해주는 방식이라서 더 간단해지는 효과도 있다. 일단 사용법은 완전히 똑같다.

HelloWorld.s14	Result
main() OLED.Print(f"Hello, World!")	Hello, World!
main() OLED.Print f"Hello, World!"	
main OLED.Print f"Hello, World!"	
OLED.Print f"Hello, World!"	
f"/wHello, World!"	

■ << 연산자 사용하기

이번에는 OLED <<를 사용해서 Hello, World를 출력하는 방법을 다뤄 보겠다. 역시 앞에서와 같은 방법으로 점점 간단해지는 순서대로 코드를 설계하면 아래와 같다. 가장 일반적으로 사용하는 Hello, World 출력이다.

HelloWorld.s14	Result
main() OLED << "Hello, World!"	Hello, World!
main OLED << "Hello, World!"	
OLED << "Hello, World!"	

■ 정리

지금까지 배운 내용을 정리하면 다음과 같다.

- **Simple-Style의 정의** : Starlic Style에서 세미콜론을 제외한 형태의 코드이며, 넓게는 괄호를 최소화하여 최대한 간결한 형태로 표현한 스타일을 의미한다.
- **Simple-Style의 성질** : Starlic Style에서 세미콜론을 제외해도 유효한 코드가 된다.
- **Simple Style에서 구문 사용** : Starlic Style과 동일하게 사용할 수 있다. 연산자, 함수, 명령어 형태 모두 사용할 수 있다.
- **main 함수의 생략** : main 함수 없이 전역변수 영역에서 작성해볼 수 있다. 다만, 전역변수가 있다면 순서에 유의해서 작성한다.
- **Simple Style 코드 예**

HelloWorld.sl4	Result
<pre>main() OLED << "Hello, World!"</pre>	Hello, World!
<pre>main OLED << "Hello, World!"</pre>	
<pre>OLED << "Hello, World!"</pre>	

205 Pythonic-Style로 출력하기

■ Pythonic-Style 소개

Pythonic-Style은 Python 특유의 `` 붙이면서 구문을 시작하는 형태를 빌려 온 형태이다. Starlic, Simple 스타일에서는 암묵적으로 구문을 분석해야 해서 컴파일러의 처리 양이 많지만, 이 방식을 사용하면 컴파일러의 처리 양이 줄어든다. 물론 컴파일 속도가 유의미하게 차이가 나는 것은 아니다. 따라서 중괄호가 있어야 할 위치에 :로 끝내 주면 끝이다.

Pythonic Style이라고 해서 함수 정의를 def로 시작하거나 하지는 않는다. Python은 인터프리터 언어인 특성상 def는 명령어가 되고 이 구문을 수행하면 함수 전체를 추가하게 되는 셈이다. 그리고 함수를 호출하면 저장됐던 구문이 실행될 뿐이다. 하지만 Starlit은 컴파일러가 함수를 저장하기 때문에 def를 넣을 필요가 없다.

■ Pythonic-Style로 Hello, World 출력하기

코드는 매우 간단하다. Simple 스타일에서 중괄호가 있어야 할 위치에 :를 붙이면 끝이기 때문이다. 실행 결과도 흰색 글자로 Hello, World가 출력된다.

HelloWorld.sl4	Result
<pre>main(): OLED.Print("Hello, World!")</pre>	Hello, World!

■ 괄호의 생략

역시 Simple Style에서 했던 괄호의 생략 역시 똑같이 적용할 수 있다.

HelloWorld.sl4	Result
<pre>main(): OLED.Print "Hello, World!"</pre>	Hello, World!
<pre>main: OLED.Print "Hello, World!"</pre>	
<pre>main: OLED.Print("Hello, World!")</pre>	

■ 세미콜론 추가

Full-Pythonic Style이라고 해서 세미콜론을 모두 추가하는 스타일도 사용할 수는 있다. 즉, 세미콜론의 추가는 자유다.

HelloWorld.sl4	Result
<pre>main(): OLED.Print("Hello, World!");</pre>	Hello, World!

세미콜론을 추가했다고 해서 괄호를 꼭 써야하는 것도 아니라서 아래와 같이 작성해도 상관없다.

HelloWorld.sl4	Result
<pre>main(): OLED.Print "Hello, World!";</pre>	Hello, World!
<pre>main(): OLED.Print "Hello, World!" ;</pre>	
<pre>main: OLED.Print "Hello, World!";</pre>	
<pre>main: OLED.Print("Hello, World!");</pre>	

■ OLED << 연산자 사용하기

Pythonic Style에서도 연산자를 사용해 **Hello, World**를 출력할 수 있다. 역시나 세미콜론은 자유롭게 사용해도 좋다.

HelloWorld.sl4	Result
<pre>main: OLED << "Hello, World!"</pre>	Hello, World!
<pre>main: OLED << "Hello, World!";</pre>	

■ f-문자열 사용하기

Pythonic Style에서 잘 어울리는 것이 있다면 f-문자열이다. 역시 **OLED << 연산자**나 **OLED.Print** 함수에서 사용할 수 있다.

HelloWorld.sl4	Result
<pre>main: OLED.Print(f"Hello, World!")</pre>	Hello, World!
<pre>main: OLED << f"Hello, World!"</pre>	

■ Printf 사용하기

Pythonic Style에서도 `OLED.Printf` 함수를 사용할 수 있다. 앞에서 다룬 Style에서와 사용 방법은 같다.

HelloWorld.sl4	Result
<pre>main: OLED.Printf("Hello, World!")</pre>	Hello, World!
<pre>main: OLED.Printf("%s", "Hello, World!")</pre>	

■ 단독 구문 사용하기

Pythonic Style에서도 단독으로 구문을 사용할 수 있다. 역시 출력은 빨간 글자로 출력되니 흰색 글자로 출력하려면 `/w`를 붙이면 된다.

HelloWorld.sl4	Result
<pre>main: "Hello, World!"</pre>	Hello, World!
<pre>main: "/wHello, World!"</pre>	Hello, World!

■ 정리

지금까지 배운 내용을 정리하면 다음과 같다.

- **Pythonic-Style의 정의** : Python의 구문의 형식을 빌려 중괄호 대신 :를 사용한 스타일.
 - ▶ Simple-Pythonic : Simple Style에서 중괄호가 있어야 할 부분에 :를 붙인 형태로, 세미콜론을 사용하지 않는다.
 - ▶ Full-Pythonic : 모든 구문에 세미콜론을 붙인 형태
 - ▶ 어떤 구문에는 세미콜론을 붙이고 어떤 구문에는 세미콜론을 붙이지 않아도 상관없다.
- **Pythonic-Style의 성질** : Simple Style에서 중괄호가 들어가는 부분에 :를 붙이면 Pythonic이 되며, Simple Style, Starlic Style과 혼용해도 된다.
- **Pythonic Style에서 구문 사용** : Simple Style 또는 Starlic Style과 동일하게 사용할 수 있다. 연산자, 함수, 명령어 형태 모두 사용할 수 있다.

● Pythonic Style 코드 예

HelloWorld.sl4	Result
<pre>main(): OLED.Print("Hello, World!")</pre>	Hello, World!
<pre>main(): OLED.Print("Hello, World!");</pre>	

206 GUE-Style로 출력하기

■ 표를 사용한 프로그래밍 발상

지금까지는 텍스트 형태로 프로그래밍을 해 왔다. 하지만, 들여쓰기를 적절히 사용하고 괄호를 생략하다 보면 이런 생각이 들 수도 있다. 표로 작성할 수 있지 않을까? 실제로 **Starlit 1**의 유일한 장점이었다면 표로 작성하기 적절한 형태로 구성된 점이다. 띄어쓰기로 구문을 구성할 수 있고, 띄어쓰기 여러 개도 한 개로 줄여주는 특성이 있어 표로 프로그래밍을 할 수 있다고 생각을 해 보았다.

start>>			
	Print	OLED	"Hello, World!"
end			

■ GUE-Style

GUE Style은 **Starlit 1**의 표현법을 재해석해서 그대로 가져와 사용하는 방식이다. **Starlit 1**과의 차이점이라면 변수명을 자유롭게 사용할 수 있고, 지역변수 사용이 허용되는 점, 그리고 변수명 앞에 `_`를 붙이면 Reference가 된다는 점이다. 하지만 **Starlit 1**에서는 띄어쓰기로 인수를 구분한 것을 그대로 반영했다. 그래서 반점과 괄호를 최소화하여 코드를 작성할 수 있다.

■ GUE-Style로 Hello, World 출력하기 - Print OLED

GUE Style로 **Hello, World**를 출력하는 방법은 위의 표대로 작성하는 **Print OLED** 방식이다. **OLED.Print**와 완전히 같지만, **.**을 생략하는 대신 함수 명칭을 앞으로 댕겨서 사용한다.

HelloWorld.sl4	Result
start>> Print OLED "Hello, World!" end	Hello, World!

■ GUE-Style로 Hello, World 출력하기 - OLED.Print

GUE Style로 Hello, World를 출력하기 위해 **OLED.Print**를 그대로 사용해도 된다. 단, 앞서와 다르게 명령어는 Print가 아닌 **OLED.Print**가 된다는 차이점이 있다. 그래서 표로 정리할 때는 아래와 같아진다.

start>>		
	OLED.Print	"Hello, World!"
end		

이 표의 내용을 그대로 정리하여 코드를 작성하면 아래와 같아진다.

HelloWorld.s14	Result
start>> OLED.Print "Hello, World!" end	Hello, World!

■ GUE-Style로 Hello, World 출력하기 - 괄호를 추가하는 경우

GUE Style에서도 괄호를 사용해서 코드를 작성해도 상관없다. 하지만, 표로 정리할 때는 하나의 구문이 통째로 들어간다는 차이점이 있다.

start>>	
	OLED.Print ("Hello, World!")
end	

이 표의 내용을 그대로 정리하여 코드를 작성하면 아래와 같아진다.

HelloWorld.s14	Result
start>> OLED.Print ("Hello, World!") end	Hello, World!

■ GUE-Style로 Hello, World 출력하기 - 연산자를 사용하는 경우

GUE Style에서도 연산자를 써도 좋다. 다만, 연산자 앞뒤로는 띄어쓰기가 유효하지 않아 표로 정리할 때는 약간 달라진다.

start>>	
	OLED << "Hello, World!"
end	

이 표의 내용을 그대로 정리하여 코드를 작성하면 아래와 같아진다.

HelloWorld.sl4	Result
start>> OLED << "Hello, World!" end	Hello, World!

연산자, 괄호 추가도 허용하는 이유는 띄어쓰기만 사용해 구분하는 데에는 한계가 있기 때문이다. 물론 취향 차이긴 하지만 출력이 있는 함수의 출력을 변수에 모두 저장해 놓고 괄호를 완전히 사용하지 않고 프로그래밍해도 상관없다. 실제로 Starlit 1, 2까지는 이렇게 작성하는 것이 공식 규칙이었다.

GUE에서 연산자 앞, 뒤로는 띄어쓰기가 무시된다. 다만, 연산자 앞은 띄고 뒤를 붙이면서 단항 연산자로 사용할 가능성이 있다고 판단될 때 한해서는 앞쪽 띄어쓰기는 유효해질 수 있다. 이 부분에 대해서는 연산자 Part에서 자세히 다룬다.

■ GUE-Style로 Hello, World 출력하기 - end 없애기

GUE Style에서는 함수의 끝은 end를 넣기도 하였다. 하지만 이것은 언제까지나 Starlit 1의 잔재이며, end는 생략해도 된다(Starlit 1에서 end는 필수사항이었다.). 따라서 end를 생략하여 작성하면 아래와 같다.

start>>			
	Print	OLED	"Hello, World!"

HelloWorld.sl4	Result
start>> Print OLED "Hello, World!"	Hello, World!

■ GUE-Style에서 f-문자열 사용

GUE Style에서도 f-문자열을 사용할 수 있다. 문자열이 들어가는 자리에 f를 붙이면 되기 때문이다. 아래 예제는 f-문자열을 사용하여 Hello, World를 출력하는 예제이다.

start>>			
	Print	OLED	f"Hello, World!"

HelloWorld.sl4		Result
start>>		Hello, World!
Print OLED f"Hello, World!"		

■ GUE-Style에서 Printf 사용

GUE Style에서도 Printf를 사용할 수 있다. Printf OLED로 작성해도 좋고, OLED.Printf로 작성해도 좋다. 역시 계속 나열하는 방식으로 작성하면 되므로 2가지 방법으로 프로그래밍하면 된다. Printf OLED를 사용해 2가지 방법으로 출력하는 예제이다.

start>>			
	Printf	OLED	"Hello, World!"

HelloWorld.sl4		Result
start>>		Hello, World!
Printf OLED "Hello, World!"		

start>>				
	Printf	OLED	"%s"	"Hello, World!"

HelloWorld.sl4		Result
start>>		Hello, World!
Printf OLED "%s" "Hello, World!"		

■ GUE-Style에서 단독 구문 사용

GUE Style에서도 단독 구문을 사용할 수 있다. 이 경우 앞에서처럼 빨간색 글자로 출력된다. 표로 나타냈을 때는 단독 구문으로 처리된다.

start>>	
	"Hello, World!"

HelloWorld.sl4	Result
start>> "Hello, World!"	Hello, World!

■ GUE-Style에서 함수 정의

GUE Style에서의 함수 정의는 >>로 끝나도 되지만 >로 시작해도 된다. 따라서 아래의 코드도 유효하다. >로 시작하는 경우는 인수가 있을 때 많이 사용할 것이다(표로 정리하기에는 >로 시작하는 쪽이 깔끔하기 때문이다.).

>start			
	Print	OLED	"Hello, World!"

HelloWorld.sl4	Result
>start Print OLED "Hello, World!"	Hello, World!

■ 정리

GUE Style은 Starlit 1의 장점을 최대한 부각하여 표로 나타낸 것과 비슷한 형태로 프로그래밍할 수 있게 설계된 스타일이다. Starlit 1에 기원을 두고 있으며, Starlit 3에서도 한시적으로 GUE라는 언어로 사용할 수 있게 하기도 했다. Starlit 4에서는 하나의 코드 스타일로 자리 잡았다고 볼 수 있다.

- **GUE Style의 특징** : 명령어 인수1 인수2의 형태로 띄어쓰기로 인수를 구분하는 특징이 있다.
- **GUE Style로의 변환** : ① 도치 규칙으로 OLED.Print 구문을 Print OLED로 명령어로 써도 된다. ② OLED.Print 자체를 명령어로 써도 된다. ③ 함수를 정의할 때는 뒤에 >>를 붙이거나 앞에 >를 붙인다. ④ 맨 끝에 end는 붙여도 되고 안 붙여도 된다.
- **GUE Style에서 Hello, World 출력 예제**

HelloWorld.sl4	Result
<pre>start>> Printf OLED "Hello, World!"</pre>	Hello, World!

207 Pseudo-GUE-Style로 출력하기

■ Pseudo-GUE-Style

GUE Style은 표로 나타낸 듯한 형태로 나타낼 수 있고, 궁극적으로는 괄호를 최소화해 깔끔한 명령어를 표현할 수 있었다. GUE-Style이 아닌 상황에서 GUE의 형태를 살리고 싶을 때가 있을 수도 있겠다. 이때는 Pseudo-GUE-Style로 최대한 유사하게 표현해볼 수도 있다.

■ Print OLED를 사용하는 방법

GUE 스타일 외에서 Print OLED를 사용하려면 반점을 아래의 코드처럼 찍어 주어야 정상 작동한다. 이렇게 해야 하는 이유는 OLED.Print(는 Print(OLED,와 등가이기 때문이다.

HelloWorld.sl4	Result
main Print OLED, "Hello, World!"	Hello, World!
main Print OLED, "Hello, World!";	
main{ Print OLED, "Hello, World!"; }	
main: Print OLED, "Hello, World!"	

■ Printf OLED를 사용하는 방법

GUE 스타일 외에서 Printf OLED를 사용하려면 Printf와 마지막 인수를 제외한 나머지 인수 뒤에 반점을 붙인다. 명령어는 반점을 붙이지 않고 인수만 반점을 붙이고 마지막은 끝을 의미하니 반점을 붙이지 않아도 되는 원리라고 생각하면 쉽다.

HelloWorld.sl4	Result
main Printf OLED, "%s", "Hello, World!"	Hello, World!
main{ Printf OLED, "%s", "Hello, World!"; }	
main: Printf OLED, "%s", "Hello, World!"	

■ \$>>를 사용하는 방법

한 줄만 GUE 스타일을 빌려 쓰고 싶다면 \$>> 뒤에 작성하면 된다. \$>>가 그 줄만 GUE Style로 바꾸라는 의미기 때문이다. 물론 GUE 스타일에서는 쓰지 않는다. 특히하게도 3번째 구문이 주목할 만한데, Full-Style임에도 세미콜론을 붙이지 않은 점이다. 한 줄만 GUE Style로 바뀌어서 쓰기 때문에 세미콜론을 붙이지 않아도 오류가 나지 않는다.

HelloWorld.sl4	Result
main	Hello, World!
\$>> Print OLED "Hello, World!"	
main{	
\$>> Print OLED "Hello, World!";	
}	
main{	Hello, World!
\$>> Print OLED "Hello, World!"	
}	
main:	
\$>> Print OLED "Hello, World!"	

■ 괄호를 추가하는 경우

사실 Print OLED에서 명령어는 Print까지이므로 Print 뒤부터 맨 끝까지 괄호를 친 것이나 다름없다. 따라서 아래와 같이 괄호를 치고 프로그래밍해도 정상 작동한다.

HelloWorld.sl4	Result
main	Hello, World!
Print(OLED, "Hello, World!")	

■ 정리

GUE Style이 아님에도 GUE Style과 비슷한 형태로 코드를 작성할 수 있는데, 이것을 Pseudo-GUE Style이라고 한다.

- 일반적인 Style에서 사용할 때 : 반점을 명령어와 끝 인수를 제외하고 모두 추가한다.

HelloWorld.sl4	Result
<pre>main Print OLED, "Hello, World!"</pre>	Hello, World!

- 부분적으로 GUE Style을 사용할 때 : 구문 앞에 \$>>를 붙여서 작성한다.

HelloWorld.sl4	Result
<pre>main \$>> Print OLED "Hello, World!"</pre>	Hello, World!

- 괄호를 추가하는 경우 : 명령어까지가 함수명이 될 수 있다. OLED.Print에서는 OLED.Print가 명령어지만 Print OLED는 Print까지만 명령어에 해당된다. 도치와 등가 관계는 맞지만, 명령어의 범위가 다름에 유의하자.

HelloWorld.sl4	Result
<pre>main Print(OLED, "Hello, World!")</pre>	Hello, World!

208 들여쓰기 규칙과 Hello, World!

■ 들여쓰기 규칙이 적용되는 스타일과 그 규칙

쉽게 말해서 들여쓰기 규칙은 중괄호를 쓰지 않는 한 모두 적용된다. 그러므로 중괄호를 사용하는 Full-Style을 제외하고는 전부 들여쓰기 규칙이 적용된다. 지금까지 들여쓰기는 탭 하나 또는 띄어쓰기 4칸을 사용했다. Starlit은 Python의 들여쓰기 규칙을 벤치마킹해서 둘 중 어느 하나를 써도 좋다. 하지만, 띄어쓰기에 한해서는 약간 널널한 규칙이 있다. 띄어쓰기 횟수가 4칸일 필요 없이 3칸이든 2칸이든 들여 쓴 위치만 맞으면 컴파일하는 데 문제없다. 따라서 아래와 같이 2칸씩만 띄고 코드를 작성해도 유효하다.

HelloWorld.sl4	Result
<pre>main OLED << "Hello, World!"</pre>	Hello, World!
<pre>main OLED << "Hello, World!"</pre>	
<pre>main OLED << "Hello, World!"</pre>	
<pre>main OLED << "Hello, World!"</pre>	
<pre>main OLED << "Hello, World!"</pre>	

위의 코드는 모두 결과가 같다. 그 이유는 들여쓰기 띄어쓰기 횟수에 상관없이 들여쓰기 했다는 것만 인정되면 유효하기 때문이다.

■ 줄 표시도 컴파일에 문제 없다.

들여쓰기 규칙이 적용되는 스타일에 한해서 `|`를 붙여 줄을 표시해도 컴파일하는데 문제가 생기지 않는다. 이런 기능까지 추가한 이유는 **if문**의 범위가 어디까지인지 쉽게 인지하게 하기 위해서이다. 주의할 점이라면 `|`를 붙이는 위치인데, 반드시 첫 번째 칸에 써야 한다. 실제로 로봇이 코드를 만드는 것이 아니라면 이런 규칙은 거의 사용하지 않을 것이다.

HelloWorld.sl4	Result
<pre>main OLED << "Hello, World!"</pre>	Hello, World!

■ 줄을 바꾼다면 추가적인 들여쓰기를 해야 한 구문으로 인식한다.

OLED << 뒤의 내용을 다음 줄에 작성하려면 들여쓰기를 한 번 더 해줘야 하나의 구문으로 인식한다. 들여쓰기를 지키지 않으면 컴파일 오류가 날 수 있으니 주의해야 한다. 물론 괄호가 닫히지 않았다면 오류 없이 작동하기도 한다.

HelloWorld.sl4	Result
<pre>main OLED << "Hello, World!"</pre>	Hello, World!

■ 정리

들여쓰기 규칙은 Python에서 따 왔지만, 들여쓰기에 사용되는 띄어쓰기 횟수는 제한을 두지 않고, 일관성 있게만 작성하면 정상 동작한다.

- **`|` 표시** : 사람이 프로그래밍할 때는 잘 쓰지 않겠지만, 중괄호의 범위를 쉽게 인식하기 위해 사용해도 좋다. 단, Full-Style에서는 사용할 수 없다.
- **줄 바꿈** : 하나의 구문이 너무 길다면 줄 바꿈을 해주어도 좋다.

209 문자열 변수를 사용하는 방법

■ f-문자열을 이용한 변수 정의

가끔은 직접 출력하지 않고 변수를 만들어서 출력하기도 한다. Starlit 역시 문자열 변수를 만들고 출력해 볼 수 있다. 문자열 변수를 정의할 때는 **f-문자열**으로 정의해야 한다. **f-문자열**을 사용하지 않으면 문자열 수정이 불가능하다.

HelloWorld.sl4	Result
<pre>main st = f"Hello, World!" OLED << st</pre>	Hello, World!

■ 단일 배열은 변수!

Starlit에서는 단일 배열은 하나의 변수로 취급하게 설계했다. 따라서 아래와 같이 단일 배열 형태로 정의하여 **Hello, World!**를 출력할 수 있다.

HelloWorld.sl4	Result
<pre>main st = ["Hello, World!"] OLED << st</pre>	Hello, World!

■ 선언 - 대입 후 출력

한편, 선언 - 대입 후 출력 방법으로 **Hello, World!**를 출력하는 코드를 구현할 수도 있다. 아래와 같이 작성해 보자.

HelloWorld.sl4	Result
<pre>main st = str ///선언 st = "Hello, World!" ///대입 OLED << st</pre>	Hello, World!

■ GUE Style에서의 사용

GUE Style에서는 변수 정의 방법이 약간 특이한 점을 주목해 볼 필요가 있다. 변수 앞에 `_`를 붙이면 변수 정의, 대입으로 인식하기 때문이다.

>start			
	<code>_st</code>	<code>str</code>	
	<code>_st</code>	<code>"Hello, World!"</code>	
	<code>Print</code>	<code>OLED</code>	<code>st</code>

HelloWorld.sl4		Result
<pre>>start _st str ///선언 _st "Hello, World!" ///대입 Print OLED st</pre>		Hello, World!

한편, 선언과 동시에 대입을 하여 아래와 같이 작성해도 좋다.

>start			
	<code>_st</code>	<code>str</code>	<code>"Hello, World!"</code>
	<code>Print</code>	<code>OLED</code>	<code>st</code>

HelloWorld.sl4		Result
<pre>>start _st str "Hello, World!" ///선언과 동시에 대입 Print OLED st</pre>		Hello, World!

■ 정리

- 문자열 변수를 정의하는 방법 : f-문자열을 사용하거나 단일 배열로 문자를 추가하여 정의한다.
- GUE Style에서 : `_st`와 같은 구문을 사용하면 선언, 대입 등을 할 수 있다.

3. 변수와 연산자

301 Starlit에서의 변수와 연산

■ Starlit에서의 자료형

Starlit 언어는 원래 정수형 자료형만 지원했지만, **Class**라는 개념과 연산자 오버로딩을 도입하여 소수형 자료도 사용할 수 있게 하였다. 따라서 Starlit은 아래와 같은 자료형이 존재한다.

Object	Type	설명	주요 리터럴/ 기본 표기법
Pointer	ref	변수의 주소 값	<code>_a</code> , <code>_b</code> , ... f 없는 문자열 (ex: <code>"Hello, World!"</code>)
Number	int	정수(32비트)	0, 1, 2, ...
	float	소수(32비트)	0.0, 1.0, 2.0, ..
	complex_t	복소수(64비트)	<code>j</code> , <code>2+j*1</code> , ...
String	str	문자열(256바이트)	f-string (ex: <code>f"Hello, World!"</code>)
Display	OLED_t	OLED 객체	<code>OLED_Black()</code>

Starlit 자료형은 위 표와 같이 존재하지만, 사용할 때 유의사항이 몇 가지 존재한다.

첫째, 문자열을 정의할 때 **f-문자열**을 사용하지 않으면 문자열 객체가 아닌 레퍼런스로 정의해서 문자열을 편집할 수 없게 된다(출력은 가능하다.).

둘째, **j**는 복소수에서 허수단위를 의미하기 때문에 다른 용도로 사용할 수 없다. 따라서 실수로 **for문**을 만들 때 사용하지 않도록 주의해야 한다.

마지막으로 강조하는 것은 변수 작성법이다. Starlit에서 **_**로 시작하는 것은 레퍼런스를 의미하므로 변수 이름은 숫자나 **_**로 시작하지 않는 것으로 통일한다. **Class**와 함수 이름도 마찬가지이다. 또, 오류의 여지를 막기 위해 **_**로 끝나는 것도 피해야 한다.

■ Starlit에서의 연산자

Starlit의 연산자는 아래 표와 같이 지원한다. 이중 괄호와 객체 참조, 반복 연산, 그리고 반점과 대입을 제외한 연산자는 모두 오버로딩할 수 있다. 모든 연산자는 오버로딩으로 정의되어 있는데, 정수와 소수는 다수의 연산자가 정의되어 있어 자유롭게 사용할 수 있다. 3장에서는 정수와 소수의 사칙연산만을 중점적으로 다루며, 두 수를 받아 합, 차, 곱, 몫과 나머지를 출력하는 코드를 다양하게 구현해 보겠다.

Priority	Operator	C	Starlit
0	Bracket	() []	() [] ²⁾
0.5	Object/Struct	. ->	. ³⁾
1	Iterator ⁴⁾	Does Not Exist	:
2	Unary Operators(단항)	! ~ * & - + ++ --	! ~ * - + ⁵⁾
3	Exponent	Does Not Exist	^^
4	Multiplications	* / %	* / % ** // %%
6	Additions	+ -	+ -
7	Bit Shift	<< >>	<< >> <<< >>>
8	Comparing Operators	< <= > >=	< <= > >=
9		== !=	== != === !==
10	Bit Logical Operators	&	&
11		^	^
12			
13	Logical Operators	&&	&&
14			
16	Substitutions	= += -= *= /= ...	= += -= *= /= ...
16.5	Iterators(for/case)	Does Not Exist	:
17	Comma	,	,

2) 배열에서 사용하는 전용 괄호

3) Starlit에서는 레퍼런스형도 `.`을 사용하여 참조한다.

4) Case/For문 등 중괄호 필요 구문에서는 우선순위가 마지막.

5) Starlit에서는 포인터 개념이 없어서 *, & 연산 자체가 존재하지 않는다.

302 Full-Style과 연산 코드

여기서부터는 두 정수를 입력받아 합, 차, 곱, 몫, 나머지를 구하고, 두 소수를 입력받아 합, 차, 곱, 몫을 구하는 코드를 구현하겠다.

■ 두 정수의 연산 결과 출력하기

변수 `a`, `b`를 `int`형으로 정의한 후 `OLED` 객체로부터 값을 받아 결과를 출력할 수 있다. 아래와 같이 작성해 보자.

Operation.sl4	Result
<pre>main(){ a = int; b = int; "a 입력 : " >> OLED >> a << "\n"; "b 입력 : " >> OLED >> b << "\n"; OLED << "합 : %d\n" % (a + b); OLED << "차 : %d\n" % (a - b); OLED << "곱 : %d\n" % (a * b); OLED << "몫 : %d\n" % (a // b); OLED << "나머지 : %d\n" % (a %% b); }</pre>	<pre> 합 : 20 차 : 10 곱 : 30 몫 : 18 나머지 : 200 나머지 : 2 나머지 : 0</pre>

코드를 보면 알겠지만 통상적인 나눗셈 연산 대신 `//`와 `%%`를 사용한 것을 확인할 수 있는데, C언어의 몫과 나머지는 수학에서 말하는 몫과 나머지와 다르게 출력돼서 보정한 것이다. 양수를 나누는 경우 상관없지만, 음수를 나누는 경우 결과가 달라진다.

■ 두 소수의 연산 결과 출력하기

변수 `a`, `b`를 `float`형으로 정의한 후 `OLED` 객체로부터 값을 받아 결과를 출력할 수 있다. 아래와 같이 작성해 보자. 정수와는 다르게 나눗셈 연산은 `'/'`만 사용하므로 주의해야 한다.

Operation.s14	Result
<pre>main(){ a = float; b = float; "a 입력 : " >> OLED >> a << "\n"; "b 입력 : " >> OLED >> b << "\n"; OLED << "합 : %.2f\n" % (a + b); OLED << "차 : %.2f\n" % (a - b); OLED << "곱 : %.2f\n" % (a * b); OLED << "몫 : %.2f\n" % (a / b); }</pre>	<pre>a 입력 : 2.0 b 입력 : 1.5 합 : 3.50 차 : 0.50 곱 : 3.00 몫 : 1.33</pre>

■ 구문 분석하기

● 변수의 정의

``a = int;``, ``b = float;``와 같이 표현함으로써 변수 `a`, `b`를 각각 정수형, 소수형으로 선언할 수 있다. 앞에서도 잠시 다루었겠지만 선언은 (변수이름) = (자료형) 꼴로 나타내면 된다. 문자열도 ``st = str`` 꼴로 선언했다.

● 사용자로부터 입력받기

이 장에서 처음 등장하는 표현이라면 사용자로부터 값을 입력받는 것이다. `scanf`와 묘하게 비슷하지만, ``"a 입력 : " >> OLED >> a << "\n";`` 표현은 사용자로부터 값을 입력받기 전 안내와 입력받은 후 줄 바꿈 동작까지 정의할 수 있어 유연하게 쓰일 수 있다. `C++`조차도 한 번에 입력과 출력이 불가능한 것을 개선한 것이다.

● OLED에 포맷 출력하기(1)

이 장에서 처음 등장한 표현인데, `%` 연산자를 사용해 숫자를 넣는 것이다. ``%d``는 정수를, ``%f``는 소수를, ``%.2f``는 소수점 아래 2자리까지 출력하는 의미이다. 정수도 마찬가지로 ``%2d``를 쓰면 2자리를 차지하게 출력하고 ``%02d``는 2자리를 출력하되 한 자리수면 앞에 0을 붙인다는 의미가 된다. `C`언어와 용법이 같다.

■ 다른 방법으로 출력하기 - f-문자열

이번에는 **f-문자열**을 사용해서 코드를 작성해 보겠다. % 연산자보다 훨씬 간단하게 사용할 수 있다. 여담이지만, 두 기능 모두 Python에서 벤치마킹하였다.

● 정수 출력하는 경우

Operation.sl4	Result
<pre>main(){ a = int; b = int; "a 입력 : " >> OLED >> a << "\n"; "b 입력 : " >> OLED >> b << "\n"; OLED << f"합 : {a+b}\n"; OLED << f"차 : {a-b}\n"; OLED << f"곱 : {a*b}\n"; OLED << f"몫 : {a//b}\n"; OLED << f"나머지 : {a%b}\n"; }</pre>	<pre>a : 20 b : 10 합 : 30 차 : 10 곱 : 200 몫 : 2 나머지 : 0</pre>

● 소수 출력하는 경우 : `:.2f`를 붙여야 소숫점 아래 2자리까지 출력할 수 있다.

Operation.sl4	Result
<pre>main(){ a = float; b = float; "a 입력 : " >> OLED >> a << "\n"; "b 입력 : " >> OLED >> b << "\n"; OLED << f"합 : {a+b:.2f}\n"; OLED << f"차 : {a-b:.2f}\n"; OLED << f"곱 : {a*b:.2f}\n"; OLED << f"몫 : {a/b:.2f}\n"; }</pre>	<pre>a : 2.0 b : 1.5 합 : 3.50 차 : 0.50 곱 : 3.00 몫 : 1.33</pre>

사용 방법은 % 연산자를 사용했을 때와 비슷하지만, `%`를 사용하지 않고 `:`를 사용한다.

■ 다른 방법으로 출력하기 - Printf 함수

이번에는 **Printf** 함수를 사용하는 방법을 소개한다. **OLED.Printf** 함수에서도 **%d**, **%.2f**를 사용해서 계산 결과를 출력해볼 수 있다. **GUE Style**에서는 이 방법을 주로 사용하니 잘 익혀두자.

● 정수 출력하는 경우

Operation.sl4	Result
<pre>main(){ a = int; b = int; "a 입력 : " >> OLED >> a << "\n"; "b 입력 : " >> OLED >> b << "\n"; OLED.Printf("합 : %d\n", a+b); OLED.Printf("차 : %d\n", a-b); OLED.Printf("곱 : %d\n", a*b); OLED.Printf("몫 : %d\n", a//b); OLED.Printf("나머지 : %d\n", a%b); }</pre>	<pre>a 입력 : 20 b 입력 : 10 합 : 30 차 : 10 곱 : 200 몫 : 2 나머지 : 0</pre>

● 소수 출력하는 경우 : `%.2f`를 붙여야 소숫점 아래 2자리까지 출력할 수 있다.

Operation.sl4	Result
<pre>main(){ a = float; b = float; "a 입력 : " >> OLED >> a << "\n"; "b 입력 : " >> OLED >> b << "\n"; OLED.Printf("합 : %.2f\n", a+b); OLED.Printf("차 : %.2f\n", a-b); OLED.Printf("곱 : %.2f\n", a*b); OLED.Printf("몫 : %.2f\n", a/b); }</pre>	<pre>a 입력 : 2.0 b 입력 : 1.5 합 : 3.50 차 : 0.50 곱 : 3.00 몫 : 1.33</pre>

사용 방법은 % 연산자와 똑같지만, % 연산자와는 다르게 하나의 인수로 값을 주는 차이점이 있다.

■ 정리

● 필수 구문

- ▶ ``a = int;`` - 변수를 선언할 때는 (변수명) = (자료형)으로 작성한다.
- ▶ ``"a 입력 : " >> OLED >> a << "\n";`` - 사용자로부터 값을 입력받는다.
이때, (입력 전 출력) >> OLED >> (변수) << (줄바꿈 등 입력 후 처리)

● 정수/소수 출력하기

- ▶ %d를 사용하면 정수를 출력할 수 있다. 단, f-문자열의 경우 끝에 :d를 붙일 필요는 없다.
- ▶ "%d를 포함한 문자열" % 값1 % 값2 % ...
- ▶ f"...{값1}...{값2}...{값3}..."
- ▶ `OLED.Printf("%d를 포함한 문자열", 값1, 값2, ...);`

● 정수 포매팅

- ▶ %d : 기본적인 형식 문자
- ▶ %2d : 2칸 차지하는 형식 문자(빈 공간은 띄어쓰기(공백)으로 처리)
- ▶ %02d : 2칸 차지하는 형식 문자(빈 공간은 0으로 처리)

● 소수 포매팅

- ▶ %f : 기본적인 형식 문자(소수점 아래 6자리)
- ▶ %.2f : 소수점 아래 2자리까지만 출력
- ▶ %6.2f : 전체 6자리(소수점 포함), 소수점 아래는 2자리가 되게 출력
- ▶ %06.2f : 전체 6자리(소수점 포함, 빈 공간은 0으로 처리), 소수점 아래는 2자리가 되게 출력.

303 Starlic/Simple/Pythonic 스타일에서 연산 코드

이번 파트에서는 앞에서 다룬 내용을 **Starlic**, **Simple**, **Pythonic Style**로 작성해 보겠다. 생각보다 간단하데, **Starlic**은 중괄호를 없애고 들여쓰기를 한 번 점검하면 되고, **Starlic**에서 세미콜론을 없애면 **Simple** 스타일이, 함수에 `:`를 붙이면 **Pythonic**이 된다.

■ 두 정수의 연산 결과 출력하기

변수 **a**, **b**를 **int**형으로 정의한 후 **OLED** 객체로부터 값을 받아 결과를 출력하는 코드를 **Full-Style**에서 **Pythonic-Style**로 바꾸는 과정이다.

Operation.sl4	Result
<pre>main(){ a = int; b = int; "a 입력 : " >> OLED >> a << "\n"; "b 입력 : " >> OLED >> b << "\n"; OLED << "합 : %d\n" % (a + b); OLED << "차 : %d\n" % (a - b); OLED << "곱 : %d\n" % (a * b); OLED << "몫 : %d\n" % (a // b); OLED << "나머지 : %d\n" % (a %% b); }</pre>	
<pre>main() a = int; b = int; "a 입력 : " >> OLED >> a << "\n"; "b 입력 : " >> OLED >> b << "\n"; OLED << "합 : %d\n" % (a + b); OLED << "차 : %d\n" % (a - b); OLED << "곱 : %d\n" % (a * b); OLED << "몫 : %d\n" % (a // b); OLED << "나머지 : %d\n" % (a %% b);</pre>	<pre>a 입력 : 20 b 입력 : 10 합 : 30 차 : 10 곱 : 200 몫 : 2 나머지 : 0</pre>

Operation.sl4	Result
<pre> main a = int b = int "a 입력 : " >> OLED >> a << "\n" "b 입력 : " >> OLED >> b << "\n" OLED << "합 : %d\n" % (a + b) OLED << "차 : %d\n" % (a - b) OLED << "곱 : %d\n" % (a * b) OLED << "몫 : %d\n" % (a // b) OLED << "나머지 : %d\n" % (a %% b) </pre>	<pre> a 입력 : 20 b 입력 : 10 합 : 30 차 : 10 곱 : 200 몫 : 2 나머지 : 0 </pre>
<pre> main: a = int b = int "a 입력 : " >> OLED >> a << "\n" "b 입력 : " >> OLED >> b << "\n" OLED << "합 : %d\n" % (a + b) OLED << "차 : %d\n" % (a - b) OLED << "곱 : %d\n" % (a * b) OLED << "몫 : %d\n" % (a // b) OLED << "나머지 : %d\n" % (a %% b) </pre>	

방법은 매우 간단하다! 중괄호를 모두 없애버리면 Starlic Style이 되어 유효한 코드가 된다. 그다음 세미콜론과 불필요한 괄호를 모두 없애버리면 Simple Style이 되고, 그 상태에서 함수 정의 뒷부분에 :를 붙이면 Pythonic이 된다. 소스 코드는 취향에 맞게 작성하면 되지만, 스타일의 변환은 쉽게 할 수 있음을 알 수 있다.

■ 두 정수의 연산 결과 출력 - Main 함수를 지우는 경우

`main` 함수를 없애고 작성해도 좋다. 아래와 같이 `main` 함수를 지우고 프로그램해 보자. 주의할 점은 들여쓰기를 없애거나 세미콜론을 붙이던가 둘 중 하나는 지켜야 한다. 들여쓰기하고 세미콜론을 붙이지 않으면 `Full-Style`이 아닌 상태에서 들여쓰기 규칙을 위반한 상태가 되는데, 여기서 세미콜론을 붙이면 `Full-Style`로 인식하여 들여쓰기 규칙이 사라지기 때문에 오류가 나지 않는 특징이 있다.

Operation.sl4	Result
<pre> a = int b = int "a 입력 : " >> OLED >> a << "\n" "b 입력 : " >> OLED >> b << "\n" OLED << "합 : %d\n" % (a + b) OLED << "차 : %d\n" % (a - b) OLED << "곱 : %d\n" % (a * b) OLED << "몫 : %d\n" % (a // b) OLED << "나머지 : %d\n" % (a %% b) </pre>	<pre> a 입력 : 20 b 입력 : 10 합 : 30 차 : 10 곱 : 200 몫 : 2 나머지 : 0 </pre>

■ 두 소수의 연산 결과 출력 - OLED.Printf 사용

이번에는 두 소수의 연산 결과를 출력하는 프로그램을 Full-Style, Starlic, Simple, Pythonic 스타일로 작성해 보겠다.

Operation.sl4	Result
<pre> main(){ a = float; b = float; "a 입력 : " >> OLED >> a << "\n"; "b 입력 : " >> OLED >> b << "\n"; OLED.Printf("합 : %.2f\n", a+b); OLED.Printf("차 : %.2f\n", a-b); OLED.Printf("곱 : %.2f\n", a*b); OLED.Printf("몫 : %.2f\n", a/b); } main() a = float; b = float; "a 입력 : " >> OLED >> a << "\n"; "b 입력 : " >> OLED >> b << "\n"; OLED.Printf("합 : %.2f\n", a+b); OLED.Printf("차 : %.2f\n", a-b); OLED.Printf("곱 : %.2f\n", a*b); OLED.Printf("몫 : %.2f\n", a/b); </pre>	<pre> a 입력 : 1.5 b 입력 : 2.5 합 : 4.00 차 : -1.00 곱 : 3.75 몫 : 0.60 </pre>

Operation.sl4	Result
<pre> main a = float b = float "a 입력 : " >> OLED >> a << "\n" "b 입력 : " >> OLED >> b << "\n" OLED.Printf("합 : %.2f\n", a+b) OLED.Printf("차 : %.2f\n", a-b) OLED.Printf("곱 : %.2f\n", a*b) OLED.Printf("몫 : %.2f\n", a/b) </pre>	
<pre> main: a = float b = float "a 입력 : " >> OLED >> a << "\n" "b 입력 : " >> OLED >> b << "\n" OLED.Printf("합 : %.2f\n", a+b) OLED.Printf("차 : %.2f\n", a-b) OLED.Printf("곱 : %.2f\n", a*b) OLED.Printf("몫 : %.2f\n", a/b) </pre>	<pre> a: 1.50 b: 3.50 합: 5.00 차: -2.00 곱: 5.25 몫: 0.43 </pre>
<pre> a = float b = float "a 입력 : " >> OLED >> a << "\n" "b 입력 : " >> OLED >> b << "\n" OLED.Printf("합 : %.2f\n", a+b) OLED.Printf("차 : %.2f\n", a-b) OLED.Printf("곱 : %.2f\n", a*b) OLED.Printf("몫 : %.2f\n", a/b) </pre>	

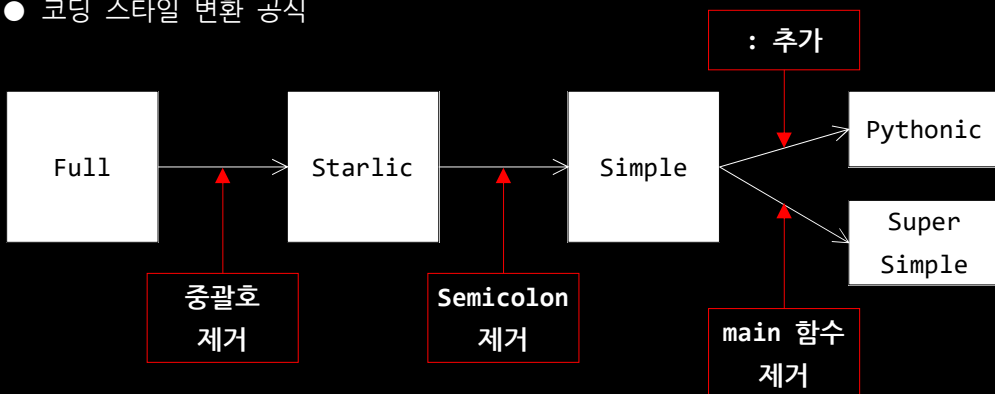
■ 정리

● 코딩 스타일에 따른 프로그래밍 방법

Style	{ }	;	indent	() - No Arg.	:
Full	0	0	선택	선택	X
Starlic	X	0	0	선택	X
Simple	X	선택	0	선택	X
Pythonic	X	선택	0	선택	0
Super-Simple	X	선택	X	선택	선택

- 중괄호 : Full-Style에서는 사용하고, 그 외의 스타일에서는 사용하지 않는다. 만약 다른 스타일에서 중괄호를 사용하면 그 구간에서는 Full-Style이 적용된다.
- 세미콜론 : Full-Style에서는 필수, Starlic에서는 명목상 필수지만 Simple Style과의 혼용이 가능하다. 나머지 스타일에서는 써도 되고 안 써도 된다. main 함수를 사용하지 않는 Super-Simple 스타일에서는 세미콜론을 사용하면 들여쓰기는 인식하지 않지만, 세미콜론을 사용하지 않으면 들여쓰기를 하면 안 된다.
- 들여쓰기 : Full-Style에서는 선택사항이고, 그 외의 스타일에서는 규칙이 적용되는데, Starlic, Simple, Pythonic에서는 들여쓰기가 필수고, Super-Simple에서는 if문과 같은 구문을 사용하지 않는 한 들여쓰기는 하지 말아야 한다.
- 괄호 : 인수가 없는 상황에서의 괄호 사용인데, 모두 선택사항이다. 심지어 Full-Style에서도 main{...}로 작성해도 문제 없다.
- 콜론 : Pythonic 고유의 특성이다. Full 스타일에서 Pythonic 스타일을 부분적으로 사용할 때도 콜론을 사용할 수 있다.(이러면 그 안에서는 Starlic, Simple, Pythonic 중 어떤 것을 써도 컴파일 오류가 안 난다.)

● 코딩 스타일 변환 공식



304 GUE 스타일에서 연산 코드

이번에는 GUE-Style에서의 연산 코드를 구현해 보려고 한다. GUE Style을 제대로 접근하려면 지금까지 했던 것과는 다른 방법으로 접근해 보는 것도 나쁘지 않다. 물론, Simple Style의 코드를 그대로 써도 문제없이 돌아간다.

■ 무늬만 GUE Style

정의는 GUE Style의 방법을 썼지만, 내용은 Simple Style과 다르게 없는 상태를 의미한다. 이렇게 해도 제대로 작동하는 이유는 출력이 있는 함수에서도 유연하게 쓸 수 있게 해야 하기 때문이다. 이렇게 하면 GUE Style만 따로 뺄 이유는 없을 것이다.

Operation.s14	Result
<pre>start>> a = int b = int "a 입력 : " >> OLED >> a << "\n" "b 입력 : " >> OLED >> b << "\n" OLED << "합 : %d\n" % (a + b) OLED << "차 : %d\n" % (a - b) OLED << "곱 : %d\n" % (a * b) OLED << "몫 : %d\n" % (a // b) OLED << "나머지 : %d\n" % (a %% b)</pre>	<pre> a 입력 : 20 b 입력 : 10 합 : 30 차 : 10 곱 : 200 몫 : 2 나머지 : 0</pre>

■ GUE Style에서의 변수 정의법

GUE Style에서 변수를 정의할 때는 대입처럼 정의해도 상관 없지만, 표의 형태로 나타내기 위해 ``_정의법``을 사용한다. `_`정의법은 아래와 같다.

```
_a int
```

이 구문은 변수 `a`를 `int`형으로 정의한다는 의미이다. 절대 변수 이름이 ``_a``인 것은 아니다. 변수 이름은 `a`인데, 정의한다는 의미에서 앞에 `_`를 붙인 것이다. 그래서 변수 이름에는 앞에 `_`가 붙을 수 없다.

■ 값 입력하기

GUE Style에서 사용하기 위한 입력 함수가 마련되어 있다. 바로 `IntVal` 함수와 `FloatVal` 함수이다. 이 함수를 사용하면 자동 출바꿈을 보장하며, 변수를 정의할 때에도 이 함수를 사용해 정의해도 된다.

```
_a IntVal OLED "a 입력 : "
```

■ 값 출력하기

GUE Style에서 값을 출력할 때는 `Print OLED` 또는 `Printf OLED`를 사용한다. `Print OLED`는 `f-문자열`을 사용할 때, `Printf OLED`는 `f-문자열`을 사용하지 않고 형식 문자를 사용할 때 쓰면 된다.

```
Print OLED "합 : {a+b}\n"
```

```
Printf OLED "합 : %d\n" a+b
```

■ GUE Style로 정수의 연산 결과 출력하기

이번에는 **Printf OLED**를 사용해 OLED에 연산 결과를 출력하는 코드를 **GUE Style**을 제대로 사용해서 구현해 보기로 한다.

start>>				
	_a	IntVal	OLED	"a 입력 : "
	_b	IntVal	OLED	"b 입력 : "
	Printf	OLED	"합 : %d\n"	a + b
	Printf	OLED	"차 : %d\n"	a - b
	Printf	OLED	"곱 : %d\n"	a * b
	Printf	OLED	"몫 : %d\n"	a // b
	Printf	OLED	"나머지 : %d\n"	a %% b
end				

Operation.sl4	Result
<pre> start>> _a IntVal OLED "a 입력 : " _b IntVal OLED "b 입력 : " Printf OLED "합 : %d\n" a + b Printf OLED "차 : %d\n" a - b Printf OLED "곱 : %d\n" a * b Printf OLED "몫 : %d\n" a // b Printf OLED "나머지 : %d\n" a %% b </pre>	<pre> a : 20 b : 10 합 : 30 차 : 10 곱 : 200 몫 : 2 나머지 : 0 </pre>

■ GUE Style로 소수의 연산 결과 출력하기

이번에도 **Printf OLED**를 사용해 OLED에 연산 결과를 출력하는 코드를 **GUE Style**을 제대로 사용해서 구현해 보기로 한다.

start>>				
	_a	FloatVal	OLED	"a 입력 : "
	_b	FloatVal	OLED	"b 입력 : "
	Printf	OLED	"합 : %.2f\n"	a + b
	Printf	OLED	"차 : %.2f\n"	a - b
	Printf	OLED	"곱 : %.2f\n"	a * b
	Printf	OLED	"몫 : %.2f\n"	a / b
end				

Operation.sl4	Result
start>>	
_a FloatVal OLED "a 입력 : "	
_b FloatVal OLED "b 입력 : "	
Printf OLED "합 : %.2f\n" a + b	합 : 2.80
Printf OLED "차 : %.2f\n" a - b	차 : 1.50
Printf OLED "곱 : %.2f\n" a * b	곱 : 3.50
Printf OLED "몫 : %.2f\n" a / b	몫 : 1.33

■ 정리

● GUE Style에서 변수의 정의

- `_정의법`을 주로 사용한다.
- 선언만 하는 경우 : `_a int`
- 사용자로부터 값을 입력받는 경우 : `_a IntVal OLED "a 입력 : "`

● GUE Style에서 연산 결과 출력

- 주로 `Printf OLED` 명령어를 사용해서 출력할 수 있음.
- a+b의 값을 출력할 때의 예 : `Printf OLED "합 : %d\n" a+b`

4. OLED에 출력하기

4장에서는 2가지 코드를 다루는데, 1~4장에서는 OLED에 글자 색을 입히고, 글자 크기를 조절하여 출력하는 예제를 5~8장에서는 OLED에 선, 그리고 도형을 출력하는 예제로 총 2가지의 코드를 다양한 스타일로 제작해 보려고 한다.

401 OLED 글자 색과 위치, 그리고 크기

■ OLED 글자 색 입히기

OLED에 글자 색 입히는 공식은 매우 간단하다. 서식 문자를 사용하면 되기 때문이다. 글자 색을 입히는 서식 문자는 ``/r``, 0와 같이 ``/`+`약자``의 형태로 표현한다. 약자는 아래 표와 같다.

서식 문자	색상	서식 문자	색상
<code>`/r`</code>	빨강(Red)	<code>`/o`</code>	주황(Orange)
<code>`/y`</code>	노랑(Yellow)	<code>`/l`</code>	연두(Lime /yellow-green)
<code>`/g`</code>	초록(Green)	<code>`/t`</code>	에메랄드/민트(minT)
<code>`/c`</code>	청록(Cyan)	<code>`/s`</code>	바다색(Sea)
<code>`/b`</code>	파랑(Blue)	<code>`/v`</code>	보라(Violet)
<code>`/m`</code>	자홍(Magenta)	<code>`/p`</code>	장미색(rose/Pink) ⁶⁾
<code>`/w`</code>	흰색(White)	<code>`/k`</code>	검정(black/Key)

● 확장된 글자 색(1) - 어두운색

어두운색은 ``/dr``, ``/dg``와 같이 작성하면 된다. 공식은 ``/`+`d`+(약자)`의 형태로 작성해 주면 된다.

● 확장된 글자 색(2) - 연한 색

연한 색은 ``/R``, ``/B``와 같이 대문자로 적으면 된다.

● 확장된 글자 색(3) - HEX Code

자연스러운 색을 원한다면 HEX Code를 써도 좋다. ``/#FF0000``, ``/#FF8030``과 같이 작성하면 된다.

6) 색상은 장미색이지만, 연하게 출력 시 분홍색이라는 의미에서 p를 사용하기로 했다.

■ OLED 글자 배경색

OLED에 글자 배경 색을 추가할 수도 있다. ``/r``이 빨간색이라면 ``$r``은 배경색을 빨갱게 한다는 의미가 된다.

- 어두운 배경색 : 검정 바탕에서는 자주 쓰인다. ``$dr``과 같이 작성하면 된다.
- 연한 배경색 : 흰 바탕에서는 연한 배경색을 자주 쓴다. ``$dR``과 같이 작성한다.
- HEX Code : 특정 직사각형 영역에서는 이 방식을 쓰는 것이 효과적이다. ``$#0000FF``와 같이 작성하면 된다.

■ OLED 글자 위치

이번에는 OLED에 글자를 출력하는 위치를 정하려고 한다. 글자의 위치는 숫자를 사용해서 적으면 된다. OLED는 96*64 크기에 대해서 고려되었기 때문에 ``/0``부터 ``/6``까지 총 7줄까지 정할 수 있다.



0번줄
1번줄
2번줄
3번줄
4번줄
5번줄
6번줄

■ OLED 글자 크기

OLED의 글자 크기는 작게(``/z``), 보통(``/a``), 크게(``/A``)로 3가지를 사용할 수 있다. 작게는 보통 크기의 2/3만큼 출력되며, 출력 가능한 글자에 제한이 있다. 큰 글자는 보통 크기의 2배만큼 출력된다.



SMALL
Medium
Large

줄 번호와 큰 글자를 같이 사용한다면 주의할 점이 하나 있다. 큰 글자는 2칸을 차지하므로 다음 줄이 2만큼 커져야 글자가 겹치지 않는다. 반대로, 이걸 역이용해서 글자를 겹칠 수도 있다. 즉, 앞줄에 ``/2/A...``를 썼다면 다음 줄은 `/4`로 시작해야 글자가 겹치지 않는다.


■ 정리

- **글자 색** : 글자 자체의 색은 `/` + `약자`로, 배경 색은 `\$` + `약자`로 적는다.
 - 어두운 색 : d를 추가한다.
 - 연한 색 : 대문자로 적는다.
 - 저채도+저명도 색 : d와 대문자를 모두 사용(d는 소문자로 작성)
 - HEX Code 사용 시 : `/` + `#` + (16진수)
- **글자 위치** : `/` + 숫자로 나타내며, OLED 객체 특성상 숫자는 0부터 6까지만 사용한다.
- **글자 크기**
 - 작은 크기 : `/z`를 사용한다. 단, 출력 가능 문자에 제한이 있다.(한글 출력 불가, 영문은 대문자만 출력 가능)
 - 보통 크기(기본값) : `/a`를 사용.
 - 큰 크기 : `/A`를 사용. 이때, 다음 줄을 표시하고 싶다면 큰 글자인 상태로 `\n`을 사용하거나 줄 번호를 2만큼 올려야 한다.

402 Full-Style에서 색깔 입혀서 출력하기

■ Hello는 큰 글자로 빨간색으로, World는 작은 글자로 초록색으로 출력하기

이번 예제는 Hello, World를 출력하는 예제로 다시 돌아오는데, Hello까지는 큰 글자로 빨간색으로, World는 작은 글자로 초록색으로 출력하는 예제이다.

Operation.sl4	Result
<pre>main(){ OLED << "/1/A/rHello,"; OLED << "/3/a/gWorld!"; }</pre>	

다음 예제는 저 구문을 이어붙이는 예제 위주로 들겠다. 첫 번째로 연달아서 출력하는 것은 아래와 같아 << 연산자를 2번 써서 작성해도 좋다. 가독성을 위해 연달아 출력하는 것을 줄을 바꿔도 상관 없다.


Operation.sl4	Result
<pre>main(){ OLED << "/1/A/rHello," << "/3/a/gWorld!"; }</pre>	
<pre>main(){ OLED << "/1/A/rHello," << "/3/a/gWorld!"; }</pre>	

또, 문자열 자체를 하나로 합쳐도 좋다. 이때는 줄을 함부로 바꾸지 않도록 주의하자. 줄바꿈이 필요하다면 /1, /3과 같은 위치를 직접 지정하는 방법이나 \n을 사용하자.

Operation.sl4	Result
<pre>main(){ OLED << "/1/A/rHello,/3/a/gWorld!"; }</pre>	

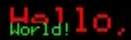

■ << 연산자는 위치 속성을 제외하고 초기화시킨다.

OLED에 출력하는 << 연산자는 글자의 속성을 초기화시키는 특성이 있다. 따라서 << 연산자를 사용하면 글자의 색은 기본 색인 흰색, 크기는 기본 크기로 바뀌게 된다. 아래 예제는 이를 잘 보여주는 코드이다.

Operation.sl4	Result
<pre>main(){ OLED << "/1/A/rHello,\n" << "World!"; }</pre>	
<pre>main(){ OLED << "/1/A/rHello,\n" << "World!"; }</pre>	

■ 큰 글자를 사용하고 줄을 바꾸면 글자가 겹칠 수 있으니 주의하자.

작은 글자 상태로 바꾸고 줄을 바꾸거나 줄 번호를 1만 증가시킨다면 글자가 겹치는 문제가 발생할 수 있으니 주의해야 한다. 단, 큰 글자 상태에서 줄 바꾸고 작은 글자로 바꾸는 것은 상관없다.

Operation.sl4	Result
<pre>main(){ OLED << "/1/A/rHello,/2/a/gWorld!"; }</pre>	
<pre>main(){ OLED << "/1/A/rHello,/a\n/gWorld!"; }</pre>	
<pre>main(){ OLED << "/1/A/rHello,\n/a/gWorld!"; }</pre>	

■ 정리

● 연이어 출력하기

- << 연산자를 여러번 사용하면 연이어서 출력할 수 있다.
- 가독성을 높이기 위해 줄을 바꿔도 상관없다.
- 단, 문자열 내부에서는 줄을 바꾸면 안 된다.

● << 연산자의 특성

- << 연산자는 위치 정보는 초기화하지 않는다.
- << 연산자는 글자 크기와 색깔은 초기화한다.


● 큰 글자 출력 시 주의사항

- 큰 글자를 출력하고 줄 번호를 1만큼 늘리면 글자가 겹치므로 2만큼 늘려야 한다.
- 큰 글자인 상태에서 줄 바꿈 문자 ``\n``을 사용하면 글자가 겹치지 않는다.
- 큰 글자 상태에서 작은 글자로 바꾼 후 줄 바꿈을 사용하면 글자가 겹친다.

403 Starlic, Simple, Pythonic에서 색깔 입혀기

■ Starlic에서 색깔 입히는 예제

앞에서 다룬 예제를 **Starlic**으로 바꾸는 공식을 활용해 색을 입히는 예제를 만들어 보겠다. 아래와 같이 간단하게 만들 수 있다.

Operation.sl4	Result
<pre>main() OLED << "/1/A/rHello," << "/3/a/gWorld!";</pre>	
<pre>main() OLED << "/1/A/rHello,\n" << "/3/a/gWorld!";</pre>	


역시나 한 줄로 출력하는 코드와 두 줄에 나눠서 출력하는 코드 모두 가능하다는 사실을 알 수 있다. 하지만 여기서 주의해야 할 점이 있다. 두 줄에 나눠서 작성하는 경우 들여쓰기 규칙을 적용해 바로 윗줄보다 더 들여쓰기해야 한다. 따라서 아래와 같이 작성하면 컴파일 오류가 날 수 있으니 주의해야 한다.

```
main()
    OLED << "/1/A/rHello,\n"
    << "/3/a/gWorld!";
```

쉽게 말해서 중괄호를 없앴으니 들여쓰기 규칙을 좀 더 준수해 달라는 것이다. 즉, 한 줄에 다 적지 못할 정도로 코드가 길다면 줄을 넘겨도 좋지만 들여쓰기를 꼭 지켜줘야 한다.

■ Simple Style에서 색깔 입히는 예제


앞에서 다른 예제를 Simple 스타일로 바꾸는 공식을 활용해 색을 입히는 예제를 만들어 보겠다. 앞에서 다른 코드에서 불필요한 괄호와 세미콜론을 지우면 끝이다.

Operation.sl4	Result
<pre>main OLED << "/1/A/rHello,\n" << "/3/a/gWorld!"</pre>	
<pre>main OLED << "/1/A/rHello,\n" << "/3/a/gWorld!"</pre>	

역시 실행 결과는 앞서와 같은데, 역시 들여쓰기를 꼭 준수해야 한다는 점은 변하지 않는다. 한 줄에 다 쓰지 못한다면 줄을 바꾸되, 들여쓰기를 준수해야 오류가 나지 않는다.


■ Pythonic에서 색깔 입히는 예제

앞에서 다른 예제에서 main:라고 적으면 끝나기 때문에 상당히 쉽게 작성할 수 있다. 역시 들여쓰기 규칙은 똑같이 적용된다.

Operation.sl4	Result
<pre>main: OLED << "/1/A/rHello,\n" << "/3/a/gWorld!"</pre>	
<pre>main: OLED << "/1/A/rHello,\n" << "/3/a/gWorld!"</pre>	

■ Super Simple에서 색깔 입히는 예제

앞에서 다른 예제에서 main 함수 없이 작성할 수 있다. 하지만, 코드가 매우 길어 지더라도 줄 바꿈이 허용되지 않는 특징이 있다. 여기서만큼은 예외인데, 함수 정의와 혼동될 여지가 있기 때문이다. 그러므로 함수 밖에는 간단한 초기화 코드를 제외하고는 작성하지 않을 것을 추천한다.

Operation.sl4	Result
<pre>OLED << "/1/A/rHello,\n" << "/3/a/gWorld!"</pre>	

■ 정리


Starlic -> Simple -> Pythonic으로 바꾸는 공식을 적용해서 OLED에 출력하는 내용에 글자색, 위치, 크기를 적용하고, 이어서 출력해 보았다.

● Starlic, Simple, Pythonic Style

- Starlic : Full-Style에서 중괄호를 제외하고 들여쓰기 규칙을 맞추면 된다.
- Simple : Starlic-Style에서 세미콜론을 제거한다.
- Pythonic : main을 main:로 변경한다.
- 주의사항 : 한 줄에 긴 코드를 작성해 줄바꿈을 원한다면 추가적인 들여쓰기를 해야 컴파일 오류가 나지 않는다.

● Super Simple Style

- main 함수 없이도 글자 크기, 색 조절하는 데 문제없이 돌아간다.
- 다만, 코드가 길어 2줄 이상에 걸쳐서 출력하는 것은 허용되지 않는다.

Operation.sl4	Result
<pre>main OLED << "/1/A/rHello,\n" << "/3/a/gWorld!"</pre>	
<pre>main OLED << "/1/A/rHello,\n" << "/3/a/gWorld!"</pre>	

404 GUE-Style에서 색깔 입히기

■ GUE-Style에서 색깔 입히기

GUE-Style에서는 문자열 객체를 정의하고 대입하는 과정을 활용하여 색깔을 입히는 예제를 설명하려고 한다. 그 이유는 OLED << 연산자보다는 Print OLED 명령구문을 주로 사용하고, 표로 나타내기 적합한 구조를 쓰기 위해서다.

물론, Simple Style에서 작성한 내용을 그대로 작성한다고 해서 컴파일 오류가 나는 것은 아니다. 다음과 같이 작성해도 정상 작동하긴 한다.

Operation.sl4	Result
start>> OLED << "/1/A/rHello,\n" << "/3/a/gWorld!"	Hello, World!
start>> OLED << "/1/A/rHello,\n"	
<< "/3/a/gWorld!"	


하지만, GUE에서는 문자열을 바로 출력하기보다는 문자열 객체에 저장해서 자유자재로 변경한 후 한꺼번에 출력하는 쪽을 지향하므로 아래와 같은 형태로 작성할 것이다.

start>>			
	_st	str	"/1/A/rHello,\n"
	_st	st +	"/3/a/gWorld!"
	Print	OLED	st
end			


Operation.sl4	Result
start>> _st str "/1/A/rHello,\n"	Hello, World!
_st st + "/3/a/gWorld!"	
Print OLED st	

■ 문자열 덧셈의 특성

문자열의 덧셈 연산은 두 문자열을 이어 붙인다는 의미로 해석된다. 따라서 **OLED** << 연산의 글자 색과 크기 초기화되는 특성을 극복할 수 있다. 아래는 **World**도 큰 글자로 출력하는 예제다.

Operation.sl4	Result
<pre>start>> _st str "/1/A/rHello,\n" _st st + "/3/gWorld!" Print OLED st</pre>	

아래는 글자 색을 유지하고 크기는 작게 출력하여 빨간색 **Hello, World!**를 출력하는 예제이다.

Operation.sl4	Result
<pre>start>> _st str "/1/A/rHello,\n" _st st + "/3/aWorld!" Print OLED st</pre>	

■ GUE 암묵의 룰

GUE Style로 작성했을 때는 **Simple Style**처럼 일반적인 함수를 사용해도 좋지만, 괄호를 최소화하는 방향을 지향한다. 그래서 출력이 있는 함수의 출력은 무조건 변수에 대입한 후 그 변수를 직접 사용해 주는 것이 좋다. 따라서 **str("/1/A/rHello,\n")**를 사용한다면 그걸 어느 변수에 저장한 뒤 다른 곳에서 사용하는 것이 좋다. 물론 필수사항은 아니다.

■ 정리

- GUE Style에서 문자열 정의 기법

```
_st str "/1/A/rHello,\n"
```

- GUE Style에서 문자열 누적 기법

```
_st st + "/3/aWorld!"
```

- GUE Style에서 문자열 2개 이상 이어붙여 출력하기

- 문자열을 누적해서 한꺼번에 출력하는 기법을 주로 사용한다.
- 문자열 누적 기법을 사용하면 글자색/크기가 초기화되는 현상을 막을 수 있다.

405 OLED 도형 출력

Starlit에서 사용하는 OLED는 Graphic 방식이라서 글자뿐 아니라 그림도 출력할 수 있다. 5장부터 8장까지는 OLED 직선 출력 함수와 OLED 사각형 출력 함수를 다룬다. 또, math 함수를 사용해서 정다각형을 어떻게 출력할지 알아보겠다.

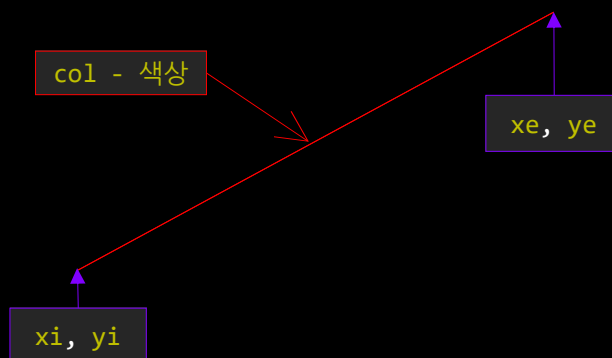
■ OLED 직선 출력

OLED에 직선을 출력하는 함수는 OLED.Line을 사용한다. 시점과 종점을 넣으면 선을 그려주는 함수이다.

● 실수 사용 시

```
OLED.Line(xi, yi, xe, ye, col)
```

- xi : 시작점 x값(0~95의 정수)
- yi : 시작점 y값(0~63의 정수)
- xe : 끝점 x값(0~95의 정수)
- ye : 끝점 y값(0~63의 정수)
- col : 색상(HEX Code 기반 16진수(0x000000~0xFFFFFF) 정수)



● 복소수 사용 시

```
OLED.Line(startPos, endPos, col)
```

- startPos : 시작점 ((0~95)+j*(0~63)의 복소수)
- endPos : 끝점 ((0~95)+j*(0~63)의 복소수)
- col : 색상(HEX Code 기반 16진수(0x000000~0xFFFFFF) 정수)

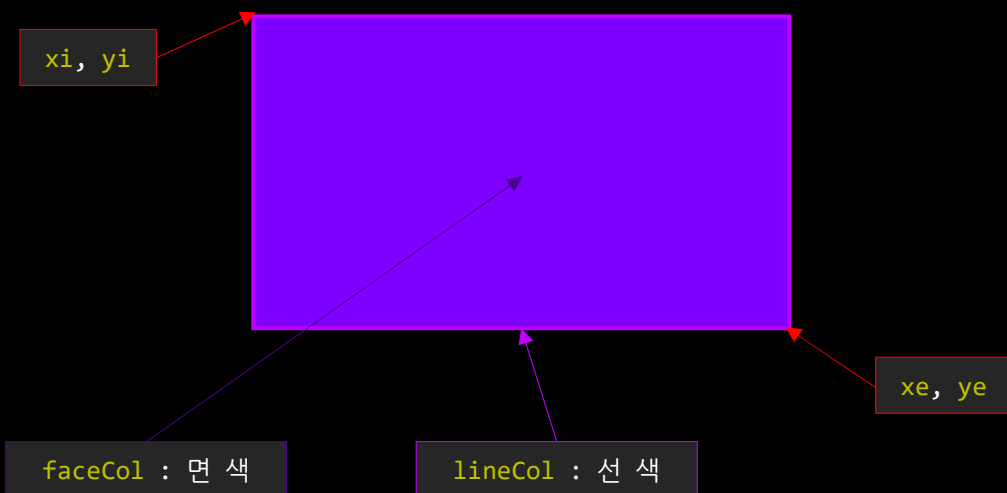
■ OLED 사각형 함수

OLED에 사각형을 출력하는 함수는 `OLED.Rectangle` 함수를 사용하면 된다. 역시 시점과 종점을 넣으면 직사각형을 생성해 주는 함수이다.

● 실수 사용 시

`OLED.Rectangle(xi, yi, xe, ye, lineCol, faceCol)`

- `xi` : 시작점 x값($0\sim95$ 의 정수)
- `yi` : 시작점 y값($0\sim63$ 의 정수)
- `xe` : 끝점 x값($0\sim95$ 의 정수)
- `ye` : 끝점 y값($0\sim63$ 의 정수)
- `lineCol` : 테두리 색상(HEX Code 기반 16진수($0x000000\sim0xFFFFFFFF$) 정수)
- `faceCol` : 면 색상(HEX Code 기반 16진수($0x000000\sim0xFFFFFFFF$) 정수)



● 복소수 사용 시

`OLED.Rectangle(startPos, endPos, lineCol, faceCol)`

- `startPos` : 시작점 ($(0\sim95)+j*(0\sim63)$ 의 복소수)
- `endPos` : 끝점 ($(0\sim95)+j*(0\sim63)$ 의 복소수)
- `lineCol` : 테두리 색상(HEX Code 기반 16진수($0x000000\sim0xFFFFFFFF$) 정수)
- `faceCol` : 면 색상(HEX Code 기반 16진수($0x000000\sim0xFFFFFFFF$) 정수)

■ Math 함수 사용하기

Starlit에서는 **Math** 함수를 사용할 수 있다. 수학에서 자주 사용하는 함수인 **sin**, **cos**, **tan**, **log**, **ln** 등을 지원한다. 각 함수의 기능을 살펴보면 다음과 같다. 우선, 함수를 사용하기 위한 주요 상수를 살펴보면 아래와 같다.

Const	Value	Type	Description
PI	3.141593	float	원주율
EXP	2.178282	float	자연로그의 밑
j	$\sqrt{-1}$	complex	허수 단위 for문의 idx와 혼동을 피하기 위해 j 사용.

그 다음 함수를 살펴보면 아래와 같다.

Function	Type	Description
exp(x:float)	float	밑을 e 로 하는 지수함수 $e^0 = 1$, $e^1 = e$
log(x:float)	float	밑을 e 로 하는 로그함수 $\log 1 = 0$, $\log e = 1$
log10(x:float)	float	밑을 10으로 하는 로그함수(상용로그) $\log_{10} 1 = 0$, $\log_{10} 10 = 1$
sin(x:float)	float	원점에서 단위원 위의 각도 x rad만큼 그렸을 때 y 좌표 $\sin 0 = 0$, $\sin \frac{\pi}{4} = 0.707$, $\sin \frac{\pi}{2} = 1$
cos(x:float)	float	원점에서 단위원 위의 각도 x rad만큼 그렸을 때 x 좌표 $\cos 0 = 1$, $\cos \frac{\pi}{4} = 0.707$, $\cos \frac{\pi}{2} = 0$
tan(x:float)	float	원점에서 단위원 위의 각도 x rad만큼 그린 선분 또는 직선의 기울기 $\tan x = \frac{\sin x}{\cos x}$
asin(x:float)	float	sin 함수의 역함수
acos(x:float)	float	cos 함수의 역함수
atan(x:float)	float	tan 함수의 역함수
rad(x:float)	float	60분법을 호도법으로 변환
deg(x:float)	float	호도법을 60분법으로 변환

■ 알아두면 유용한 Math 함수들

도형을 다룰 때 알아두면 좋은 Math 함수들이 있다. 이런 함수들은 잘 기억해 두고 써먹으면 유용하다.

Function	Type	Description
<code>pow(x:float, a:float)</code>	<code>float</code>	x^a , 지수 출력
<code>atan2(x:float, y:float)</code>	<code>float</code>	(x,y) 좌표를 입력받아서 각도를 radian 단위로 출력.
<code>abs(x:int)</code> <code>abs(x:float)</code> <code>abs(x:complex_t)</code>	<code>int</code> <code>float</code>	$ x $ 절댓값을 출력 복소수의 절댓값도 float로 출력됨.
<code>ceil(x:float)</code>	<code>float</code>	올림한 값을 출력
<code>floor(x:float)</code>	<code>float</code>	버림한 값을 출력
<code>round(x:float)</code>	<code>float</code>	반올림한 값을 출력
<code>lround(x:float)</code>	<code>float</code>	반올림한 값을 정수로 출력 1가 아닌 1이 붙었는데, C언어에서 long이라는 의미로 추정.

■ 복소수 체계에서도 대부분의 함수를 사용할 수 있다!

역삼각함수와 올림/버림/반올림 함수를 제외한 앞에서 언급한 함수들은 복소수에서도 유효하다. 단, `atan2` 함수는 실수 체계에서 사용하라고 만든 함수이고, 복소수 체계에서 여기에 대응되는 함수는 `arg` 함수로 따로 정의되어 있다. 이처럼 복소수에서 사용할 수 있는 함수들은 다음과 같다.

Function	Type	Description
<code>Re(x:complex_t)</code>	<code>float</code>	실수부 출력
<code>Im(x:complex_t)</code>	<code>float</code>	허수부 출력
<code>conj(x:complex_t)</code>	<code>complex_t</code>	켈레복소수(Complex Conjugate) 출력
<code>abs(x:complex_t)</code>	<code>float</code>	복소수의 절댓값 출력 복소평면에서 나타냈을 때 원점과의 거리. $ a + jb = \sqrt{a^2 + b^2}$
<code>arg(x:complex_t)</code>	<code>float</code>	복소수의 편각 출력(radian 단위) 복소평면에서 나타냈을 때 x축과의 각도.

■ 정리

● OLED에 선 출력하기

- `OLED.Line(xi, yi, xe, ye, col)`
- `OLED.Line(startPos, endPos, col)`
- 시점 좌표, 종점 좌표를 넣어 선 출력.

● OLED에 면(사각형) 출력하기

- `OLED.Rectangle(xi, yi, xe, ye, lineCol, faceCol)`
- `OLED.Rectangle(startPos, endPos, lineCol, faceCol)`
- 시점 좌표, 종점 좌표를 넣어 면 출력.
- lineCol : 테두리 색상, faceCol : 면 색상

● 수학 함수

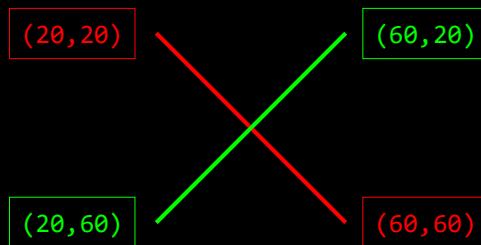
- 수학함수에는 `exp`, `log`, `sqrt`, `sin`, `cos`, `tan` 등이 있다.
- 실수, 복소수에서 지원된다.
- 복소수에서는 `abs`함수와 `arg`함수를 유용하게 사용할 수 있다.

406 Full-Style에서 도형 출력하기

6~8장에서는 도형을 출력하는 예를 몇 가지 들어보려고 한다. 6장에서 다룬 예제를 7, 8장에서 스타일을 바꿔 가면서 다룬 예정이다. 도형 출력은 x 모양, 사각형 출력, 정다각형 출력과 별 모양 출력으로 4가지 예제를 들어볼 예정이며, 그 과정에서 수학적인 요소가 추가될 수 있으니 수학이 어렵다면 가볍게 보고 넘어가도 좋다.

■ x 모양 출력하기

처음에는 가장 간단한 x 모양을 출력하는 예를 들어보겠다. 아래 그림과 같이 (20,20), (60,60)을 잇고, (20,60), (60,20)을 이어서 x 모양을 출력해 보겠다. 첫 번째 직선은 빨간색(FF0000)으로, 두 번째 직선은 초록색(00FF00)으로 출력한다.



● 실수(정수) 사용하기

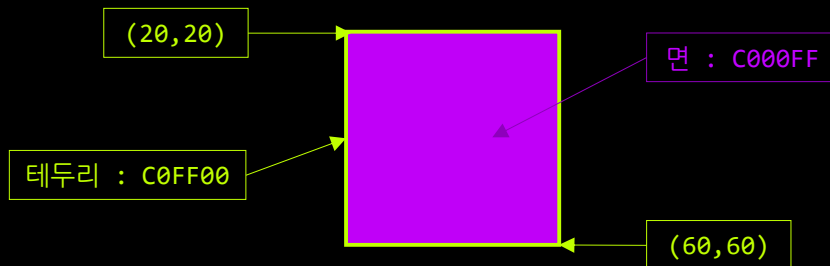
Geometry.sl4	Result
<pre>main(){ OLED.Line(20, 20, 60, 60, 0xFF0000); OLED.Line(20, 60, 60, 20, 0x00FF00); }</pre>	

● 복소수 사용하기

Geometry.sl4	Result
<pre>main(){ OLED.Line(20+j*20, 60+j*60, 0xFF0000); OLED.Line(20+j*60, 60+j*20, 0x00FF00); }</pre>	

■ 사각형 출력하기

다음으로는 OLED에 사각형을 출력하는 코드를 구현해 보려고 한다. 시작점과 끝점은 각각 (20,20), (60,60)으로 설정하고, 테두리는 연두색(C0FF00), 속은 보라색(C000FF)으로 설정해 보자.



● 실수(정수) 사용하기

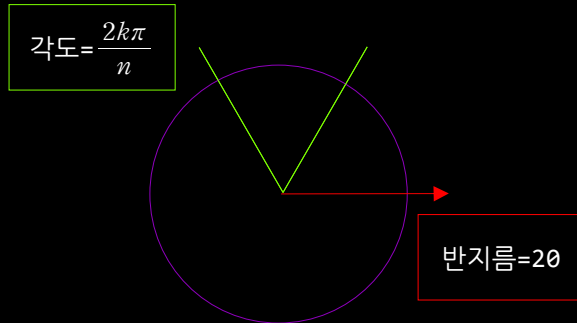
Geometry.s14	Result
<pre>main(){ OLED.Rectangle(20, 20, 60, 60, 0xC000FF, 0xC0FF00); }</pre>	

● 복소수 사용하기

Geometry.s14	Result
<pre>main(){ OLED.Rectangle(20+j*20, 60+j*60, 0xC000FF, 0xC0FF00); }</pre>	

■ 정다각형 출력하기

사용자로부터 값을 입력받아서 정다각형을 출력하는 예제를 살펴보자. 꼭짓점의 위치를 잘 파악하면 정다각형을 출력하는 것은 어렵지 않다. 그러면 꼭짓점을 어떻게 찾으면 될까? 바로 원을 그려서 생각하면 된다.



자, 반지름이 20인 원이 있다고 생각해 보면, 정다각형의 k 번째 꼭짓점이 n 개 있다고 볼 수 있으며, 각도는 $\frac{2k\pi}{n}$ rad가 된다. 따라서 각각의 좌표는 $[\cos(\frac{2k\pi}{n}), \sin(\frac{2k\pi}{n})]$ 라고 놓을 수 있다.

한편, 원의 중심을 $(0,0)$ 으로 하면 원이 잘릴 것이니 OLED의 중심인 $(48,32)$ 로 해야 한다. 따라서 실제 좌표는 $[48 + \cos(\frac{2k\pi}{n}), 32 + \sin(\frac{2k\pi}{n})]$ 로 정해준다.

● 반복 연산자를 사용해서 출력하기


$k:1:n$ 이라고 작성하면 반복변수 k 를 생성해서 1부터 n 까지 반복하여 구문을 실행하라는 의미이다. 또, k 의 반복 속성을 한 번 정의하면 그 k 는 계속 사용할 수 있다. 따라서 아래와 같이 정다각형을 출력하는 코드를 구현할 수 있다.

Geometry.s14	Result
<pre> main(){ n = int; "a 입력 : " >> OLED >> n << "\n"; OLED.Line(48 + 20 * cos(2*(k:1:n-1)*PI / n), 32 + 20 * sin(2*(k-1)*PI / n), 48 + 20 * cos(2*k*PI / n), 32 + 20 * sin(2*k*PI / n), 0xFF0000); } </pre>	<p>a 입력 : 5</p>

● 복소수를 사용하는 경우


복소수를 사용한다면 코드가 매우 간단해질 수 있다. 그 이유는 $e^{jx} = \cos x + j \sin x$ 로 알려진 세상에서 가장 아름다운 등식 때문이다. 따라서 아래와 같이 간단하게 작성해서 정다각형을 출력해 볼 수 있다.

$[48 + \cos(\frac{2k\pi}{n}), 32 + \sin(\frac{2k\pi}{n})]$ 좌표를 복소수로 표현하면 $[48 + \cos(\frac{2k\pi}{n}) + j(32 + \sin(\frac{2k\pi}{n}))]$ 가 되어 교환법칙과 분배법칙을 적용하면 $[48 + j32 + \cos(\frac{2k\pi}{n}) + j\sin(\frac{2k\pi}{n})]$ 이 되는데, 오일러의 등식을 적용해서 표현하면 $[48 + j32 + \exp(j\frac{2k\pi}{n})]$ 로 간단하게 쓸 수 있다. 따라서 아래와 같이 작성해도 결과는 똑같이 출력된다.

Geometry.s14	Result
<pre>main(){ n = int; "a 입력 : " >> OLED >> n << "\n"; OLED.Line(48+j*32 + 20 * exp(j*2*(k:1:n-1)*PI / n), 48+j*32 + 20 * exp(j*2*k*PI / n), 0xFF0000); }</pre>	<p>a 입력 : 5</p> 

■ 별 모양 출력하기

n을 5로 놓고, k 자리에 2를 곱해버리면, 즉, 앞의 코드에서 $j*2*...$ 를 $j*4*...$ 로 바꾸면 별 모양을 출력할 수 있다. 원리는 꼭짓점을 2칸씩 띄어서 이어버리는 방법이다. 직접 그려보면 이해할 수 있다.

Geometry.s14	Result
<pre>main(){ n = 5; OLED.Line(48+j*32 + 20 * exp(j*4*(k:1:n-1)*PI / n), 48+j*32 + 20 * exp(j*4*k*PI / n), 0xFF0000); }</pre>	

■ 정리

● OLED에 직선 출력하기

- OLED.Line을 사용해서 직선을 출력할 수 있다.

● OLED에 사각형 출력하기

- OLED.Rectangle을 사용해서 사각형을 출력할 수 있다.

● OLED에 정다각형 출력하기

- 반복 연산자 : k:시작:끝 형태로 사용하며, 한 번 정의하면 그 구문 안에서 인덱스 변수(k)를 사용할 수 있다.
- 반복 연산자와 좌표 구하는 식을 이용해 정다각형을 출력할 수 있다.
- 출력 공식(실수) : cos와 sin 함수로 좌표를 구한다.

```
OLED.Line(48 + 20 * cos(2*(k:1:n-1)*PI / n),  
          32 + 20 * sin(2*(k-1)*PI / n),  
          48 + 20 * cos(2*k*PI / n),  
          32 + 20 * sin(2*k*PI / n), 0xFF0000);
```

- 출력 공식(복소수) : exp 함수에 순허수를 넣어 정다각형을 출력할 수 있다.

```
OLED.Line(48+j*32 + 20 * exp(j*2*(k:1:n-1)*PI / n),  
          48+j*32 + 20 * exp(j*2*k*PI / n), 0xFF0000);
```

● OLED에 별 출력하기

- 정다각형 출력 알고리즘에서 꼭짓점을 넘어가는 단위를 변조하면 별을 출력할 수 있다. 이를테면 2칸씩 띄워서 출력하고, n을 5로 하면 오각별꼴이 그려진다.
- 주의할 점 : 꼭짓점을 뛰어넘는 개수와 n은 서로소여야 하며, 그 개수는 1또는 n-1이 될 수 없다. 따라서 이 방식으로는 육각별꼴을 그리는 것은 불가능하다.

407 Starlic, Simple, Pythonic에서 도형 출력하기


7장에서는 Full-Style에서 다룬 예제를 Starlic, Simple, Pythonic에서 다뤄 보는 시간이다.

■ X 모양 출력하기

앞에서 다룬 X 모양 출력을 Starlic, Simple, Pythonic에서 작성하는 과정을 정리하면 공식을 그대로 적용하면 된다.


● 실수에서 출력하는 경우

아래 코드는 순서대로 Full-Style -> Starlic-Style -> Simple-Style -> Pythonic-Style로 작성한 것이다.

Geometry.s14	Result
<pre>main(){ OLED.Line(20, 20, 60, 60, 0xFF0000); OLED.Line(20, 60, 60, 20, 0x00FF00); }</pre>	
<pre>main() OLED.Line(20, 20, 60, 60, 0xFF0000); OLED.Line(20, 60, 60, 20, 0x00FF00);</pre>	
<pre>main OLED.Line(20, 20, 60, 60, 0xFF0000) OLED.Line(20, 60, 60, 20, 0x00FF00)</pre>	
<pre>main: OLED.Line(20, 20, 60, 60, 0xFF0000) OLED.Line(20, 60, 60, 20, 0x00FF00)</pre>	


● 복소수에서 출력하는 경우

복소수에서 역시 실수 출력할 때와 똑같은 공식을 적용하면 된다. 역시 순서대로 Full-Style -> Starlic-Style -> Simple-Style -> Pythonic-Style로 작성했다.

Geometry.s14	Result
<pre>main(){ OLED.Line(20+j*20, 60+j*60, 0xFF0000); OLED.Line(20+j*60, 60+j*20, 0x00FF00); }</pre>	
<pre>main() OLED.Line(20+j*20, 60+j*60, 0xFF0000); OLED.Line(20+j*60, 60+j*20, 0x00FF00);</pre>	
<pre>main OLED.Line(20+j*20, 60+j*60, 0xFF0000) OLED.Line(20+j*60, 60+j*20, 0x00FF00)</pre>	
<pre>main: OLED.Line(20+j*20, 60+j*60, 0xFF0000) OLED.Line(20+j*60, 60+j*20, 0x00FF00)</pre>	

● Super-Simple Style


이 코드는 Super-Simple Style이 적용될 수 있다. Simple Style에서 main을 지우고 실행해 보자. 다만, 들여쓰기 규칙은 한 번 검토해야 한다.

Geometry.s14	Result
<pre>OLED.Line(20, 20, 60, 60, 0xFF0000) OLED.Line(20, 60, 60, 20, 0x00FF00) OLED.Line(20+j*20, 60+j*60, 0xFF0000) OLED.Line(20+j*60, 60+j*20, 0x00FF00)</pre>	


■ 사각형 출력하기

앞에서 다른 사각형 출력 프로그램을 **Starlic**, **Simple**, **Pythonic**에서 작성해 보면 다음과 같다.

● 실수를 사용하는 경우


Geometry.sl4	Result
main(){ OLED.Rectangle(20, 20, 60, 60, 0xC000FF, 0xC0FF00); }	
main() OLED.Rectangle(20, 20, 60, 60, 0xC000FF, 0xC0FF00);	
main OLED.Rectangle(20, 20, 60, 60, 0xC000FF, 0xC0FF00)	
main: OLED.Rectangle(20, 20, 60, 60, 0xC000FF, 0xC0FF00)	

● 복소수를 사용하는 경우

Geometry.sl4	Result
main(){ OLED.Rectangle(20+j*20, 60+j*60, 0xC000FF, 0xC0FF00); }	
main() OLED.Rectangle(20+j*20, 60+j*60, 0xC000FF, 0xC0FF00);	
main OLED.Rectangle(20+j*20, 60+j*60, 0xC000FF, 0xC0FF00)	
main: OLED.Rectangle(20+j*20, 60+j*60, 0xC000FF, 0xC0FF00)	

● Super-Simple Style에서 사용

이 프로그램도 Super-Simple Style에서 사용할 수 있다. 한 줄로 간편하게 코드를 작성해도 좋다.





Geometry.sl4	Result
OLED.Rectangle(20, 20, 60, 60, 0xC000FF, 0xC0FF00)	
OLED.Rectangle(20+j*20, 60+j*60, 0xC000FF, 0xC0FF00)	

■ 정다각형 출력하기

앞에서 다룬 정다각형 출력 프로그램을 **Starlic**, **Simple**, **Pythonic**에서 작성해 보겠다.

● 실수를 사용하는 경우

실수를 사용하는 경우 **cos** 함수와 **sin** 함수를 사용해서 정다각형을 구현할 수 있다. 역시 **Full Style**부터 **Pythonic Style**로의 여정은 아래와 같다. 여러 줄에 나누어 작성한 코드라고 해서 반점을 지우면 절대 안된다는 점도 유의하자.

Geometry.sl4	Result
<pre>main(){ n = int; "a 입력 : " >> OLED >> n << "\n"; OLED.Line(48 + 20 * cos(2*(k:1:n-1)*PI / n), 32 + 20 * sin(2*(k-1)*PI / n), 48 + 20 * cos(2*k*PI / n), 32 + 20 * sin(2*k*PI / n), 0xFF0000); }</pre>	<pre>a 입력 : 5</pre> 
<pre>main() n = int; "a 입력 : " >> OLED >> n << "\n"; OLED.Line(48 + 20 * cos(2*(k:1:n-1)*PI / n), 32 + 20 * sin(2*(k-1)*PI / n), 48 + 20 * cos(2*k*PI / n), 32 + 20 * sin(2*k*PI / n), 0xFF0000);</pre>	<pre>a 입력 : 6</pre> 
<pre>main n = int; "a 입력 : " >> OLED >> n << "\n" OLED.Line(48 + 20 * cos(2*(k:1:n-1)*PI / n), 32 + 20 * sin(2*(k-1)*PI / n), 48 + 20 * cos(2*k*PI / n), 32 + 20 * sin(2*k*PI / n), 0xFF0000)</pre>	<pre>a 입력 : 7</pre> 
<pre>main: n = int; "a 입력 : " >> OLED >> n << "\n" OLED.Line 48 + 20 * cos(2*(k:1:n-1)*PI / n), 32 + 20 * sin(2*(k-1)*PI / n), 48 + 20 * cos(2*k*PI / n), 32 + 20 * sin(2*k*PI / n), 0xFF0000</pre>	<pre>a 입력 : 8</pre> 

Pythonic에서 약간 눈에 띄는 변화가 있었는데, 여러 줄에 걸쳐서 출력한다면 괄호를 과감하게 쓰지 않아야 마지막에 괄호를 안 적는 실수를 줄일 수 있어서 그렇게 작성해 보았다.


● 복소수를 사용하는 경우


복소수를 사용한다면 `exp` 함수를 사용해서 손쉽게 그릴 수 있다. 따라서 코드의 양을 대폭 줄일 수 있다. 역시 **Full Style**부터 **Pythonic Style**로의 여정은 아래와 같다. 여러 줄에 나뉘어 작성한 코드라고 해서 반점을 지우면 절대 안된다는 점도 유의하자.

Geometry.s14	Result
<pre>main(){ n = int; "a 입력 : " >> OLED >> n << "\n"; OLED.Line(48+j*32 + 20 * exp(j*2*(k:1:n-1)*PI / n), 48+j*32 + 20 * exp(j*2*k*PI / n), 0xFF0000); }</pre>	<p>a 입력 : 5</p> 
<pre>main() n = int; "a 입력 : " >> OLED >> n << "\n"; OLED.Line(48+j*32 + 20 * exp(j*2*(k:1:n-1)*PI / n), 48+j*32 + 20 * exp(j*2*k*PI / n), 0xFF0000);</pre>	<p>a 입력 : 6</p> 
<pre>main n = int "a 입력 : " >> OLED >> n << "\n"; OLED.Line(48+j*32 + 20 * exp(j*2*(k:1:n-1)*PI / n), 48+j*32 + 20 * exp(j*2*k*PI / n), 0xFF0000)</pre>	<p>a 입력 : 7</p> 
<pre>main: n = int "a 입력 : " >> OLED >> n << "\n"; OLED.Line 48+j*32 + 20 * exp(j*2*(k:1:n-1)*PI / n), 48+j*32 + 20 * exp(j*2*k*PI / n), 0xFF0000</pre>	<p>a 입력 : 8</p> 

■ 별모양 출력하기

앞에서는 5각별꼴만 다뤘지만, 이번에는 7각별꼴을 출력하는 방법을 다뤄 본다. 물론, 스타일을 다르게 해서 7각별꼴을 출력하는 것이다. 역시 **Full Style**부터 **Pythonic Style**로의 여정은 아래와 같다.


Geometry.s14	Result
<pre>main(){ n = 7; OLED.Line(48+j*32 + 20 * exp(j*4*(k:1:n-1)*PI / n), 48+j*32 + 20 * exp(j*4*k*PI / n), 0xFF0000); }</pre>	
<pre>main() n = 7; OLED.Line(48+j*32 + 20 * exp(j*4*(k:1:n-1)*PI / n), 48+j*32 + 20 * exp(j*4*k*PI / n), 0xFF0000);</pre>	
<pre>main n = 7 OLED.Line(48+j*32 + 20 * exp(j*4*(k:1:n-1)*PI / n), 48+j*32 + 20 * exp(j*4*k*PI / n), 0xFF0000)</pre>	
<pre>main n = 7 OLED.Line 48+j*32 + 20 * exp(j*4*(k:1:n-1)*PI / n), 48+j*32 + 20 * exp(j*4*k*PI / n), 0xFF0000</pre>	

Geometry.s14	Result
<pre>main(){ n = 7; OLED.Line(48+j*32 + 20 * exp(j*6*(k:1:n-1)*PI / n), 48+j*32 + 20 * exp(j*6*k*PI / n), 0x00FF00); }</pre>	
<pre>main() n = 7; OLED.Line(48+j*32 + 20 * exp(j*6*(k:1:n-1)*PI / n), 48+j*32 + 20 * exp(j*6*k*PI / n), 0x00FF00);</pre>	
<pre>main n = 7 OLED.Line(48+j*32 + 20 * exp(j*6*(k:1:n-1)*PI / n), 48+j*32 + 20 * exp(j*6*k*PI / n), 0x00FF00)</pre>	
<pre>main n = 7 OLED.Line 48+j*32 + 20 * exp(j*6*(k:1:n-1)*PI / n), 48+j*32 + 20 * exp(j*6*k*PI / n), 0x00FF00</pre>	


이번에는 2가지 모양의 별을 그렸는데, 하나는 꼭짓점을 2칸 건너뛴 것이고, 다른 하나는 꼭짓점을 3칸 건너뛰어서 얻었다.

■ Super-Simple-Style에서 별을 출력할 수 있을까?

앞에서 Super-Simple-Style에서는 여러 줄에 걸쳐서 출력하지 않는다고 하였다. 따라서 한 구문이 2줄에 걸쳐 있을 때는 Super-Simple-Style을 적용할 수 없다. 하지만 놀랍게도 아래 코드는 유효한 코드이다. 왜 유효할까?

Geometry.s14	Result
<pre>n = 7 OLED.Line(48+j*32 + 20 * exp(j*6*(k:1:n-1)*PI / n), 48+j*32 + 20 * exp(j*6*k*PI / n), 0x00FF00)</pre>	

유효한 이유는 괄호에 있다. 괄호가 닫히지 않으면 구문이 끝나지 않았다고 인식하기 때문에 계속 읽기 때문이다. 다만, 아래와 같이 작성하면 컴파일되지 않으니 주의해야 한다.

Geometry.s14	Result
<pre>n = 7 OLED.Line 48+j*32 + 20 * exp(j*6*(k:1:n-1)*PI / n), 48+j*32 + 20 * exp(j*6*k*PI / n), 0x00FF00</pre>	

■ 정리

● 스타일의 변환 과정

- Style의 변환은 Full-Style에서 Pythonic-Style로의 공식을 그대로 적용하면 된다.
- Pythonic Style에서 인수를 여러 줄에 걸쳐 출력할 때 실수로 괄호를 닫지 않을 것에 대비해 끝에 있는 괄호를 생략해도 된다(Starlic, Simple도 유효).
- Super-Simple Style에서는 함수를 사용하지 않고 쓰기 때문에 두 줄 이상 걸쳐서 작성할 경우 괄호를 절대 생략하지 말아야 한다.

● 별 출력

- 7각별꼴은 7과 서로소인 수가 1, 2, 3, 4, 5, 6 모두인 탓에 1, 6을 제외한 2, 3, 4, 5에 대응되는 별을 생성할 수 있는데, 2와 5에 대응되는 별과 3과 4에 대응하는 별은 모양이 같으므로 총 2개의 별을 만들 수 있다.
- n과 서로소인 자연수가 m개 있으면 $\frac{m}{2}-1$ 가지의 별을 생성할 수 있다.

408 GUE에서 도형 출력하기


GUE에서 도형 출력하는 방법은 앞에서 했던 내용과 많이 달라진다. `exp`, `sin` 함수를 괄호를 써서 작성해도 좋지만, 임시 변수에 출력해서 써도 좋기 때문이다. 어떻게 해야 최소한의 연산을 쓸 수 있을지 고민해 보는 것도 괜찮다.

■ X 모양 출력하기

앞에서 다른 X 모양 출력을 GUE에서 작성하는 방법은 큰 차이 없다. 다만, **Line OLED** 명령어를 사용한다는 차이점이 있다.


● 실수를 사용하는 경우

start>>							
	Line	OLED	20	20	60	60	0xFF0000
	Line	OLED	20	60	60	20	0xFF0000
end							

Geometry.s14	Result
<pre>start>> Line OLED 20 20 60 60 0xFF0000 Line OLED 20 60 60 20 0x00FF00</pre>	

● 복소수를 사용하는 경우

start>>					
	Line	OLED	$20+j*20$	$60+j*60$	0xFF0000
	Line	OLED	$20+j*60$	$60+j*20$	0xFF0000
end					


Geometry.s14	Result
<pre>start>> Line OLED 20+j*20 60+j*60 0xFF0000 Line OLED 20+j*60 60+j*20 0x00FF00</pre>	

■ 사각형 출력하기

사각형 출력 역시 GUE에서 작성하는 방법은 큰 차이 없다. 역시나 명령어에서 약간의 차이가 있는데, `OLED.Rectangle` 대신에 `Rectangle OLED`를 사용한다는 차이점만 있다.


● 정수를 사용하는 경우

start>>									
	<code>Rectangle</code>	<code>OLED</code>	<code>20</code>	<code>20</code>	<code>60</code>	<code>60</code>	<code>0xC000FF</code>	<code>0xC0FF00</code>	
end									

Geometry.s14	Result
<pre>start>> Rectangle OLED 20 20 60 60 0xC000FF 0xC0FF00</pre>	

● 복소수를 사용하는 경우

start>>									
	<code>Rectangle</code>	<code>OLED</code>	<code>20+j*20</code>	<code>60+j*60</code>	<code>0xC000FF</code>	<code>0xC0FF00</code>			
end									


Geometry.s14	Result
<pre>start>> Rectangle OLED 20+j*20 60+j*60 0xC000FF 0xC0FF00</pre>	

■ 정다각형 출력하기

정다각형 출력부터는 약간 달라진다. \sin 함수와 \cos 함수의 결과를 변수에 넣으면 좋겠지만, 예외적으로 수학 연산에 쓰이는 괄호는 가독성을 위해 생략하지 않기로 한다.

● 실수를 사용하는 경우


start>>					
	_n	IntVal	OLED	"a 입력 : "	
	Line	OLED	48 + 20 * cos(2*(k:1:n-1)*PI / n)		
			32 + 20 * sin(2*(k-1)*PI / n)		
			48 + 20 * cos(2*k*PI / n)		
			32 + 20 * sin(2*k*PI / n)		
			0xFF0000		
end					

Geometry.s14	Result
<pre>start>> _n IntVal OLED "a 입력 : " Line OLED 48 + 20 * $\cos(2*(k:1:n-1)*PI / n)$ 32 + 20 * $\sin(2*(k-1)*PI / n)$ 48 + 20 * $\cos(2*k*PI / n)$ 32 + 20 * $\sin(2*k*PI / n)$ 0xFF0000</pre>	<pre>a 입력 : 5</pre> 

GUE Style에서도 코드가 매우 길다면 여러 줄에 걸쳐서 작성해도 좋다. 여기서 특징이라면 반점 없이 줄바꿈만으로 다음 인수로 넘어갔다는 점에 주목하자. 인수가 너무 많으면 이렇게 작성해도 좋다.

● 복소수를 사용하는 경우


start>>					
	_n	IntVal	OLED	"a 입력 : "	
	Line	OLED	48+j*32 + 20 * exp(j*2*(k:1:n-1)*PI / n)		
			48+j*32 + 20 * exp(j*2*k*PI / n)		
			0xFF0000		
end					

Geometry.s14	Result
<pre>start>> _n IntVal OLED "a 입력 : " Line OLED 48+j*32 + 20 * $\exp(j*2*(k:1:n-1)*PI / n)$ 48+j*32 + 20 * $\exp(j*2*k*PI / n)$ 0xFF0000</pre>	<pre>a 입력 : 5</pre> 

■ 별모양 출력하기

별모양 출력하기 알고리즘은 정다각형 출력 알고리즘에서 파생되었다. 이번에는 사용자로부터 Step 값도 입력받아서 별모양을 좀 더 다양하게 출력할 수 있게 만들어 보기로 하겠다.

start>>				
	<code>_n</code>	<code>IntVal</code>	<code>OLED</code>	<code>"a 입력 : "</code>
	<code>_step</code>	<code>IntVal</code>	<code>OLED</code>	<code>"step 입력 : "</code>
	<code>Clear</code>	<code>OLED</code>		
	<code>Line</code>	<code>OLED</code>	<code>48+j*32 + 20 * exp(j*2*step*(k:1:n-1)*PI / n)</code>	
			<code>48+j*32 + 20 * exp(j*2*step*k*PI / n)</code>	
			<code>0xFF0000</code>	
end				

Geometry.s14	Result
<pre>start>> _n IntVal OLED "a 입력 : " _step IntVal OLED "step 입력 : " Clear OLED Line OLED 48+j*32 + 20 * exp(j*2*step*(k:1:n-1)*PI / n) 48+j*32 + 20 * exp(j*2*step*k*PI / n) 0xFF0000</pre>	<pre>a 입력 : 8 step 입력 : 3</pre> 

■ 별모양 출력 알고리즘의 한계

별모양 출력 알고리즘에 서로소가 아닌 수를 대입하면 어떤 일이 일어날까? 마치 약분한 것을 실행한 듯한 느낌을 가져오게 된다. 즉, 5-2로 하나 10-4로 하나 결과는 같다는 의미이다. 그 이유는 꼭짓점의 위치가 5개를 2칸씩 띄우는 것이나 10개를 4칸씩 띄우는 것이나 완전히 같기 때문이다.

Result1	Result2	Result3	Result4
<pre>a 입력 : 5 step 입력 : 2</pre> 	<pre>a 입력 : 10 step 입력 : 4</pre> 	<pre>a 입력 : 7 step 입력 : 3</pre> 	<pre>a 입력 : 14 step 입력 : 6</pre> 

다시 말해서 서로소가 아닌 수를 입력하게 되면 n각별꼴에서 최대공약수만큼 나눈 값만큼의 모양이 나오게 된다는 사실을 알 수 있다.

■ 정리

● GUE Style로 도형 출력 알고리즘 정리

- 짧은 구문인 경우 한 줄에 반점 없이 띄어쓰기로 구문을 구분할 수 있다.
- 긴 구문인 경우 줄 바꿈을 통해 구문을 구분할 수 있다.

Line	OLED	
		$48 + j * 32 + 20 * \exp(j * 2 * (k : 1 : n - 1) * \pi / n)$
		$48 + j * 32 + 20 * \exp(j * 2 * k * \pi / n)$
		0xFF0000

● 별모양 출력 정리

- 별모양 출력이 가능할 조건 : n과 step 값이 서로소여야 하고, step은 1 또는 n-1이 아니어야 한다.
- 서로소가 아닌 경우의 출력 : 최대공약수로 두 값을 나눈 것과 같은 결과가 출력된다. 예 - 10-6으로 입력할 때 최대공약수는 2이므로 5-3으로 적은 것과 같은 결과가 출력된다.

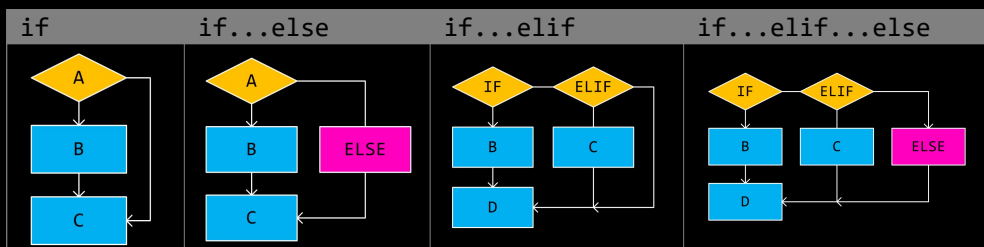
5. 조건문 - if/when

501 조건문 if의 이해

조건문은 **특정 조건이 참일 때** 실행하는 구문이다. 조건이 참이 되면 중괄호로 묶여 있는 구문을 실행한 후 넘어가고, 거짓이 되면 중괄호로 묶인 부분은 건너뛰고 다음 구문으로 넘어가게 된다.

■ 조건문의 구조

조건문은 크게 4가지 유형으로 나눌 수 있다. **if**만 단독으로 쓰이는 상황, **if**와 **else**를 사용하는 상황, **if...elif**를 사용하는 상황과 **if...elif...else**를 쓰는 상황이다. 여기서 중요한 점은 **if**, **elif**, **else**가 나오는 순서이다. 언제나 **if**가 가장 먼저 있어야 하고 **elif**는 그 다음에, **else**는 마지막에 나오게 된다.



■ if 계열의 조건문

if 계열의 조건문은 3가지가 있다. **if**는 조건이 참이면 실행, 거짓이면 실행하지 않는 구문이다. **elif**는 앞의 **if**, **elif**가 모두 거짓이고 조건이 참이면 실행, 앞의 **if**가 참이거나 **elif**의 내용이 거짓이면 실행하지 않는다. **else**는 앞의 **if**, **elif**가 모두 거짓이면 실행한다.

■ 정리

- **if** : 인수가 참이면 중괄호 단위로 실행, 거짓이면 넘어감.
- **elif** : 앞의 **if...elif**가 연달아 거짓이고, 인수가 참이면 실행, 둘 중 하나라도 아니면 넘어감.
- **else** : 앞의 **if...elif**가 연달아 모두 거짓이면 실행.

502 Full-Style에서 if문 사용하기

이번에는 조건문을 사용한 예제를 들어 보겠다. 사용자로부터 값을 입력받고 20대 인지, 그리고 20대라면 초반, 중반, 후반인지를 출력하는 코드를 예로 들어보겠다. 조건문의 순서는 20대 초반인지->중반인지->후반인지->모두 아닌지 순으로 가는 것이 가장 깔끔하다.

■ if문의 예제 - 나는 어디에 해당되는가?

아래는 if문을 활용하여 사용자로부터 값을 입력받은 뒤 어느 범위에 해당되는지 출력하는 코드이다.

Age.s14	Result
<pre>main(){ a = int; "나이 입력 : " >> OLED >> a << "\n"; if(a >= 20 && a <= 22){ OLED << "/r20대 초반\n"; } elif(a >= 23 && a <= 26){ OLED << "/r20대 중반\n"; } elif(a >= 27 && a <= 29){ OLED << "/r20대 후반\n"; } else{ OLED << "/r20대 아님\n"; } }</pre>	나이 입력 : 21 20대 초반
	나이 입력 : 25 20대 중반
	나이 입력 : 28 20대 후반
	나이 입력 : 33 20대 아님

실행 결과 조건에 따라 다르게 출력되는 것을 알 수 있다. 33을 입력했다면 앞의 모든 if와 elif의 조건을 만족하지 않아 else문을 실행하게 된다.

■ 비교 연산자

이번에 처음 등장하는 연산자는 비교 연산자이다. 비교 연산자는 6가지 종류가 있으며, 아래와 같다.

Priority	Operator	Description	Result
8	<	왼쪽 값이 오른쪽 값보다 작으면 참	0:False 1:True
	<=	왼쪽 값이 오른쪽 값보다 작거나 같으면 참	
	>	왼쪽 값이 오른쪽 값보다 크면 참	
	>=	왼쪽 값이 오른쪽 값보다 크거나 같으면 참	
9	==	왼쪽 값과 오른쪽 값이 같으면 참	
	!=	왼쪽 값과 오른쪽 값이 다르면 참	

■ 논리 연산자

다음으로 등장하는 연산자는 논리 연산자이다. AND를 의미하는 `&&`, OR를 의미하는 `||`, 그리고 NOT을 의미하는 `!`로 3가지 종류가 있다.

Priority	Operator	Description	Result
2	!	True 입력 시 False, False 입력 시 True 출력(값 반전)	0:False 1:True
13	&&	두 입력 모두 True여야 True 출력	
14		두 입력 중 어느 하나가 True일 때 True 출력.	

■ if문 사용 시 주의사항

NOT 연산자가 가장 우선순위에 있고, 비교 연산자가 그 중간 순위, 논리 연산자가 가장 나중 순위에 있다. 그래서 `!` 연산자를 사용한다면 뒤에 괄호를 쳐 주는 것도 좋은 방법이다.

■ 중괄호의 생략

if문을 작성할 때는 중괄호를 생략해도 상관없다. 다만, C언어와는 다르게 한 칸 이상 들여 써야 한다.

Age.s14	Result
<pre>main(){ a = int; "나이 입력 : " >> OLED >> a << "\n"; if(a >= 20 && a <= 22) OLED << "/r20대 초반\n"; elif(a >= 23 && a <= 26) OLED << "/r20대 중반\n"; elif(a >= 27 && a <= 29) OLED << "/r20대 후반\n"; else OLED << "/r20대 아님\n"; }</pre>	<div>나이 입력 : 21 20대 초반</div> <div>나이 입력 : 25 20대 중반</div> <div>나이 입력 : 28 20대 후반</div> <div>나이 입력 : 33 20대 아님</div>

Full-Style에서 **if**문은 원칙상 중괄호를 사용해야 하지만, 중괄호 없이도 위와 같이 작성해도 상관없다. 하지만, 부분적으로 **Starlic Style**로 변환되어 해석하기 때문에 들여쓰기 규칙이 적용된다.

■ 한 줄에 작성

if문을 한 줄에 작성할 때는 ``:``를 붙이고 작성한다. 이것 역시 ``:``를 붙이는 순간 부분적으로 **Pythonic Style**로 바뀌는 방식을 사용한다. 주의할 점은 ``:``를 붙인 뒤에는 한 칸 띄워 줘야 한다는 것이다.

Age.s14	Result
<pre>main(){ a = int; "나이 입력 : " >> OLED >> a << "\n"; if(a >= 20 && a <= 22): OLED << "/r20대 초반\n"; elif(a >= 23 && a <= 26): OLED << "/r20대 중반\n"; elif(a >= 27 && a <= 29): OLED << "/r20대 후반\n"; else: OLED << "/r20대 아님\n"; }</pre>	<div>나이 입력 : 21 20대 초반</div> <div>나이 입력 : 25 20대 중반</div> <div>나이 입력 : 28 20대 후반</div> <div>나이 입력 : 33 20대 아님</div>

■ 정리

● if문과 관련 있는 연산자

- 비교 연산자 : 값을 비교할 때 사용하는 연산자
 - 부등호로는 >=, <=, >, <가 있다.
 - 등호로는 ==, !=가 있다.
 - 등호를 == 대신 =를 사용해 대입으로 만드는 실수를 방지하기 위해 ==의 좌변에 의도적으로 상수를 넣기도 한다.
- 논리 연산자 : 비교해야 하는 연산자가 여러 가지가 있을 때 사용.
 - ! : 단항 연산자로 NOT 연산을 진행, 우선순위가 매우 높으므로 괄호를 사용하는 것을 추천한다.
 - && : 논리 AND 연산
 - || : 논리 OR 연산
 - &&과 ||이 같이 있다면 &&을 우선 연산한다.

● Full-Style에서 if문 사용

- if 뒤에는 조건을 적는다.
- if가 참일 때 실행할 구문은 중괄호를 추가하여 안에 적어준다.
- 중괄호를 사용하지 않고 if문을 작성 시에는 들여쓰기를 적용하면 된다.
- 한 줄에 작성할 때는 `:`를 붙이고 한 칸 띄운 후 그 뒤를 이어 주면 된다.

503 Starlic, Simple, Pythonic에서 if문 사용하기

이번에는 Starlic, Simple, Pythonic에서 if문을 사용해 보겠다. 이 스타일에서는 if문을 2가지 방법으로 사용할 수 있다. 따라서 취향에 맞게 선택하면 된다.

■ Starlic-Style

Starlic-Style에서 if문은 중괄호만 없애면 된다. 대신 들여쓰기 잘 되어 있는지 확인한다. 단순히 중괄호만 없앴을 뿐인데 코드가 매우 간결해지는 것을 알 수 있다. 다만, 세미콜론은 꼭 붙여준다. 한편, 아래와 같이 if문에 괄호를 생략해도 상관 없다.

Age.s14	Result
<pre>main() a = int; "나이 입력 : " >> OLED >> a << "\n"; if(a >= 20 && a <= 22) OLED << "/r20대 초반\n"; elif(a >= 23 && a <= 26) OLED << "/r20대 중반\n"; elif(a >= 27 && a <= 29) OLED << "/r20대 후반\n"; else OLED << "/r20대 아님\n";</pre>	<div>나이 입력 : 21 20대 초반</div> <div>나이 입력 : 25 20대 중반</div>
<pre>main() a = int; "나이 입력 : " >> OLED >> a << "\n"; if a >= 20 && a <= 22 OLED << "/r20대 초반\n"; elif a >= 23 && a <= 26 OLED << "/r20대 중반\n"; elif a >= 27 && a <= 29 OLED << "/r20대 후반\n"; else OLED << "/r20대 아님\n";</pre>	<div>나이 입력 : 28 20대 후반</div> <div>나이 입력 : 33 20대 아님</div>

■ Simple-Style

Simple-Style은 Starlic-Style의 코드에서 세미콜론과 main의 괄호만 없애면 된다.

Age.s14	Result
<pre>main a = int "나이 입력 : " >> OLED >> a << "\n" if(a >= 20 && a <= 22) OLED << "/r20대 초반\n" elif(a >= 23 && a <= 26) OLED << "/r20대 중반\n" elif(a >= 27 && a <= 29) OLED << "/r20대 후반\n" else OLED << "/r20대 아님\n"</pre>	<pre>나이 입력 : 21 20대 초반</pre>
<pre>main a = int "나이 입력 : " >> OLED >> a << "\n" if a >= 20 && a <= 22 OLED << "/r20대 초반\n" elif a >= 23 && a <= 26 OLED << "/r20대 중반\n" elif a >= 27 && a <= 29 OLED << "/r20대 후반\n" else OLED << "/r20대 아님\n"</pre>	<pre>나이 입력 : 25 20대 중반</pre>
<pre>main a = int "나이 입력 : " >> OLED >> a << "\n" if a >= 20 && a <= 22 OLED << "/r20대 초반\n" elif a >= 23 && a <= 26 OLED << "/r20대 중반\n" elif a >= 27 && a <= 29 OLED << "/r20대 후반\n" else OLED << "/r20대 아님\n"</pre>	<pre>나이 입력 : 28 20대 후반</pre>
<pre>main a = int "나이 입력 : " >> OLED >> a << "\n" if a >= 20 && a <= 22 OLED << "/r20대 초반\n" elif a >= 23 && a <= 26 OLED << "/r20대 중반\n" elif a >= 27 && a <= 29 OLED << "/r20대 후반\n" else OLED << "/r20대 아님\n"</pre>	<pre>나이 입력 : 33 20대 아님</pre>

■ Pythonic-Style

Pythonic-Style은 Simple-Style의 코드에서 **if**와 **main**에 ``:``를 붙이면 된다. 여기서 확인해야 할 사항은 **if**에도 ``:``를 붙인다는 점이다. 물론 안 붙인다고 컴파일 오류나는 것은 아니다.

Age.s14	Result
<pre>main: a = int "나이 입력 : " >> OLED >> a << "\n" if(a >= 20 && a <= 22): OLED << "/r20대 초반\n" elif(a >= 23 && a <= 26): OLED << "/r20대 중반\n" elif(a >= 27 && a <= 29): OLED << "/r20대 후반\n" else: OLED << "/r20대 아님\n"</pre>	<pre>나이 입력 : 21 20대 초반</pre>
<pre>main: a = int "나이 입력 : " >> OLED >> a << "\n" if a >= 20 && a <= 22: OLED << "/r20대 초반\n" elif a >= 23 && a <= 26: OLED << "/r20대 중반\n" elif a >= 27 && a <= 29: OLED << "/r20대 후반\n" else: OLED << "/r20대 아님\n"</pre>	<pre>나이 입력 : 25 20대 중반</pre>
<pre>main: a = int "나이 입력 : " >> OLED >> a << "\n" if a >= 20 && a <= 22: OLED << "/r20대 초반\n" elif a >= 23 && a <= 26: OLED << "/r20대 중반\n" elif a >= 27 && a <= 29: OLED << "/r20대 후반\n" else: OLED << "/r20대 아님\n"</pre>	<pre>나이 입력 : 28 20대 후반</pre>
<pre>main: a = int "나이 입력 : " >> OLED >> a << "\n" if a >= 20 && a <= 22: OLED << "/r20대 초반\n" elif a >= 23 && a <= 26: OLED << "/r20대 중반\n" elif a >= 27 && a <= 29: OLED << "/r20대 후반\n" else: OLED << "/r20대 아님\n"</pre>	<pre>나이 입력 : 33 20대 아님</pre>

■ Super-Simple Style

Super-Simple Style 자체가 main 함수를 없애고 적는 것이기 때문에 if문 안에서 Simple Style의 규칙이 적용된다.

Age.s14	Result
a = int "나이 입력 : " >> OLED >> a << "\n" if(a >= 20 && a <= 22) OLED << "/r20대 초반\n"	나이 입력 : 21 20대 초반
elif(a >= 23 && a <= 26) OLED << "/r20대 중반\n"	나이 입력 : 25 20대 중반
elif(a >= 27 && a <= 29) OLED << "/r20대 후반\n"	나이 입력 : 28 20대 후반
else OLED << "/r20대 아님\n"	나이 입력 : 33 20대 아님

■ 한 줄에 작성 - Pythonic Style 빌리기

Super-Simple Style이 아니라면 한 줄에 작성하는 것도 가능하다. 따라서 아래 코드 역시 유효하다. Simple Style에서 작성한 코드지만, Pythonic은 물론이고, Starlic에서도 사용가능하다. 주의할 점은 `:` 뒤에는 한 칸 띄워서 줄을 바꿨다는 것을 간접적으로 확인할 수 있게 작성해야 한다.

Age.s14	Result
main a = int "나이 입력 : " >> OLED >> a << "\n" if(a >= 20 && a <= 22): OLED << "/r20대 초반\n"	나이 입력 : 21 20대 초반
elif(a >= 23 && a <= 26): OLED << "/r20대 중반\n"	나이 입력 : 25 20대 중반
elif(a >= 27 && a <= 29): OLED << "/r20대 후반\n"	나이 입력 : 28 20대 후반
else: OLED << "/r20대 아님\n"	나이 입력 : 33 20대 아님

■ 정리

● 각각의 Style에서 if문

- Starlic : if문을 쓰고 중괄호의 영역에는 들여쓰기를 if문 안쪽으로 한다.
- Simple : Starlic과 동일하게 if문의 내용을 들여쓰기한다.
- Pythonic : if문 뒤에 ``:``를 찍고 Starlic, Simple과 동일하게 if문의 내용을 들여쓰기한다.

● if문의 괄호 생략

- Starlic, Simple, Pythonic 모두 if문의 괄호 생략이 가능하다.
- Starlic의 경우 괄호 생략을 안 하는 쪽을, 나머지 스타일은 if문의 괄호를 생략 하는 쪽이 보기 좋다.

● 한 줄 코드 작성

- Starlic, Simple, Pythonic 모두 ``:``를 쓰고 한 칸 띄 후 작성한다.

504 GUE-Style에서 if문 사용하기

GUE-Style에서의 if문을 익히고, 이를 표로 나타내는 방법을 소개하려고 한다. 지금까지 사용한 if문의 스타일과는 비슷하면서도 다르니 참고로 알아두자.

■ Simple-Style을 박아도 작동되나요?

GUE-Style에서 첫 코드는 `start>>` 안에 Simple-Style의 코드를 넣은 것이었다. 역시 여기에서도 Simple-Style의 코드를 넣으면 작동이 되는지 확인하고 다음으로 넘어간다. 물론 `start>>` 내부에 Simple-Style을 넣어도 잘 작동한다. 물론 표로 나타내기 곤란해서 그렇지.

Age.s14	Result
<code>start>></code>	
<code> a = int</code>	나이 입력 : 21 20대 초반
<code> "나이 입력 : " >> OLED >> a << "\n"</code>	
<code> if a >= 20 && a <= 22</code>	나이 입력 : 25 20대 중반
<code> OLED << "/r20대 초반\n"</code>	
<code> elif a >= 23 && a <= 26</code>	
<code> OLED << "/r20대 중반\n"</code>	나이 입력 : 28 20대 후반
<code> elif a >= 27 && a <= 29</code>	
<code> OLED << "/r20대 후반\n"</code>	
<code> else</code>	나이 입력 : 33 20대 아님
<code> OLED << "/r20대 아님\n"</code>	

■ GUE-Style에서 if문

GUE-Style에서 if문은 Simple-Style에서와 똑같이 작성하면 된다. 다만, 괄호는 치지 않는 쪽으로 작성한다.

■ GUE-Style로의 변환

● a값 입력받기

변경 전	변경 후
<code>a = int</code>	<code>_a IntVal OLED "나이 입력 : "</code>
<code>"나이 입력 : " >> OLED >> a << "\n"</code>	
<code>if a >= 20 && a <= 22</code>	<code>if a >= 20 && a <= 22</code>
	<code>if a>=20&&a<=22</code>
<code>OLED << "/r20대 초반\n"</code>	<code>Print OLED "/r20대 초반\n"</code>
start>>	
	<code>_a IntVal OLED "나이 입력 : "</code>
<code>if</code>	<code>a >= 20 && a <= 22</code>
	<code>Print OLED "/r20대 초반\n"</code>
<code>elif</code>	<code>a >= 23 && a <= 26</code>
	<code>Print OLED "/r20대 중반\n"</code>
<code>elif</code>	<code>a >= 27 && a <= 29</code>
	<code>Print OLED "/r20대 후반\n"</code>
<code>else</code>	
	<code>Print OLED "/r20대 아님\n"</code>
<code>end</code>	

Age.s14	Result
<code>start>></code>	
<code>_a IntVal OLED "나이 입력 : "</code>	나이 입력 : 21 20대 초반
<code>if a >= 20 && a <= 22</code>	
<code>Print OLED "/r20대 초반\n"</code>	나이 입력 : 25 20대 중반
<code>elif a >= 23 && a <= 26</code>	
<code>Print OLED "/r20대 중반\n"</code>	나이 입력 : 28 20대 후반
<code>elif a >= 27 && a <= 29</code>	
<code>Print OLED "/r20대 후반\n"</code>	나이 입력 : 33 20대 아님
<code>else</code>	
<code>Print OLED "/r20대 아님\n"</code>	

■ 정리

GUE-Style에서 **if**문은 Simple-Style에서와 똑같이 작성하면 된다.

● 표로 나타내기 위한 코드 가이드

- if문에 괄호를 치지 않고 작성한다.
- 띄어쓰기를 헛갈릴 수 있으니 연산자 간에는 띄어쓰기를 하지 않는 형태인 ``a>=20&&a<=22``의 형태도 권장한다.

● GUE Style로의 변환

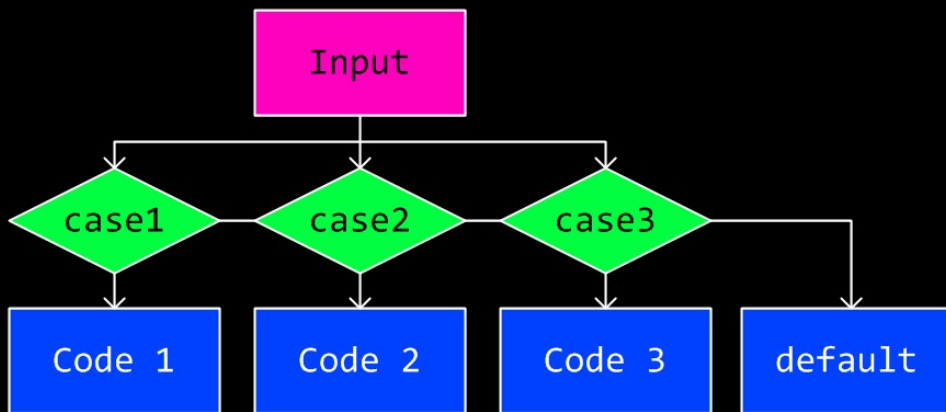
- 사용자로부터 값을 입력받는 과정은 ``_a IntVal OLED "나이 입력 : "``의 형태로 작성하면 된다.
- if문은 Simple Style의 것을 그대로 사용한다.
- OLED <<로 출력했다면 `Print OLED`로 바꿔준다.

505 조건문 when의 이해

when문은 값을 입력해서 특정 조건에 맞는 값이면 그 위치만을 실행하는 구문이다. C/C++에서의 **switch**문과 Python의 **match**문에 가깝다고 생각하면 된다.

■ When문의 구조

when문은 내부에 **case**문을 포함하여 **case**의 조건에 맞으면 그 구문을 실행한다. C언어에서의 **switch...case**는 중복 조건이 들어가면 컴파일 오류가 났던 것과 다르게 **when**문은 중복 조건이 들어가도 앞쪽의 조건에 맞으면 앞쪽을 우선으로 실행한다.



■ case문의 작성 - 일반

C/C++가 그랬듯 **case** 뒤에는 상수를 넣는 것을 원칙으로 한다. 그래서 변수의 값이 **case**의 값과 정확히 일치할 때 실행한다.

```
case(20){(코드)}
```

이렇게 작성하면 입력한 값이 20일 때 코드를 실행하게 된다. 20이 아니라면 다른 숫자 또는 **default**를 실행하게 되는 것이다.

■ case문의 작성 - 범위

case문은 범위로 작성해도 된다. 이때, 경수를 사용할 때 이 방식을 쓰기 좋다. 범위 작성은 반복 연산자를 사용했을 때와 비슷한 방법을 사용한다. 다만, `:` 연산자의 우선순위는 최하위가 된다.

```
case(20:30){(코드)}
```

이렇게 작성하면 입력한 값이 20일 때부터 30일 때까지는 이 코드를 실행하게 된다. 19 이하 또는 31 이상일 때는 이 코드가 실행되지 않는다. 주의할 점은 마지막 수인 30도 포함한다는 점이다.

■ case문의 작성 - 열린 구간

case문을 범위로 작성할 때 범위가 열려 있으면 상당히 곤란해질 때가 있다. 이럴 때는 부등호를 포함해서 작성해도 좋다.

```
case(<=20){(코드)}
```

이렇게 작성하면 20보다 작거나 같을 때 실행된다는 의미가 된다. 20까지는 이 코드가 실행되고, 21부터는 이 코드가 실행되지 않는 것이다.

■ 정리

when문에는 값이 들어가서 값이 조건에 맞으면 조건에 대한 구문을 실행하고, 모두 맞지 않을 때는 **default** 구문을 실행한다.

● **When**문의 구조

- **when** 뒤에는 변수나 수식 등 정수(또는 소수 등) 값을 넣을 수 있는 식을 인수로 넣는다.
- **when** 내부에는 **case**를 작성해서 경우에 따라 실행하게 설계한다.

● **case**문의 작성

- **case** 안에 숫자를 단독으로 넣는 경우 그 값과 일치할 때 실행한다.
- **case** 안에 20:30과 같이 범위를 넣을 수 있다. 이때, 처음 값과 끝값은 모두 포함한다.
- **case** 안에 범위가 열려 있으면 부등호를 포함해서 넣을 수 있다. ≥ 20 , >20 , ≤ 20 , <20 등이 사용가능하다.

506 Full-Style에서 when문 사용하기

앞에서 다른 나이를 입력하여 20대 어느 위치인지 확인하는 코드를 이번에는 when 문으로 구현할 것이다.

■ when문의 예제 - 나는 어디에 해당되는가?

아래는 when문을 활용하여 사용자로부터 값을 입력받은 뒤 어느 범위에 해당되는지 출력하는 코드이다.

Age.s14	Result
<pre>main(){ a = int; "나이 입력 : " >> OLED >> a << "\n"; when(a){ case(20:22){ OLED << "\r20대 초반\n"; } case(23:26){ OLED << "\r20대 중반\n"; } case(27:29){ OLED << "\r20대 후반\n"; } else{ OLED << "\r20대 아님\n"; } } }</pre>	나이 입력 : 21 20대 초반
	나이 입력 : 25 20대 중반
	나이 입력 : 28 20대 후반
	나이 입력 : 33 20대 아님

when문을 사용해서 범위를 간편하게 정해 출력할 수 있었다. 하지만, 중괄호가 많이 감싸지면서 코드의 부피가 커지는 문제가 있다. 그래서 다음 페이지에서는 조금 더 간략하게 C언어의 느낌을 살리면서 작성하는 방법을 소개한다.

■ when문의 간략화 - C언어처럼 쓸 순 없는 걸까?

C언어처럼 **case**문을 작성하는 방법은 아래와 같다. C언어처럼 쓸 순 있지만, 들여쓰기를 꼭 지켜야 한다. 그 이유는 **case**문 내부는 **Full-Pythonic** 규칙이 일시적으로 실행되기 때문이다.

Age.s14	Result
<pre>main(){ a = int; "나이 입력 : " >> OLED >> a << "\n"; when(a){ case 20:22: OLED << "/r20대 초반\n"; case 23:26: OLED << "/r20대 중반\n"; case 27:29: OLED << "/r20대 후반\n"; default: OLED << "/r20대 아님\n"; } }</pre>	<div>나이 입력 : 21 20대 초반</div>
	<div>나이 입력 : 25 20대 중반</div>
	<div>나이 입력 : 28 20대 후반</div>
	<div>나이 입력 : 33 20대 아님</div>

이 방식을 C언어랑 비교하면 **switch** 대신 **when**이 쓰인 점과 **case** 뒤에 범위가 쓰인 것 정도의 차이가 있다. **default** 대신 **else**를 써도 프로그램상에 문제가 발생하지는 않는다. 확실히 **case**문에 중괄호를 썼을 때보다 코드가 간결해졌음을 알 수 있다.

■ 정리

● when문의 구조

- when문 안에는 case문을 추가로 넣어 when의 값이 case문의 값 또는 범위에 맞으면 그 구문을 실행한다.

● 범위를 넣는 case문

- case 20:22와 같이 작성하면 20, 21, 22일 때 구문이 실행되어 20대 초반이 출력된다.

● default 또는 else

- else 또는 default를 사용하면 case문의 범위 또는 값이 모두 일치하지 않을 때가 실행된다. default를 쓰는 것은 C/C++언어에서, else를 쓰는 것은 Kotlin 문법이기도 하다.

● when문의 문제점과 간략화

- when문을 사용하면 코드의 부피가 지나치게 커지는 스폰지 현상이 발생할 수 있다. 따라서 부피를 줄이기 위해 부분적으로 Pythonic 스타일을 사용할 수 있다.
- 다행히 Pythonic Style에는 Starlic Style의 문법도 호환되므로 case XX:로 시작하는 C와 비슷한 형태로 작성할 수 있다.
- case XX:의 형태를 사용한다면 들여쓰기 규칙을 준수해야 한다.

507 Starlic, Simple, Pythonic에서 when문 사용하기

이번에는 앞에서 다룬 when문 코드를 Starlic, Simple, Pythonic Style에서 표현해 보려고 한다.

■ Starlic-Style : 중괄호를 없애라!

Starlic-Style은 중괄호를 없애고 작성하면 된다. 다만, case문은 가독성 향상을 위해 Pythonic하게 작성하는 쪽을 더 권장한다.

Age.s14	Result
<pre>main() a = int; "나이 입력 : " >> OLED >> a << "\n"; when(a) case 20:22: OLED << "/r20대 초반\n"; case 23:26: OLED << "/r20대 중반\n"; case 27:29: OLED << "/r20대 후반\n"; default: OLED << "/r20대 아님\n";</pre>	<pre>나이 입력 : 21 20대 초반 나이 입력 : 25 20대 중반 나이 입력 : 28 20대 후반 나이 입력 : 33 20대 아님</pre>

이번에는 Starlic-Style로 변환하기 위해 중괄호를 없앨 뿐 아니라 when문의 내부를 상당히 고친 것도 확인할 수 있다. 사실은 C언어처럼 표기하는 방식을 그대로 가져왔다고 해석하는 쪽이 좋다. case문만 Pythonic Style로 작성함으로써 어떤 상황에서 작동하는지 직관적으로 확인할 수 있다는 장점이 있다.

■ Starlic-Style : 한 줄에 작성하는 경우

case문에서 작동하는 문장이 하나인 경우 한 줄 코드로 작성하는 것이 더욱 간결하고 깔끔하다. **Starlic**에서도 한 줄 코드를 작성할 수 있다. 앞에서 했던 것처럼 **`:`**를 적고 한 칸 띄워서 작성하면 된다.

Age.s14	Result
main()	나이 입력 : 21 20대 초반
a = int;	
"나이 입력 : " >> OLED >> a << "\n";	나이 입력 : 25 20대 중반
when(a)	
case 20:22: OLED << "/r20대 초반\n";	나이 입력 : 28 20대 후반
case 23:26: OLED << "/r20대 중반\n";	
case 27:29: OLED << "/r20대 후반\n";	나이 입력 : 33 20대 아님
default: OLED << "/r20대 아님\n";	

■ Simple-Style : 세미콜론 지우기

이번에는 **Starlic-Style**에서 세미콜론을 지워서 **Simple-Style**로 고쳐 본다. **main**함수의 ()와 **when**의 ()도 지워서 더 간결하게 작성할 수 있다.

Age.s14	Result
main	나이 입력 : 21 20대 초반
a = int	
"나이 입력 : " >> OLED >> a << "\n"	
when a	
case 20:22:	나이 입력 : 25 20대 중반
OLED << "/r20대 초반\n"	
case 23:26:	나이 입력 : 28 20대 후반
OLED << "/r20대 중반\n"	
case 27:29:	나이 입력 : 33 20대 아님
OLED << "/r20대 후반\n"	
default:	
OLED << "/r20대 아님\n"	

■ Simple-Style : 한 줄에 작성

이번에는 Simple-Style에서도 한 줄에 작성하는 코드를 구현해 본다. 역시 코드를 압축하는 효과가 있어 훨씬 간결해 보인다.

Age.s14	Result
main	
a = int	
"나이 입력 : " >> OLED >> a << "\n"	나이 입력 : 21 20대 초반
when a	
case 20:22: OLED << "/r20대 초반\n"	나이 입력 : 25 20대 중반
case 23:26: OLED << "/r20대 중반\n"	나이 입력 : 28 20대 후반
case 27:29: OLED << "/r20대 후반\n"	나이 입력 : 33 20대 아님
default: OLED << "/r20대 아님\n"	

■ Super-Simple-Style : main 없이 작성

이번에는 main 함수 없이 함수 밖에서 작성해 본다. when문 내부부터는 simple-style이 적용되어 사실상 main함수 빼고 들여쓰기만 없애면 된다. 다만, 주의할 점은 when문에 a를 쓸 때는 괄호를 쳐 주어야 한다.

Age.s14	Result
a = int	
"나이 입력 : " >> OLED >> a << "\n"	나이 입력 : 21 20대 초반
when(a)	
case 20:22: OLED << "/r20대 초반\n"	나이 입력 : 25 20대 중반
case 23:26: OLED << "/r20대 중반\n"	나이 입력 : 28 20대 후반
case 27:29: OLED << "/r20대 후반\n"	나이 입력 : 33 20대 아님
default: OLED << "/r20대 아님\n"	

■ Pythonic-Style : main, when에 `:` 붙이기

Pythonic Style로 바꿀 때는 Simple-style에서 **main**과 **when**에만 `:`를 붙이면 된다.

Age.s14	Result
main:	
a = int	나이 입력 : 21 20대 초반
"나이 입력 : " >> OLED >> a << "\n"	
when a:	
case 20:22:	나이 입력 : 25 20대 중반
OLED << "/r20대 초반\n"	
case 23:26:	나이 입력 : 28 20대 후반
OLED << "/r20대 중반\n"	
case 27:29:	나이 입력 : 33 20대 아님
OLED << "/r20대 후반\n"	
default:	
OLED << "/r20대 아님\n"	

■ Pythonic-Style : 한 줄에 작성

Pythonic Style에서도 앞에서 했던 것처럼 한 줄에 작성하는 것이 가능하다.

Age.s14	Result
main:	
a = int	나이 입력 : 21 20대 초반
"나이 입력 : " >> OLED >> a << "\n"	
when a:	
case 20:22: OLED << "/r20대 초반\n"	나이 입력 : 25 20대 중반
case 23:26: OLED << "/r20대 중반\n"	나이 입력 : 28 20대 후반
case 27:29: OLED << "/r20대 후반\n"	나이 입력 : 33 20대 아님
default: OLED << "/r20대 아님\n"	

■ 정리

● Starlic Style에서의 when 작성 요령

- Starlic Style에서는 중괄호를 없애면 된다. 하지만, 가독성을 위해서 case문 만큼은 괄호를 없애고 Pythonic Style로 작성한다.

● Simple Style과 Pythonic Style에서 when 작성 요령

- Simple Style에서는 세미콜론을 없애면 된다.
- Pythonic Style에서는 main과 when, case에 :를 붙이면 된다.

● 한 줄 코드 가이드

- case문에 구문이 1개 사용된다면 한 줄 코드를 사용한다.
- 한 줄 코드는 `:` 사용 후 한 칸 띄고 작성하면 된다.

508 GUE-Style에서 when문 사용하기

이번에는 GUE-Style에서 **when**문을 사용하는 방법을 알아보겠다. GUE-Style에서는 **Pythonic-Style**을 도입하는 순간 표로 나타내는 형태가 깨지므로 예외적으로 GUE-Style 안에서만 모든 코드를 작성하는 쪽을 권장한다.

■ Simple-Style도 작동될까?

Simple-Style 코드를 **start>>** 안에 넣으면 결론적으로 작동되긴 한다. 하지만, **case**문 내부가 **Pythonic**하게 작성되어 취지에 맞지 않는 코드가 된다. 오류가 나는 것은 아니니 한 번 코드를 참고만 해 두자.

Age.s14	Result
<pre>start>> a = int "나이 입력 : " >> OLED >> a << "\n" when a case 20:22: OLED << "/r20대 초반\n" case 23:26: OLED << "/r20대 중반\n" case 27:29: OLED << "/r20대 후반\n" default: OLED << "/r20대 아님\n"</pre>	<pre>나이 입력 : 21 20대 초반 나이 입력 : 25 20대 중반 나이 입력 : 28 20대 후반 나이 입력 : 33 20대 아님</pre>

■ GUE-Style로 변환하기

이번에는 GUE-Style에 맞게 코드를 변환하는 작업을 해 보겠다. 대응되는 코드는 앞에서 설명했으니 간단히 짚고 넘어간다.

● a값 입력받기

변경 전		변경 후		
a = int		_a IntVal OLED "나이 입력 : "		
"나이 입력 : " >> OLED >> a << "\n"				
when a		when a		
case 20:22:		case 20:22		
OLED << "/r20대 초반\n"		Print OLED "/r20대 초반\n"		
start>>				
	_a	IntVal	OLED	"나이 입력 : "
	when	a		
	case	20:22		
		Print	OLED	"/r20대 초반\n"
	case	23:26		
		Print	OLED	"/r20대 중반\n"
	case	27:29		
		Print	OLED	"/r20대 후반\n"
	default			
		Print	OLED	"/r20대 아님\n"
end				

Age.s14		Result
start>>		
_a IntVal OLED "나이 입력 : "		나이 입력 : 21 20대 초반
when a		
case 20:22		
Print OLED "/r20대 초반\n"		나이 입력 : 25 20대 중반
case 23:26		
Print OLED "/r20대 중반\n"		나이 입력 : 28 20대 후반
case 27:29		
Print OLED "/r20대 후반\n"		
default		
Print OLED "/r20대 아님\n"		나이 입력 : 33 20대 아님

■ 정리

GUE-Style에서 **when**문은 Simple-Style에서와는 다르게 Pythonic Style 구문을 사용하지 않는 방향으로 작성한다. 한 줄 코드 역시 불가능하다.

● 표로 나타내기 위한 코드 가이드

- **case**문 끝에 ``:``를 붙이지 않는다. 붙이는 순간 Pythonic이 되어 **Print OLED** 명령어 사용이 불가능하다.

● GUE Style로의 변환

- 사용자로부터 값을 입력받는 과정은 ``_a IntVal OLED "나이 입력 : "``의 형태로 작성하면 된다.
- **when** 구문은 **when** a의 형태를 그대로 사용한다.
- **case** 구문은 끝에 ``:``가 있다면 지운다.
- **OLED** <<로 출력했다면 **Print OLED**로 바꿔준다.

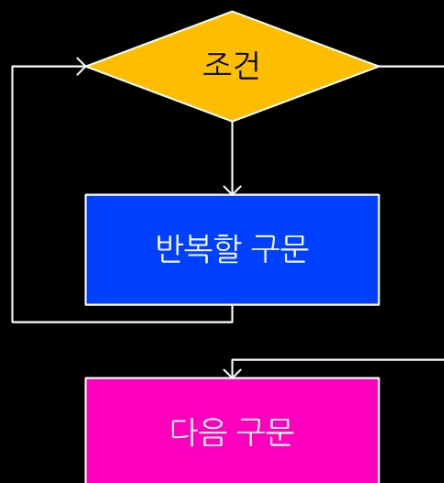
6. 반복문 - while/for

601 반복문 while의 이해

반복문은 조건에 따라 반복 실행하는 구문을 의미한다. 반복을 몇 번 했는지 알 수 있어야 할 때가 있는데, 이것을 **index**라고 한다. 보통은 **i**로 표현하는데, 이로 인해 복소수 단위는 **j**로 표현한다. **index**를 다루는 프로그램은 **for**문에서 더 자세히 다루고, 여기서는 무한 반복 구문 위주로 다뤄볼 예정이다.

■ 조건이 참이면 계속 반복하는 구문, while

while문은 조건을 검사해서 참(0이 아닌 값)이면 구문을 실행하고 다시 조건을 검사해서 참이면 구문을 또 실행하는 과정을 반복한다. 그래서 조건을 검사했을 때 계속 참이면 계속 반복한다.



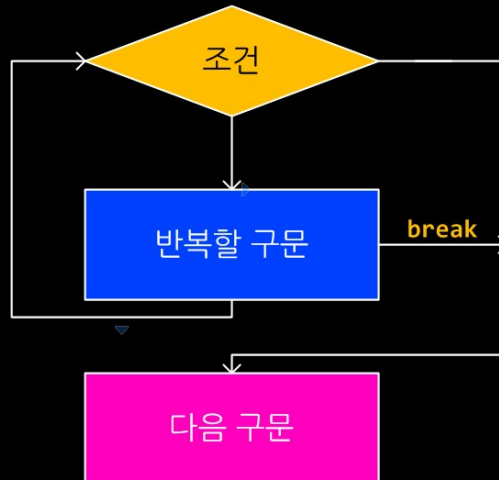
■ 무한 반복

while문은 조건에 1을 넣으면 언제나 참이 되기 때문에 무한 반복을 하게 된다. 그러면 이 구문은 아래와 같은 형태로 간략화할 수 있다.



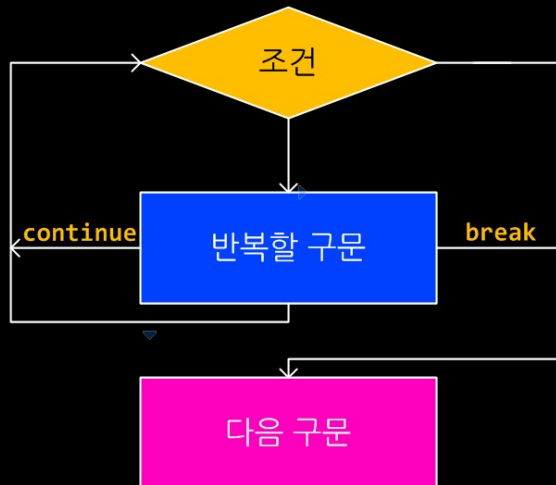
■ 반복문에서 빠져나가는 방법 - break

반복문에서 고의로 탈출하는 방법이 있다. 바로 **break**를 사용하면 된다. **break**를 사용하면 그 지점에서 반복문 다음 구문으로 넘어간다. 심지어 무한 반복이어도 이 **break**를 사용하면 반복문을 탈출하고 다음 구문으로 넘어갈 수 있다.



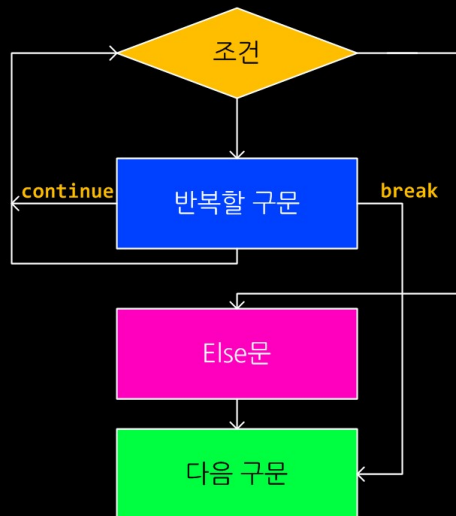
■ 반복문을 처음부터 다시 실행하는 continue문

다음으로는 반복문의 조건 부분에서부터 다시 시작하는 **continue**가 있다. **continue**를 실행하면 조건 부분부터 다시 시작해 조건이 맞는지 확인 후 반복 구문을 다시 실행한다.



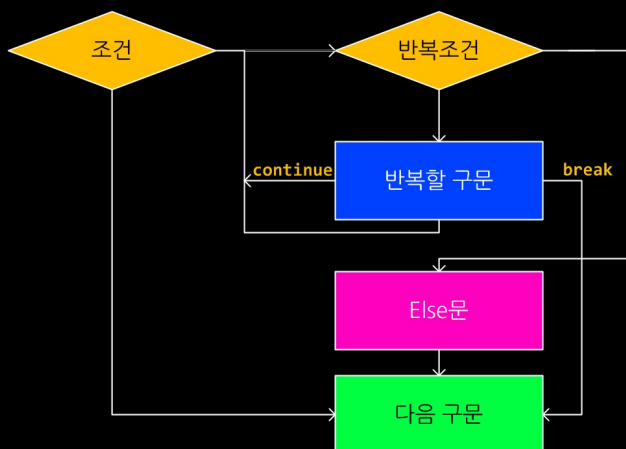
■ 반복문에도 else를 쓸 수 있나요?

Python에서 반복문에 **else**를 쓰면 **break**가 걸리지 않을 때 실행한다고 정의하였다. Starlit을 설계할 때 이것 고려하지 않고 설계했지만, 놀랍게도 Python과 완벽하게 똑같이 사용할 수 있다. **break**가 걸리면 **else**를 실행하지 않는데, **break**가 걸리지 않고 조건이 만족되지 못해서 빠져나갔을 때만 **else**를 실행한다.



■ elwhile문

if문도 **elif**와 **else**가 있었듯이 **while**문도 **while**과 **else**가 있으면 **elwhile**이 있다. **elwhile**은 앞의 **while** 또는 **if**문이 거짓일 때에만 작동하는 **while**문이다. 앞의 **if**문이 참이거나 **while**문이 처음부터 거짓이라면 이 구문이 실행되지 않는다.



■ 정리

● While문의 정의

- 조건이 참일 때까지 계속 반복하는 구문
- while(1)을 사용하면 무한 반복을 구현할 수 있다.

● break와 continue

- break는 반복문을 탈출할 때 사용한다.
- continue는 반복문을 앞에서부터 계속 반복할 때 사용한다.

● else와 elwhile

- 반복문에서 else는 break로 중간에 탈출한 것이 아닌 반복조건을 만족하지 못해 탈출했을 때 실행한다.
- elwhile은 앞의 반복문에서 else가 실행되거나 앞의 조건문이 실행되지 않았을 때 실행하는 반복문이다. 하지만, 잘 사용되지 않으니 참고용으로 알아두자.

602 Full-Style에서 while문 사용하기

2~4장에서는 while문을 사용하여 등비급수의 합의 근삿값을 구하는 예제를 들어 보겠다. 예를 들어 등비급수 $1 + \frac{1}{2} + \frac{1}{4} + \dots$ 의 값은 2임이 알려져 있다. 이게 진짜로 맞는지 확인하기 위해 이 알고리즘을 구현한다. 이때, 급수의 합은 1보다 언제나 크고, float형은 소숫점 아래 6자리까지 정밀하므로 0.000001보다 누적 값이 작으면 프로그램을 종료하게 만들면 된다.

사용자로부터 첫째항과 공비를 받은 뒤 공비가 0~1 사이이면 등비급수의 합을 구하면 된다. -1~1일 때가 아니라 0~1 사이로 사용하는 것은 값이 계속 증가하게 해서 정확하게 출력하는 것을 보장하기 위해서다. 언제까지 하나면 첫째항의 0.000001배보다 누적 값이 작아질때까지 반복한다.

■ while문 조건과 예제

첫째항의 0.000001배보다 작을 때 반복문을 멈추므로 반복조건은 그 반대인 첫째항의 0.000001배보다 큰 상황에서 반복해야 한다. 첫째항을 `a1`, 공비를 `r`, 합을 `an`이라고 놓으면 조건은 ``an > a1 * 0.000001``이라고 한다. 연산량을 줄이기 위해 ``thr = a1 * 0.000001``을 추가한 뒤 ``an > thr``이라고 적어도 좋다. GUE에서는 이 방법을 사용할 것이다.

Age.s14	Result
<pre>main(){ a1 = float; r = float; "첫째항 : " >> OLED >> a1 << "\n"; "공비 : " >> OLED >> r << "\n"; thr = a1 * 0.000001; sum = float; if(r <= 0 r >= 1){ OLED << "/r발산!"; } else{ while(a1 > thr){ sum += a1; a1 *= r; } OLED << f"/g급수의 합= {sum:.4f}"; } }</pre>	<pre>첫째항 : 1 공비 : 0.5 급수의 합= 2.0000</pre>
	<pre>첫째항 : 1 공비 : 0.333333 급수의 합= 1.5000</pre>
	<pre>첫째항 : 2 공비 : 2 발산!</pre>

■ 실행 결과 분석

a1과 r은 각각 첫째항과 공비를 받는다. a1의 값은 r을 계속 곱하면서 sum에 누적하면서 등비급수의 합을 구할 수 있다. 이때, r을 계속 곱하다보면 a1은 기하급수적으로 작아지는데, 어느순간부터 유효숫자의 범위보다 a1이 작아지는 순간이 발생한다(a1 <= thr). 이때 반복문을 탈출하여 급수의 합을 출력한다. 양수 r에 대해서 급수가 수렴하는 범위는 0과 1 사이(1은 포함하지 않음)이므로 while문 전에 r의 범위를 확인하고 급수의 합을 구하게 된다.

■ 무한반복 후 반복문 탈출시키기

break문을 활용해서 앞의 코드를 그대로 구현할 수도 있다. 아래와 같이 작성해보자.

Age.s14	Result
<pre>main(){ a1 = float; r = float; "첫째항 : " >> OLED >> a1 << "\n"; "공비 : " >> OLED >> r << "\n"; thr = a1 * 0.000001; sum = float; if(r <= 0 r >= 1){ OLED << "/r발산!"; } else{ while(1){ sum += a1; a1 *= r; if(a1 <= thr){break;} } OLED << f"/g급수의 합= {sum:.4f}"; } }</pre>	<pre>첫째항 : 1 공비 : 0.5 급수의 합= 2.0000</pre>
	<pre>첫째항 : 1 공비 : 0.333333 급수의 합= 1.5000</pre>
	<pre>첫째항 : 2 공비 : 2 발산!</pre>

■ 코드 부피 줄이기

Full-Style은 그 특성상 **if**나 **while**을 사용하면 코드의 부피가 늘어나는 **Sponge Effect**가 발생한다. 그래서 잠시 **Starlic Style**을 빌려서 중괄호를 일부분만 생략해볼 수도 있다. 아래는 **while**문과 **break**문에 한해서 **Starlic Style**과 **Pythonic Style**을 빌려 작성해 보았다.

Age.s14	Result
<pre> main(){ a1 = float; r = float; "첫째항 : " >> OLED >> a1 << "\n"; "공비 : " >> OLED >> r << "\n"; thr = a1 * 0.000001; sum = float; if(r <= 0 r >= 1){ OLED << "/r발산!"; } else{ while(1) sum += a1; a1 *= r; if(a1 <= thr): break; OLED << f"/g급수의 합= {sum:.4f}"; } } </pre>	<pre> 첫째항 : 1 공비 : 0.5 급수의 합= 2.0000 </pre>
	<pre> 첫째항 : 1 공비 : 0.333333 급수의 합= 1.5000 </pre>
	<pre> 첫째항 : 2 공비 : 2 발산! </pre>

■ 버튼 인식하기 예제

`BUTTON_Read` 함수를 사용해서 버튼을 인식해보는 예제를 알아보겠다. `while`문에 `!BUTTON_Read()`를 넣어주면 버튼 누를 때까지 기다린다는 의미가 된다. 이를 이용해 어떤 버튼을 눌렀는지 출력하는 프로그램을 작성하면 아래와 같다.

Age.s14	Result
<pre>main(){ btn = int; while(!(btn = BUTTON_Read())){} when(btn){ case BUTTON_0: OLED << "/r0번 버튼 입력!"; case BUTTON_1: OLED << "/g1번 버튼 입력!"; case BUTTON_2: OLED << "/y2번 버튼 입력!"; } }</pre>	<div>0번 버튼 입력!</div> <div>1번 버튼 입력!</div> <div>2번 버튼 입력!</div>

■ 버튼 인식 함수의 사용

`BUTTON_Read` 함수는 `while`문 안에서 사용해서 실시간으로 버튼이 눌렀는지 감지할 수 있다. 위의 코드처럼 구동한다면 버튼 누를 때까지 기다렸다가 눌렀으면 변수 `btn`값을 확인해 사용하는 것이고, 아예 아래처럼 이 구문 자체를 `while`문 안에 넣어 계속 버튼을 누르게 구성할 수도 있다(프로그램 종료가 되지 않으니 주의!).

Age.s14	Result
<pre>main(){ btn = int; while(1){ btn = BUTTON_Read(); when(btn){ case BUTTON_0: OLED << "/r0번 버튼 입력!"; case BUTTON_1: OLED << "/g1번 버튼 입력!"; case BUTTON_2: OLED << "/y2번 버튼 입력!"; } } }</pre>	<div>0번 버튼 입력! 1번 버튼 입력! 2번 버튼 입력!</div> <div>0번 버튼 입력! 1번 버튼 입력! 2번 버튼 입력!</div> <div>0번 버튼 입력! 1번 버튼 입력! 2번 버튼 입력!</div>

■ 정리

● while문의 사용

while문을 사용하면 특정 조건이 참일 때까지 계속 반복하게 만들 수 있다. 이를 이용해 등비급수의 합을 구하는 알고리즘을 구현할 수 있다.

● 반복문 탈출

break를 사용하여 반복문을 탈출할 수 있다. 반복문을 탈출하면 else를 실행하지 않으며, 반복문 조건이 거짓이라서 탈출했을 때 else가 작동한다.

● 버튼 함수의 사용

- BUTTON_Read 함수를 사용하여 버튼의 입력을 감지할 수 있다.
- BUTTON_Read 함수는 while문 안에서 사용한다.

603 Starlic, Simple, Pythonic에서 while문 사용하기

3장에서는 2장에서 다룬 예제 3가지를 **Starlic**, **Simple**, **Pythonic**에서 작성해 볼 차례이다. 등비급수의 합 구하는 예제, 그리고 버튼을 인식하는 예제를 **Starlic**, **Simple**, **Pythonic**으로 바꾸면서 **while**문을 사용하는 방법을 알아보자.

■ Starlic - 등비급수 예제

Starlic으로 등비급수 작성하는 코드는 아래와 같다. 중괄호를 완전히 없애고 들여쓰기를 확인해 주자. 아래의 OLED <<가 있는 부분은 들여쓰기 위치가 맞는지 확인하고 참고하자. 만약에 이걸 맞추기 어렵다 싶으면 | 사용법을 보조적으로 사용하는 것도 좋은 방법이다.

Age.s14	Result
<pre>main() a1 = float; r = float; "첫째항 : " >> OLED >> a1 << "\n"; "공비 : " >> OLED >> r << "\n"; thr = a1 * 0.000001; sum = float; if(r <= 0 r >= 1) OLED << "/r발산!"; else while(1) sum += a1; a1 *= r; if(a1 <= thr): break; OLED << f"/g급수의 합= {sum:.4f}";</pre>	<pre>첫째항 : 1 공비 : 0.5 급수의 합= 2.0000</pre>
<pre>main() a1 = float; r = float; "첫째항 : " >> OLED >> a1 << "\n"; "공비 : " >> OLED >> r << "\n"; thr = a1 * 0.000001; sum = float; if(r <= 0 r >= 1) OLED << "/r발산!"; else while(1) sum += a1; a1 *= r; if(a1 <= thr): break; OLED << f"/g급수의 합= {sum:.4f}";</pre>	<pre>첫째항 : 1 공비 : 0.333333 급수의 합= 1.5000</pre>
	<pre>첫째항 : 2 공비 : 2 발산!</pre>

■ Simple - 등비급수 예제

Simple 스타일로 등비급수 작성하는 코드는 아래와 같다. 세미콜론만 지우면 된다. 역시 들여쓰기가 헛갈린다면 `|` 표기법의 사용도 추천한다.

Age.s14	Result
<pre> main() a1 = float r = float "첫째항 : " >> OLED >> a1 << "\n" "공비 : " >> OLED >> r << "\n" thr = a1 * 0.000001 sum = float if(r <= 0 r >= 1) OLED << "/r발산!" else while(1) sum += a1 a1 *= r if(a1 <= thr): break OLED << f"/g급수의 합= {sum:.4f}" </pre>	<pre> 첫째항 : 1 공비 : 0.5 급수의 합= 2.0000 </pre>
<pre> main() a1 = float r = float "첫째항 : " >> OLED >> a1 << "\n" "공비 : " >> OLED >> r << "\n" thr = a1 * 0.000001 sum = float if(r <= 0 r >= 1) OLED << "/r발산!" else while(1) sum += a1 a1 *= r if(a1 <= thr): break OLED << f"/g급수의 합= {sum:.4f}" </pre>	<pre> 첫째항 : 1 공비 : 0.333333 급수의 합= 1.5000 </pre>
<pre> main() a1 = float r = float "첫째항 : " >> OLED >> a1 << "\n" "공비 : " >> OLED >> r << "\n" thr = a1 * 0.000001 sum = float if(r <= 0 r >= 1) OLED << "/r발산!" else while(1) sum += a1 a1 *= r if(a1 <= thr): break OLED << f"/g급수의 합= {sum:.4f}" </pre>	<pre> 첫째항 : 2 공비 : 2 발산! </pre>

■ Pythonic - 등비급수 예제

Pythonic 스타일로 등비급수 작성하는 코드는 아래와 같다. **if**, **else**, **while** 등에 **:**를 붙이면 된다.

Age.s14	Result
<pre> main a1 = float r = float "첫째항 : " >> OLED >> a1 << "\n" "공비 : " >> OLED >> r << "\n" thr = a1 * 0.000001 sum = float if r <= 0 r >= 1: OLED << "/r발산!" else: while 1: sum += a1 a1 *= r if a1 <= thr: break OLED << f"/g급수의 합= {sum:.4f}" </pre>	<pre> 첫째항 : 1 공비 : 0.5 급수의 합= 2.0000 </pre>
<pre> main a1 = float r = float "첫째항 : " >> OLED >> a1 << "\n" "공비 : " >> OLED >> r << "\n" thr = a1 * 0.000001 sum = float if r <= 0 r >= 1: OLED << "/r발산!" else: while 1: sum += a1 a1 *= r if a1 <= thr: break OLED << f"/g급수의 합= {sum:.4f}" </pre>	<pre> 첫째항 : 1 공비 : 0.333333 급수의 합= 1.5000 </pre>
<pre> main a1 = float r = float "첫째항 : " >> OLED >> a1 << "\n" "공비 : " >> OLED >> r << "\n" thr = a1 * 0.000001 sum = float if r <= 0 r >= 1: OLED << "/r발산!" else: while 1: sum += a1 a1 *= r if a1 <= thr: break OLED << f"/g급수의 합= {sum:.4f}" </pre>	<pre> 첫째항 : 2 공비 : 2 발산! </pre>

■ Super-Simple - 등비급수 예제

Simple-Style에서 main함수를 빼면 된다. 이때, if문의 괄호는 없애면 오류가 날 수 있으니 유의해야 한다.

Age.s14	Result
<pre> a1 = float r = float "첫째항 : " >> OLED >> a1 << "\n" "공비 : " >> OLED >> r << "\n" thr = a1 * 0.000001 sum = float if(r <= 0 r >= 1) OLED << "/r발산!" else while(1) sum += a1 a1 *= r if(a1 <= thr): break OLED << f"/g급수의 합= {sum:.4f}" </pre>	<pre> 첫째항 : 1 공비 : 0.5 급수의 합= 2.0000 </pre>
<pre> a1 = float r = float "첫째항 : " >> OLED >> a1 << "\n" "공비 : " >> OLED >> r << "\n" thr = a1 * 0.000001 sum = float if(r <= 0 r >= 1) OLED << "/r발산!" else while(1) sum += a1 a1 *= r if(a1 <= thr): break OLED << f"/g급수의 합= {sum:.4f}" </pre>	<pre> 첫째항 : 1 공비 : 0.333333 급수의 합= 1.5000 </pre>
<pre> a1 = float r = float "첫째항 : " >> OLED >> a1 << "\n" "공비 : " >> OLED >> r << "\n" thr = a1 * 0.000001 sum = float if(r <= 0 r >= 1) OLED << "/r발산!" else while(1) sum += a1 a1 *= r if(a1 <= thr): break OLED << f"/g급수의 합= {sum:.4f}" </pre>	<pre> 첫째항 : 2 공비 : 2 발산! </pre>

■ Starlic - 버튼 입력 예제(2가지)

Starlic으로 버튼 인식 코드는 아래와 같이 작성할 수 있다. 앞에서 작성한 코드에서 중괄호를 없애고 들여쓰기만 맞추면 된다. 첫 번째 코드는 버튼을 한 번만 입력하고 종료하는 코드이고, 두 번째 코드는 계속 입력되지만 누적해서 출력하는 코드이다.

Age.s14	Result
<pre>main() btn = int; while(!(btn = BUTTON_Read())) when(btn) case BUTTON_0: OLED << "/r0번 버튼 입력!"; case BUTTON_1: OLED << "/g1번 버튼 입력!"; case BUTTON_2: OLED << "/y2번 버튼 입력!";</pre>	<div>0번 버튼 입력!</div> <div>1번 버튼 입력!</div> <div>2번 버튼 입력!</div>
<pre>main() btn = int; while(1) btn = BUTTON_Read(); when(btn) case BUTTON_0: OLED << "/r0번 버튼 입력!"; case BUTTON_1: OLED << "/g1번 버튼 입력!"; case BUTTON_2: OLED << "/y2번 버튼 입력!";</pre>	<div>0번 버튼 입력!</div> <div>1번 버튼 입력!</div> <div>2번 버튼 입력!</div> <div>0번 버튼 입력!</div> <div>1번 버튼 입력!</div> <div>2번 버튼 입력!</div> <div>0번 버튼 입력!</div> <div>1번 버튼 입력!</div> <div>2번 버튼 입력!</div>

■ Simple - 버튼 입력 예제(2가지)

Simple style로 버튼 인식 코드는 아래와 같이 작성할 수 있다. 앞에서 작성한 코드에서 세미콜론만 없애면 된다. 첫 번째 코드는 버튼을 한 번만 입력하고 종료하는 코드이고, 두 번째 코드는 계속 입력되지만 누적해서 출력하는 코드이다.

Age.s14	Result
<pre>main btn = int while !(btn = BUTTON_Read()) when(btn) case BUTTON_0: OLED << "/r0번 버튼 입력!" case BUTTON_1: OLED << "/g1번 버튼 입력!" case BUTTON_2: OLED << "/y2번 버튼 입력!"</pre>	<div>0번 버튼 입력!</div> <div>1번 버튼 입력!</div> <div>2번 버튼 입력!</div>
<pre>main btn = int while(1) btn = BUTTON_Read() when(btn) case BUTTON_0: OLED << "/r0번 버튼 입력!" case BUTTON_1: OLED << "/g1번 버튼 입력!" case BUTTON_2: OLED << "/y2번 버튼 입력!"</pre>	<div>0번 버튼 입력! 1번 버튼 입력! 2번 버튼 입력!</div> <div>1번 버튼 입력! 2번 버튼 입력! 3번 버튼 입력!</div>

■ Pythonic - 버튼 입력 예제(2가지)

Pythonic style로 버튼 인식 코드는 아래와 같이 작성할 수 있다. 앞에서 작성한 코드에서 **main**, **while**, **when**에만 ``:``를 붙이면 된다. 첫 번째 코드는 버튼을 한 번만 입력하고 종료하는 코드이고, 두 번째 코드는 계속 입력되지만 누적해서 출력하는 코드이다. 다만, 아무것도 없는 **while**문은 **continue** 하나 정도 붙여주는 것이 가독성에 유리하다.

Age.s14	Result
<pre> main: btn = int while !(btn = BUTTON_Read()): continue when btn: case BUTTON_0: OLED << "/r0번 버튼 입력!" case BUTTON_1: OLED << "/g1번 버튼 입력!" case BUTTON_2: OLED << "/y2번 버튼 입력!" </pre>	<div>0번 버튼 입력!</div> <div>1번 버튼 입력!</div> <div>2번 버튼 입력!</div>
<pre> main btn = int; while 1 btn = BUTTON_Read(); when btn case BUTTON_0: OLED << "/r0번 버튼 입력!" case BUTTON_1: OLED << "/g1번 버튼 입력!" case BUTTON_2: OLED << "/y2번 버튼 입력!" </pre>	<div>0번 버튼 입력! 0번 버튼 입력! 0번 버튼 입력! 0번 버튼 입력!</div> <div>1번 버튼 입력! 1번 버튼 입력! 1번 버튼 입력! 1번 버튼 입력!</div> <div>2번 버튼 입력! 2번 버튼 입력! 2번 버튼 입력! 2번 버튼 입력!</div>

■ 정리

이번에는 스타일 변경에 따른 문법 변경이 미묘하게 달라서 아래와 같이 바뀌어야 하는 요소들 위주로 정리해 본다.

변경내용	Starlic	Simple	Pythonic	Super-Simple
중괄호	X	X	X	X
세미콜론	0	선택	선택	선택
소괄호 생략	가능	가능	가능	불가 (if/while안에서는 가능)
아무것도 없는 while 문	{ } 사용 권장 (의무아님)	아무것도 작성 안함	: 사용 continue 작성 권장(선택)	continue 작성 (필수)
case 뒤에 : 사용	권장	권장	관례상 필수	권장
사용	가능	가능	가능	가능

● | 사용법 이용하기

들여쓰기 후 한참 있다가 원래대로 돌아오는 경우 들여쓰기를 몇 칸 했는지 헛갈릴 수 있다. 이때, | 사용법을 적용하면 들여쓰기 위치를 쉽게 파악할 수 있어 유용하게 쓰일 수 있다.

● 아무것도 없는 **while**문

가독성 향상을 위해 들여쓰기를 쓰는 상황에서는 **continue**를 내부에 작성해 주는 것을 추천한다. 다만, 함수 밖에서 작성하는 **Super-Simple** 스타일에서는 의무적으로 작성해 주어야 오류가 나지 않는다. 자동으로 중괄호를 추가해 버리기 때문이다.

604 GUE에서 while문 사용하기

GUE에서 **while**문의 사용은 전통적으로 ``while` 조건`을 그대로 사용한다. 덕분에 이번에는 지금까지 변환했던 방식 그대로 적용해도 문제없다. 이번에는 2장에서 다룬 코드들을 **GUE Style**로 변환하고 실행해 보기로 하자.

■ Simple-Style을 그대로 실행하는 경우(등비급수)

매번 그랬듯이 `start>>` 뒤에 **simple-style**을 그대로 사용해도 작동에 오류가 없는 것이 일반적이었다. 이 과정을 거치는 이유는 **simple-style**에서 출발해 **GUE**로 변환하기 때문이다. 역시나 제대로 실행된다!

Age.s14	Result
<pre>start>> a1 = float r = float "첫째항 : " >> OLED >> a1 << "\n" "공비 : " >> OLED >> r << "\n" thr = a1 * 0.000001 sum = float if(r <= 0 r >= 1) OLED << "/r발산!" else while(1) sum += a1 a1 *= r if(a1 <= thr): break OLED << f"/g급수의 합= {sum:.4f}"</pre>	<pre>첫째항 : 1 공비 : 0.5 급수의 합= 2.0000</pre>
<pre>start>> a1 = float r = float "첫째항 : " >> OLED >> a1 << "\n" "공비 : " >> OLED >> r << "\n" thr = a1 * 0.000001 sum = float if(r <= 0 r >= 1) OLED << "/r발산!" else while(1) sum += a1 a1 *= r if(a1 <= thr): break OLED << f"/g급수의 합= {sum:.4f}"</pre>	<pre>첫째항 : 1 공비 : 0.333333 급수의 합= 1.5000</pre>
<pre>start>> a1 = float r = float "첫째항 : " >> OLED >> a1 << "\n" "공비 : " >> OLED >> r << "\n" thr = a1 * 0.000001 sum = float if(r <= 0 r >= 1) OLED << "/r발산!" else while(1) sum += a1 a1 *= r if(a1 <= thr): break OLED << f"/g급수의 합= {sum:.4f}"</pre>	<pre>첫째항 : 2 공비 : 2 발산!</pre>

■ GUE-Style로의 변환(등비급수)

이번에는 앞에서 다룬 코드를 GUE로 변환하는 과정을 살펴본다. 의외로 변환할 내용들이 많은데, 변수 대입은 _대입법을, if문의 괄호는 없애고, 한 줄 코드로 작성한 게 있으면 두 줄로 나눠준다.

변경 전	변경 후
<code>a1 = float</code> <code>"첫째항 : " >> OLED >> a1 << "\n"</code>	<code>_a1 FloatVal OLED "첫째항 : "</code>
<code>thr = a1 * 0.000001</code>	<code>_thr a1 * 0.000001</code>
<code>sum = float</code>	<code>_sum float</code>
<code>if(r <= 0 r >= 1)</code>	<code>if r <= 0 r >= 1</code>
<code>sum += a1</code>	<code>_sum sum + a1</code>
<code>if(a1 <= thr): break</code>	<code>if a1 <= thr</code> <code>break</code>
<code>OLED << f"/g급수의 합= {sum:.4f}"</code>	<code>Print OLED f"/g급수의 합= {sum:.4f}"</code>

start>>				
	<code>_a1</code>	<code>FloatVal</code>	<code>OLED</code>	<code>"첫째항 : "</code>
	<code>_r</code>	<code>FloatVal</code>	<code>OLED</code>	<code>"공비 : "</code>
	<code>_thr</code>	<code>a1 * 0.000001</code>		
	<code>_sum</code>	<code>float</code>		
	<code>if</code>	<code>r <= 0 r >= 1</code>		
		<code>Print</code>	<code>OLED</code>	<code>"/r발산!"</code>
	<code>else</code>			
		<code>Print</code>	<code>OLED</code>	<code>"/r20대 중반\n"</code>
		<code>while</code>	<code>1</code>	
			<code>_sum</code>	<code>sum + a1</code>
			<code>_a1</code>	<code>a1 * r</code>
			<code>if</code>	<code>a1 <= thr</code>
				<code>break</code>
		<code>Print</code>	<code>OLED</code>	<code>f"/g급수의 합= {sum:.4f}"</code>
<code>end</code>				

Age.s14	Result
<pre> start>> _a1 FloatVal OLED "첫째항 : " _r FloatVal OLED "공비 : " _thr a1 * 0.000001 _sum float if r <= 0 r >= 1 Print OLED "/r발산!" else while 1 _sum sum + a1 _a1 a1 * r if a1 <= thr break Print OLED f"/g급수의 합= {sum:.4f}" </pre>	<pre> 첫째항 : 1 공비 : 0.5 급수의 합= 2.0000 </pre>
<pre> start>> _a1 FloatVal OLED "첫째항 : " _r FloatVal OLED "공비 : " _thr a1 * 0.000001 _sum float if r <= 0 r >= 1 Print OLED "/r발산!" else while 1 _sum sum + a1 _a1 a1 * r if a1 <= thr break Print OLED f"/g급수의 합= {sum:.4f}" </pre>	<pre> 첫째항 : 1 공비 : 0.333333 급수의 합= 1.5000 </pre>
<pre> start>> _a1 FloatVal OLED "첫째항 : " _r FloatVal OLED "공비 : " _thr a1 * 0.000001 _sum float if r <= 0 r >= 1 Print OLED "/r발산!" else while 1 _sum sum + a1 _a1 a1 * r if a1 <= thr break Print OLED f"/g급수의 합= {sum:.4f}" </pre>	<pre> 첫째항 : 2 공비 : 2 발산! </pre>

■ Simple-Style을 그대로 실행하는 경우(버튼 입력)

매번 그랬듯이 `start>>` 뒤에 `simple-style`을 그대로 사용해도 작동에 오류가 없는 것이 일반적이었다. 이 과정을 거치는 이유는 `simple-style`에서 출발해 `GUE`로 변환하기 때문이다. 역시나 제대로 실행된다!

Age.s14	Result
<pre>start>> btn = int while !(btn = BUTTON_Read()) when(btn) case BUTTON_0: OLED << "/r0번 버튼 입력!" case BUTTON_1: OLED << "/g1번 버튼 입력!" case BUTTON_2: OLED << "/y2번 버튼 입력!"</pre>	<div>0번 버튼 입력!</div> <div>1번 버튼 입력!</div> <div>2번 버튼 입력!</div>
<pre>start>> btn = int while(1) btn = BUTTON_Read() when(btn) case BUTTON_0: OLED << "/r0번 버튼 입력!" case BUTTON_1: OLED << "/g1번 버튼 입력!" case BUTTON_2: OLED << "/y2번 버튼 입력!"</pre>	<div>0번 버튼 입력! 1번 버튼 입력! 2번 버튼 입력!</div> <div>0번 버튼 입력! 1번 버튼 입력! 2번 버튼 입력!</div> <div>0번 버튼 입력! 1번 버튼 입력! 2번 버튼 입력!</div>

■ GUE-Style로의 변환(버튼 입력1)

이번에는 앞에서 다룬 코드를 GUE로 변환하는 과정을 살펴본다. 의외로 변환할 내용들이 많은데, 변수 대입은 _대입법을, if문의 괄호는 없애고, 한 줄 코드로 작성한 게 있으면 두 줄로 나눠준다.

변경 전	변경 후
<code>btn = int</code>	<code>_btn int</code>
<code>when(btn)</code>	<code>when btn</code>
<code>case BUTTON_0:</code>	<code>case BUTTON_0</code>
<code>OLED << "/r0번 버튼 입력!"</code>	<code>Print OLED "/r0번 버튼 입력!"</code>

start>>					
	<code>_btn</code>	<code>int</code>			
	<code>while</code>	<code>!(btn = BUTTON_Read())</code>			
	<code>when</code>	<code>btn</code>			
		<code>case</code>	<code>BUTTON_0</code>		
			<code>Print</code>	<code>OLED</code>	<code>"/r0번 버튼 입력!"</code>
		<code>case</code>	<code>BUTTON_1</code>		
			<code>Print</code>	<code>OLED</code>	<code>"/g1번 버튼 입력!"</code>
		<code>case</code>	<code>BUTTON_2</code>		
			<code>Print</code>	<code>OLED</code>	<code>"/y2번 버튼 입력!"</code>
end					

Age.s14	Result
start>>	
<code>_btn int</code>	
<code>while !(btn = BUTTON_Read())</code>	
<code>when btn</code>	
<code>case BUTTON_0</code>	0번 버튼 입력!
<code>Print OLED "/r0번 버튼 입력!"</code>	
<code>case BUTTON_1</code>	1번 버튼 입력!
<code>Print OLED "/g1번 버튼 입력!"</code>	
<code>case BUTTON_2</code>	2번 버튼 입력!
<code>Print OLED "/y2번 버튼 입력!"</code>	

■ GUE-Style로의 변환(버튼 입력2)

이번에는 앞에서 다룬 코드를 GUE로 변환하는 과정을 살펴본다. 의외로 변환할 내용들이 많은데, 변수 대입은 _대입법을, if문의 괄호는 없애고, 한 줄 코드로 작성한 게 있으면 두 줄로 나눠준다.

변경 전	변경 후
<code>btn = int</code>	<code>_btn int</code>
<code>when(btn)</code>	<code>when btn</code>
<code>case BUTTON_0:</code>	<code>case BUTTON_0</code>
<code>OLED << "/r0번 버튼 입력!"</code>	<code>Print OLED "/r0번 버튼 입력!"</code>

start>>					
	<code>_btn</code>	<code>int</code>			
	<code>while</code>	<code>1</code>			
		<code>_btn</code>	<code>BUTTON_Read()</code>		
		<code>when</code>	<code>btn</code>		
			<code>case</code>	<code>BUTTON_0</code>	
			<code>Print</code>	<code>OLED</code>	<code>"/r0번 버튼 입력!"</code>
			<code>case</code>	<code>BUTTON_1</code>	
			<code>Print</code>	<code>OLED</code>	<code>"/g1번 버튼 입력!"</code>
			<code>case</code>	<code>BUTTON_2</code>	
			<code>Print</code>	<code>OLED</code>	<code>"/y2번 버튼 입력!"</code>
end					

Age.s14	Result
start>>	
<code>_btn int</code>	
<code>while 1</code>	
<code>_btn BUTTON_Read()</code>	0번 버튼 입력!
<code>when btn</code>	
<code>case BUTTON_0</code>	
<code>Print OLED "/r0번 버튼 입력!"</code>	1번 버튼 입력!
<code>case BUTTON_1</code>	
<code>Print OLED "/g1번 버튼 입력!"</code>	
<code>case BUTTON_2</code>	
<code>Print OLED "/y2번 버튼 입력!"</code>	2번 버튼 입력!

■ 정리

● GUE에서 while문 사용

- GUE에서의 while문 사용은 Simple과 비슷하게 사용한다.
- GUE-Style에 맞게 바꾸는 작업만 거치면 쉽게 사용할 수 있다.

● 아무것도 하지 않는 while문

- GUE에서는 아무것도 하지 않는 while문은 명령어 취급한다. 따라서 명령어 작성 하듯이 작성하면 되며, continue문을 안에 따로 넣을 필요는 없다.

● GUE 변환표

- 지금까지 다룬 내용을 바탕으로 아래와 같이 변환표를 정리해 보았다. 참고해 두자.

변경 전	변경 후
<code>a1 = float "첫째항 : " >> OLED >> a1 << "\n"</code>	<code>_a1 FloatVal OLED "첫째항 : "</code>
<code>thr = a1 * 0.000001</code>	<code>_thr a1 * 0.000001</code>
<code>sum = float</code>	<code>_sum float</code>
<code>if(r <= 0 r >= 1)</code>	<code>if r <= 0 r >= 1</code>
<code>sum += a1</code>	<code>_sum sum + a1</code>
<code>if(a1 <= thr): break</code>	<code>if a1 <= thr break</code>
<code>case BUTTON_0:</code>	<code>case BUTTON_0</code>
<code>OLED << f"/g급수의 합= {sum:.4f}"</code>	<code>Print OLED f"/g급수의 합= {sum:.4f}"</code>

605 반복문 for의 이해

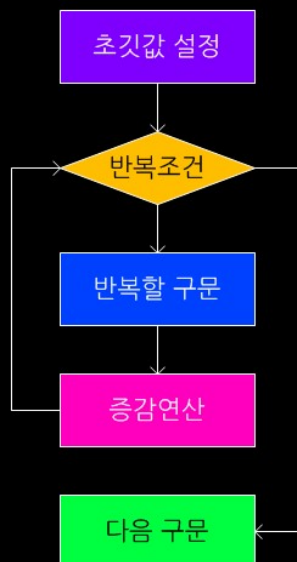
5~8장에서는 **for**문을 다뤄 보도록 하겠다. **for**문은 **while**문과 다르게 유한한 반복문에서 주로 사용하며, 반복변수를 사용해 반복 인덱스를 사용해볼 수 있다. 기본적으로 **for**문은 C언어에서의 사용법과 똑같이 쓸 수 있으며, C언어에는 없는 **for**문의 사용법도 같이 다루어 볼 예정이다.

■ for문의 요소

C언어에서 **for**문은 3가지 요소를 이용해 정의하였다. 흔히 **for**(i=0; i<20; i++)와 같은 형태로 **for**문을 사용했을 것이다. **Starlit**에서도 비슷하게 사용할 수 있다. 하지만 C언어에서와 다르게 ++ 연산자는 따로 없으므로 i+=1의 형태로 쓰기로 한다.

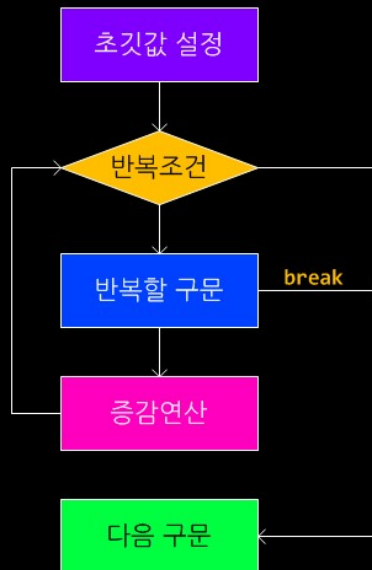
```
for(초깃값; 조건; 증감연산){  
    반복 구문  
}
```

위와 같은 형태로 작성했을 때 **for**문은 아래와 같이 작동한다. 우선 초깃값을 설정하고 반복 조건에 따라 반복 구문을 수행하고 증감 연산 후 반복 조건이 참이 되면 반복문을 계속 실행하고 거짓이 되면 다음 구문을 실행한다.



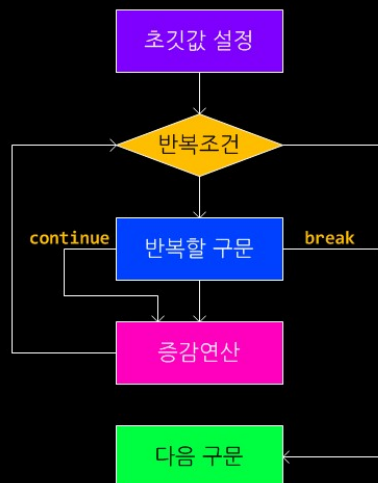
■ for문을 빠져나가는 break

for문 역시 **break**를 통해 빠져나갈 수 있다. **break**를 통해 빠져나간 경우 증감 연산을 건너뛰고 다음 구문으로 넘어간다.



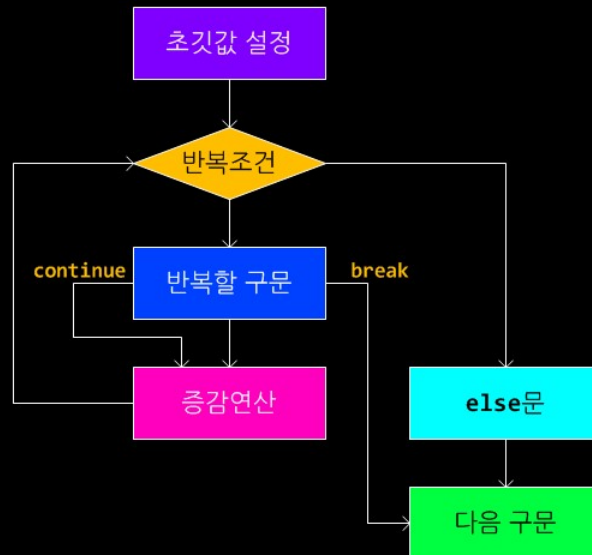
■ for문을 계속 실행하는 continue

for문을 실행하던 도중 **continue**를 통해 반복문을 계속 실행할 수 있다. 이 경우 증감연산을 한 번 거치고 반복조건으로 넘어간다.



■ for문과 else문

for문에서도 else와 같이 사용할 수 있다. 이것 역시 while문과 비슷하게 break로 나갔을 때는 else가 실행되지 않고 break 없이 반복조건이 맞지 않았을 때 실행되는 구문이다.



■ elif문

elwhile과 비슷한 맥락으로 elif문을 사용할 수 있다. for문에서 else를 사용해야 하는 상황에서 for를 쓸 때나 앞의 조건문에서 else 뒤에 for를 쓸 때 사용한다.

■ 정리

● for문의 정의

- for(초기설정;반복조건;증감연산)의 형태로 for문을 사용한다.
- 초기설정 : for문을 시작하기 앞서 반복 변수를 정의 또는 시작값 대입에 사용한다.
- 반복조건 : 조건이 참일 때만 반복한다. while의 인수와 역할이 같다.
- 증감연산 : 반복변수의 값을 증가 또는 감소시킨다.

● break와 continue

- break는 반복문을 탈출할 때 사용한다.
- continue는 반복문을 앞에서부터 계속 반복할 때 사용한다.

● else와 elif

- 반복문에서 else는 break로 중간에 탈출한 것이 아닌 반복조건을 만족하지 못해 탈출했을 때 실행한다.
- elif은 앞의 반복문에서 else가 실행되거나 앞의 조건문이 실행되지 않았을 때 실행하는 반복문이다. 하지만, 잘 사용되지 않으니 참고용으로 알아두자.

606 Full-Style에서 for문 사용하기

이번 장에서는 **for**문을 사용한 예제를 하나 들어 보겠다. 사용자가 값을 입력하면 1부터 그 값까지 출력하는 예제를 들어 보겠다.

■ for문 예제 - 1부터 n까지 출력하는 예제

사용자로부터 1 이상의 정수를 입력해서 OLED에 1부터 그 수까지 빨간색과 노란색을 번갈아 가면서 출력하는 예제이다. 반복변수 *i*를 사용해서 *i*가 2로 나누어떨어지면 노란색을, 그렇지 않으면 빨간색으로 출력하게 설계했다.

Age.s14	Result
<pre>main(){ n = int; "정수 입력 : " >> OLED >> n << "\n"; for(i=1; i<=n; i+=1){ if(i%2){ OLED << f"/r{i}"; } else{ OLED << f"/y{i}"; } } }</pre>	<pre>정수 입력 : 20 123456789101112 131415161718192 0</pre>

■ for...else 사용하기 - 버튼 눌러서 숫자 출력하기

사용자로부터 버튼을 입력받아 숫자를 출력하는 예제이다. 버튼을 배열로 선언해서 그 버튼값과 맞으면 숫자를 출력하고, 숫자 버튼이 눌리지 않으면 프로그램을 종료하는 방식이다. 아래와 같이 작성해볼 수 있다.

- 배열의 선언 : `arr = [a1, a2, ...]`의 형태로 선언

Starlit에서는 배열을 한 번 선언하면 크기가 고정되기 때문에 선언할 때 크기를 잘 생각해야 한다.

- 배열의 사용 : `arr[0]`, `arr[1]`, `arr[2]`와 같이 사용할 수 있다.

- 배열은 전역변수에 정의하고 사용할 수 있고, 지역변수에 정의하고 사용할 수 있으나, 이번 프로그램에서는 지역변수에서 정의한다.

Age.s14	Result
<pre>main(){ barr = [BUTTON_0, BUTTON_1, BUTTON_2, BUTTON_3, BUTTON_4, BUTTON_5, BUTTON_6, BUTTON_7, BUTTON_8, BUTTON_9]; while(1){ btn = BUTTON_Read(); for(i=0; i<10; i+=1){ if(btn == barr[i]){ OLED << f"/r{i}"; break; } } elif(btn){ break; } } }</pre>	201023

※ 실행 결과는 2->0->1->0->2->3 순으로 입력했을 때의 결과이다.

이 프로그램에서 **break**문이 2개 있는데, 처음 나오는 **break**문은 **for**문을 탈출하는 것이고, 두 번째 나오는 **break**문은 **while**문을 탈출하는 것이다. 만약 반복문 2개 이상 탈출하고 싶다면 **break 3**으로 작성하면 된다. 이것은 중괄호 3개를 빠져나가겠다는 의미이다.

■ for문의 압축

for문은 압축해서 사용할 수 있다. *i*가 0부터 9까지라면 간단하게 *i:0:9*라고 적으면 되기 때문이다. 처음값과 끝값 모두 포함한다.

Age.s14	Result
<pre>main(){ barr = [BUTTON_0, BUTTON_1, BUTTON_2, BUTTON_3, BUTTON_4, BUTTON_5, BUTTON_6, BUTTON_7, BUTTON_8, BUTTON_9]; while(1){ btn = BUTTON_Read(); for(i:0:9){ if(btn == barr[i]){ OLED << f"/r{i}"; break; } } elif(btn){ break; } } }</pre>	201023

※ 실행 결과는 2->0->1->0->2->3 순으로 입력했을 때의 결과이다.

■ for문의 직관적 표현

for문은 아래와 같이 **from**과 **to**를 사용해 작성해도 좋다. 역시 처음 값과 끝값은 모두 포함한다.

Age.s14	Result
<pre>main(){ barr = [BUTTON_0, BUTTON_1, BUTTON_2, BUTTON_3, BUTTON_4, BUTTON_5, BUTTON_6, BUTTON_7, BUTTON_8, BUTTON_9]; while(1){ btn = BUTTON_Read(); for i from 0 to 9{ if(btn == barr[i]){ OLED << f"/r{i}"; break; } } elif(btn){ break; } } }</pre>	201023

※ 실행 결과는 2->0->1->0->2->3 순으로 입력했을 때의 결과이다.

■ 정리

Full-Style에서 for문을 사용하여 순차적이고 유한한 반복 예제를 살펴 보았다.
여기서 배운 코드들을 정리해 보면 아래와 같다.

● for문의 작성법 - 0부터 9까지를 표현하기

- `for(i=0; i<=9; i+=1)`
- `for(i:0:9)`
- `for i from 0 to 9`

● 배열의 정의와 사용법

- 배열 선언 및 초기화 : `a = [0, 1, 2, 3, 4];`
- 배열 선언만 하는 경우 : `a = int[20];`

● for문과 else의 사용

- for문에서 break문으로 탈출 시 : else문 실행 안함.
- for문에서 반복조건에 맞지 않아 탈출 시 : else문 실행.

607 Starlic, Simple, Pythonic에서 for문 사용하기

이번에는 6장에서 다룬 두 가지 코드에 대해서 **Starlic**, **Simple**, **Pythonic** 스타일에서 작성하는 방법을 알아보도록 하겠다.

■ Starlic-Style - 1부터 n까지 출력하는 예제

앞에서 다룬 1부터 n까지 색을 바꿔 가면서 출력하는 예제를 Starlic으로 작성하겠다. 여기서도 중괄호만 없애면 된다.

Age.s14	Result
<pre>main() n = int; "정수 입력 : " >> OLED >> n << "\n"; for(i=1; i<=n; i+=1) if(i%2) OLED << f"/r{i}"; else OLED << f"/y{i}";</pre>	<pre>정수 입력 : 20 123456789101112 131415161718192 0</pre>

■ Simple-Style - 1부터 n까지 출력하는 예제

역시 지금까지 그래왔듯 세미콜론과 불필요한 괄호를 없애면 된다. 아래와 같이 작성하면 된다.

Age.s14	Result
<pre>main n = int "정수 입력 : " >> OLED >> n << "\n" for(i=1; i<=n; i+=1) if(i%2) OLED << f"/r{i}" else OLED << f"/y{i}"</pre>	<pre>정수 입력 : 20 123456789101112 131415161718192 0</pre>

■ Pythonic-Style - 1부터 n까지 출력하는 예제

이번에는 Pythonic-Style에서 1부터 n까지 출력하는 예제이다. **for**, **if**, **else**, **main**에 `:`를 붙이자. Pythonic-Style이라고 해서 **for**문을 Python 스타일로 작성하라는 것은 아니므로 `for i in range(0,9)`와 같은 형태로 작성하지 않도록 주의하자.

Age.s14	Result
<pre> main: n = int "정수 입력 : " >> OLED >> n << "\n" for i=1, i<=n, i+=1: if i%2: OLED << f"/r{i}" else: OLED << f"/y{i}" </pre>	<pre> 정수 입력 : 20 123456789101112 131415161718192 0 </pre>

Pythonic을 작성할 때 **for**문에 괄호를 없앤다면 주의해야 한다. 세미콜론으로 **for**문의 구분을 진행하고 컴파일하다보면 코드가 분리되어 컴파일 오류가 날 수 있기 때문이다. 따라서 괄호를 없앨 거면 세미콜론 대신 반점을 사용하도록 주의해야 한다.

■ Super-Simple-Style - 1부터 n까지 출력하는 예제

이번에는 **main** 함수를 없애고 작성해 보겠다. **main**함수를 없앤다면 소괄호를 없앨 수 없다는 점 꼭 주의해야 한다.

Age.s14	Result
<pre> n = int "정수 입력 : " >> OLED >> n << "\n" for(i=1; i<=n; i+=1) if(i%2) OLED << f"/r{i}" else OLED << f"/y{i}" </pre>	<pre> 정수 입력 : 20 123456789101112 131415161718192 0 </pre>

■ Starlic-Style - 버튼 입력 예제

이번에는 **for...else**를 사용해서 버튼 입력 예제를 **Starlic-Style**에서 작성해 보겠다. **for a from b to c** 형태와 **for(a:0:9)** 형태 모두 사용한다.

Age.s14	Result
<pre> main() barr = [BUTTON_0, BUTTON_1, BUTTON_2, BUTTON_3, BUTTON_4, BUTTON_5, BUTTON_6, BUTTON_7, BUTTON_8, BUTTON_9]; while(1) btn = BUTTON_Read(); for(i:0:9) if(btn == barr[i]) OLED << f"/r{i}"; break; elif(btn) break; </pre>	201023
<pre> main() barr = [BUTTON_0, BUTTON_1, BUTTON_2, BUTTON_3, BUTTON_4, BUTTON_5, BUTTON_6, BUTTON_7, BUTTON_8, BUTTON_9]; while(1) btn = BUTTON_Read(); for i from 0 to 9 if(btn == barr[i]) OLED << f"/r{i}"; break; elif(btn) break; </pre>	

※ 실행 결과는 2->0->1->0->2->3 순으로 입력했을 때의 결과이다.

이렇게 작성할 때 주의점을 하나 말하자면, 배열 생성할 때 들여쓰기 규칙을 실수할 위험이 있다는 점이다. 줄 바꿈이 필요하다면 그다음 줄은 들여쓰기 규칙을 준수해야 컴파일 오류가 나지 않는다.

■ Simple-Style - 버튼 입력 예제

Starlic-Style에서 세미콜론을 없애고 불필요한 괄호를 없애면 Simple-Style에서도 작성할 수 있다.

Age.s14	Result
<pre> main barr = [BUTTON_0, BUTTON_1, BUTTON_2, BUTTON_3, BUTTON_4, BUTTON_5, BUTTON_6, BUTTON_7, BUTTON_8, BUTTON_9] while(1) btn = BUTTON_Read() for(i:0:9) if(btn == barr[i]) OLED << f"/g{i}" break elif(btn) break </pre>	201023
<pre> main barr = [BUTTON_0, BUTTON_1, BUTTON_2, BUTTON_3, BUTTON_4, BUTTON_5, BUTTON_6, BUTTON_7, BUTTON_8, BUTTON_9] while(1) btn = BUTTON_Read() for i from 0 to 9 if(btn == barr[i]) OLED << f"/g{i}" break elif(btn) break </pre>	

※ 실행 결과는 2->0->1->0->2->3 순으로 입력했을 때의 결과이다.

■ Pythonic-Style - 버튼 입력 예제

Pythonic 역시 Simple-Style에서 `main`, `while`, `for`, `if`와 `elif` 뒤에 ``:``를 추가하여 작성하면 된다.

Age.s14	Result
<pre> main: barr = [BUTTON_0, BUTTON_1, BUTTON_2, BUTTON_3, BUTTON_4, BUTTON_5, BUTTON_6, BUTTON_7, BUTTON_8, BUTTON_9] while 1: btn = BUTTON_Read() for i:0:9: if btn == barr[i]: OLED << f"/y{i}" break elif btn: break </pre>	201023
<pre> main: barr = [BUTTON_0, BUTTON_1, BUTTON_2, BUTTON_3, BUTTON_4, BUTTON_5, BUTTON_6, BUTTON_7, BUTTON_8, BUTTON_9] while 1: btn = BUTTON_Read() for i from 0 to 9: if btn == barr[i]: OLED << f"/y{i}" break elif btn: break </pre>	

※ 실행 결과는 2->0->1->0->2->3 순으로 입력했을 때의 결과이다.

■ Super-Simple-Style - 버튼 입력 예제

Super-Simple-Style에서는 Simple-Style에서 `main`함수를 없애면 된다. 하지만, `for a from b to c` 형태는 사용할 수 없으니 주의하자.

Age.s14	Result
<pre> barr = [BUTTON_0, BUTTON_1, BUTTON_2, BUTTON_3, BUTTON_4, BUTTON_5, BUTTON_6, BUTTON_7, BUTTON_8, BUTTON_9] while(1) btn = BUTTON_Read() for(i:0:9) if(btn == barr[i]) OLED << f"/g{i}" break elif(btn) break </pre>	201023

※ 실행 결과는 2->0->1->0->2->3 순으로 입력했을 때의 결과이다.

■ 정리

지금까지 한 내용을 바탕으로 Full-Style, Starlic, Simple, Pythonic, Super-Simple Style에서 For문의 사용 방법을 정리하면 다음과 같다.

Style	For	설명
Full	<code>for(i=0;i<=9;i+=1){}</code>	기본적인 형태의 for문
	<code>for(i:0:9){}</code>	0부터 9까지 증가하는 for문
	<code>for(i:9:0){}</code>	9에서 0까지 감소하는 for문
	<code>for i from 0 to 9 {}</code>	0에서 9까지 증가하는 for문 (감소로 사용 불가)
Starlic /Simple	<code>for(i=0;i<=9;i+=1)</code>	기본적인 형태의 for문
	<code>for(i:0:9)</code>	0부터 9까지 증가하는 for문
	<code>for(i:9:0)</code>	9에서 0까지 감소하는 for문
	<code>for i from 0 to 9</code>	0에서 9까지 증가하는 for문 (감소로 사용 불가)
Pythonic	<code>for i=0,i<=9,i+=1:</code>	기본적인 형태의 for문 세미콜론 사용 불가, 반점 사용
	<code>for i:0:9:</code>	0부터 9까지 증가하는 for문
	<code>for i:9:0:</code>	9에서 0까지 감소하는 for문
	<code>for i from 0 to 9:</code>	0에서 9까지 증가하는 for문 (감소로 사용 불가)
Super- Simple	<code>for(i=0;i<=9;i+=1)</code>	기본적인 형태의 for문 괄호 필수 사용
	<code>for(i:0:9)</code>	0부터 9까지 증가하는 for문 괄호 필수 사용
	<code>for(i:9:0)</code>	9에서 0까지 감소하는 for문 괄호 필수 사용
	<code>for i from 0 to 9</code>	사용 불가