



러플(RUR-PLE)

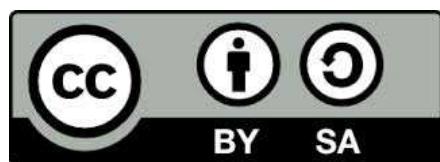
Learning Python: Child's Play with RUR-PLE!

Version 1.0 ('14.08)

저자: Andre Roberge

번역: 이광춘, 한정수 (xwmooc)

감수: 이은정



한국어로 번역하며

첫 인터넷 웹 브라우저를 만든 마크 앤더슨은 소프트웨어가 세상을 먹고 있다("Software is eating the world")는 자극적인 표현으로 2011년 월스트리트 저널에 기고를 했고, 카네기 멜론 대학의 샘 넬링 교수는 이론적 사고(Theoretical Thinking), 실험적 사고(Experimental Thinking)와 더불어 정보적 사고(Computational Thinking)가 현재도 그렇지만 앞으로 인간의 사고를 지배하는 중추적인 역할을 할 것이라고 주장했다. 특히 전 세계적으로 정보적 사고를 학습하고 소프트웨어를 이해하고 활용하는 사람과 그렇지 못한 사람과의 차이는 산업경제의 빈부격차보다 더 큰 디지털 경제의 정보 불평등(Digital Divide)를 야기할 것으로 예측했다.

우리나라의 산업발달과정을 보면 1950~1960년대 수입대체를 목표로 신발, 섬유가 중심이 된 경공업시대, 1970~1980년대 수출진흥을 목표로 철강, 조선, 자동차가 중심이 된 중공업시대, 1990~2000년대 기술혁신을 통한 반도체, 핸드폰, 디스플레이가 중심이 된 ICT 산업을 지나 2010~2020년대는 소프트웨어, 콘텐트, 과학기술이 중심이 되는 융합•지식 창조산업 시대로 발전해 갈 것으로 기대하고 있다.

이에 정부는 '14년 7월 국내외 경제, 사회 환경이 소프트웨어 중심사회로 급격히 변화하고 있으며, 소프트웨어가 혁신과 성장, 가치 창출의 중심이 되고, 개인•기업•국가의 경쟁력을 좌우하는 중요한 역할을 하고 있음에도 불구하고, 우리나라에는 법정부적, 국민적 관심이 미흡한 상황이라고 진단하고, 미국, 영국, 이스라엘 등 선진국과 마찬가지로, 초•중•고에서 소프트웨어를 필수로 이수할 수 있는 방안을 강구하고 있다.

유소년기 초기에 소프트웨어를 쉽고 재밌게 배우기 위해서 파이썬을 기반으로 하는 러플을 해외에서 많이 활용하고 있지만, 국내에서는 단편적으로 부분적으로 활용되고 있어 저자와 감수자가 원 저작자(Andre Roberge)와 협의하여 한국어 번역을 최초로 시도하여 많이 분들이 영어에 대한 부담없이 소프트웨어를 부담없이 재미있게 배울 수 있도록 결실을 맺었습니다. 마지막으로 러플(RUR-PLE) 교육자료를 자유롭게 이용할 수 있도록 허락해 주신 이천세무고등학교 김윤영 선생님께도 감사의 말씀을 전합니다.

러플 관련된 자료는 공식 홈페이지(<http://rur-ple.sourceforge.net/>) 및 xwmooc의 한국어로 번역사이트(<http://rur-ple.xwmooc.net/>)에서 최신의 콘텐츠를 접할 수 있습니다.

들어가며

컴퓨터 프로그램을 배우는 것은 남녀노소할 것 없이 재미있어야 합니다. 러플(RUR-PLE)은 파이썬 언어를 사용하여 컴퓨터 프로그램을 쉽게 배울 수 있게 특별히 개발된 환경입니다.

로봇이 다양한 작업을 수행하도록 프로그램된 인공의 환경하에서, 파이썬 구문(Python syntax)를 통해서 프로그램을 작성한다는 진정한 의미를 깨닫게 될 것입니다. 또한 내장된 인터프리터를 통해서 학습한 프로그래밍 기술을 좀더 전통적인 환경에서도 적용할 수 있습니다. 지금 당장 이런 말들이 여러분에게 큰 의미를 주지는 않지만, 걱정하지 마세요.

프로그램을 배우는 것은 재미있을 수 있지만, 여러분도 약간의 작업이 될 수 있습니다. 저는 여러분에게 가이드로서 여러분이 프로그램을 배울 수 있도록 도와드릴 수 있지만, 자신이 직접 프로그램을 작성하지 않는다면 여러분은 프로그램을 영영 배우지 못할 것입니다. 다음의 규칙을 따르는 것은 중요합니다.

규칙 1.

컴퓨터 프로그램을 배우는 것은 마치 음악 악기를 배우는 것과 같습니다: 여러분이 직접 프로그램을 해봐야 합니다. 단지 읽는 것만으로는 부족합니다.

당신이 알아야 할 두 번째 - 좋은 컴퓨터 프로그램을 만들기 위한 가장 중요한 일급비밀입니다.

규칙 2.

다른 사람이 읽기 쉬운 프로그램을 작성하세요.

자신이 직접 컴퓨터 프로그램을 작성하는데, 다른 분들이 자신이 직접 작성한 프로그램처럼 이해하기 쉽게 작성하세요. 사람의 언어가 사람들간에 커뮤니케이션이 되도록 진화하였듯이, 컴퓨터 언어는 여러분이 컴퓨터와 커뮤니케이션 하도록 설계되었습니다. 하지만, 컴퓨터 언어는 사람의 언어보다 무척이나 간단하지만, 다른 프로그래머와 프로그램을 공유하기 위해서 사용되기도 합니다. 마치 잘 쓰여진 국어 소설을 읽어서 국어 작문 실력을 향상하듯이, 잘 쓰여진 컴퓨터 프로그램을 읽음으로써 프로그래밍 기술을 향상시킬 수 있습니다. 하지만, 프로그램을 어떻게 하는지를 배우기 위해서는 자신이 직접 프로그램을 작성해야 합니다.

컴퓨터 프로그램을 작성하는 것을 시작하기 위해서, 리보그(Reeborg)로 불리는 로봇을 만들고, 컴퓨터 화면에 작업을 수행할 것입니다. 그러면서 자연스럽게 보편적인

프로그래밍 개념과 파이썬 언어에 대해서 배울 것입니다. 후에는, 리봇이 작업을 수행하는 것에서 벗어나, 컴퓨터를 사용하여 다른 것들을 배울 것입니다.

Contents

한국어로 번역하며	i
들어가며	ii
Contents	v
제 I 부	1
러플(RUR-PLE)소개 및 설치	1
러플(RUR-PLE) 소개 및 설치	2
제 II 부	8
학습 목록	8
브라우저(About the Browser) 소개	9
월드를 탐색하는 리보그	10
첫 번째 프로그램	12
오류(eRRoRs) 다루기	14
주석(Comments)	15
왼쪽으로 돌기	17
비퍼(Beepers)	20
 이놈의 버그!	22
벽 만들기	25
확실히 반복 피하기	29
다시 반복 피하기	38
리보그 스스로 결정을 할 수 있다면	43
제 말을 들으세요... 그렇지 않다면	46
if, else, if, else,	52
참이 아닙니까?	54
While 문	57
놀라운 Part 1	60
놀라운 Part 2	62
놀라운 Part 3	64
놀라운 Part 4	65
놀라운 Part 5	66
비가 와요	70
폭풍이 지나간 후에	73
정렬하기	75
반복 피하기 – 가져오는 것	78
리보그에게 덧셈 가르치기	83
파이썬은 이미 더하기를 알고 있어요	89
제 III 부	94
연습문제	94
문제 1	95
문제 2	96
문제 3	97
문제 4	98

문제 5	99
문제 6	100
문제 7	101
문제 8	102
문제 9	103
제 IV 부.....	104
러플(RUR-PLE) 사용법	104
프로그램 및 화면 구성.....	105
기본 명령	107
월드 꾸미기	114

제 I 부

러플(RUR-PLE)소개 및 설치

러플(RUR-PLE) 소개 및 설치

(1) 러플(RUR-PLE)은?

러플(RUR-PLE)(RUR-Python Learning Environment)는 André Roberge가 개발한 파이썬 언어를 이용한 프로그래밍 학습 환경입니다.

(2) 러플(RUR-PLE)의 특징

1) 로봇을 이용한 프로그래밍 학습

프로그래밍을 처음 접하는 학생들에게 추상적인 수치를 다루는 내용보다는 간단한 명령의 조작을 통해 눈에 보이는 로봇을 조작함으로써 프로그래밍을 보다 쉽게 이해 할 수 있도록 합니다.

2) 파이썬 언어 사용

문법은 간단하지만 강력한 기능을 지원하고 다양한 분야에 이용되는 파이썬 언어로 로봇을 제어합니다.

3) 튜토리얼 제공

혼자서 학습할 수 있는 튜토리얼(영문)을 제공합니다.

4) 파이썬 쉘과 에디터 제공

파이썬을 설치하지 않아도 기본적으로 파이썬 쉘(2.5.4 버전)과 에디터를 제공합니다.

5) 다양한 운영체제에서 지원

Windows, OSX, Linux에서 실행할 수 있는 배포판을 제공합니다.

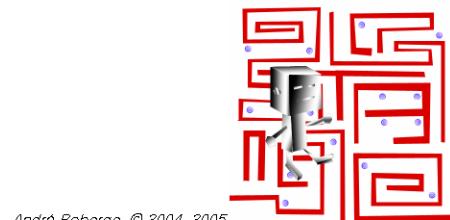
(3) 설치하기

1) 러플(RUR-PLE) 설치파일 다운로드

① <http://rur-ple.sourceforge.net> 에 접속하고 'go to the download page'를 클릭 합니다.

Learning Python: Child's Play with RUR-PLE!

Apprendre le langage Python avec RUR-PLE: un jeu d'enfants!



André Robarge, © 2004, 2005

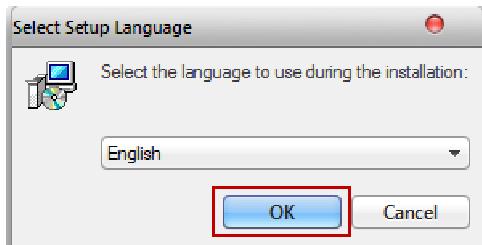
Learning to program computer should be fun, for adults and children alike. RUR-PLE is an environment designed to help you learn Python. To use RUR-PLE, you need [wxPython](#). You can [learn more about RUR-PLE](#) or you can [go to the download page](#).

- ② 'Download windows-setup-purple1.Orc3.exe (5. 1 MB)'를 클릭하여 파일을 다운로드 받습니다.

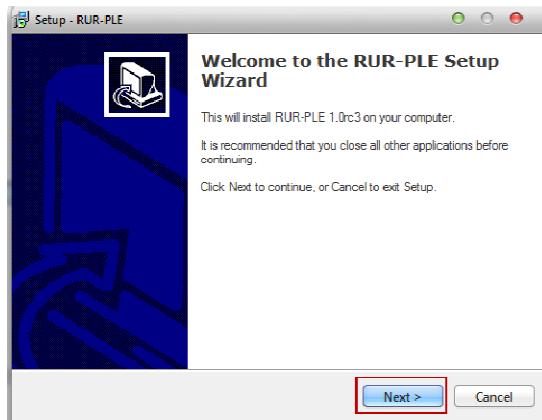


2) 러플(RUR-PLE) 설치

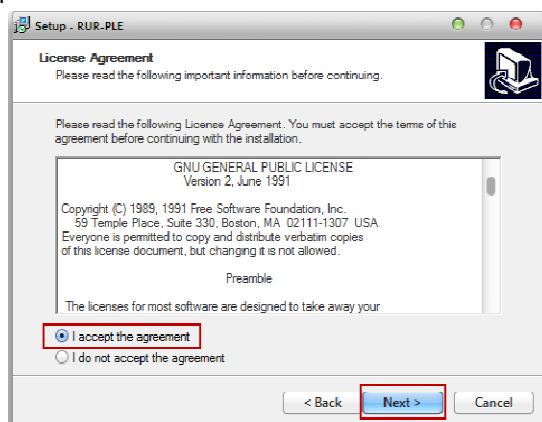
- ① 다운 받은 파일을 실행하고, 설치언어는 영어(English)를 선택합니다.



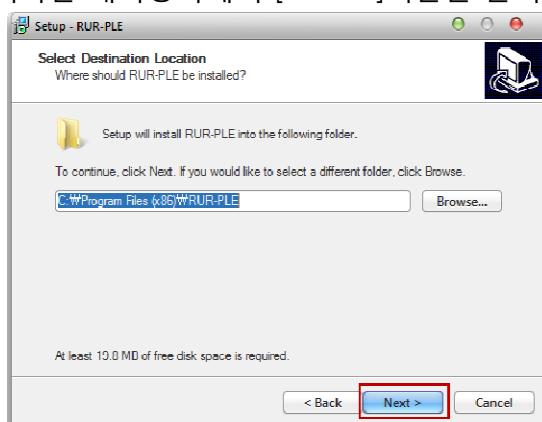
② 리플 설치화면이 표시되면 [Next] 버튼을 클릭합니다.



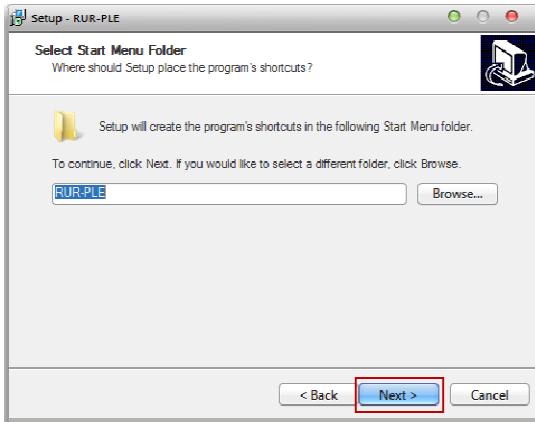
③ 설치를 동의하는 대화상자에서 'I accept the agreement'를 클릭하고, [Next >] 버튼을 클릭합니다.



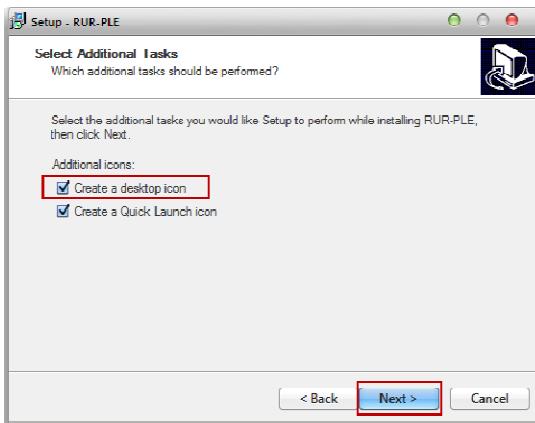
④ 설치폴더를 선택하는 대화상자에서 [Next >]버튼을 클릭합니다.



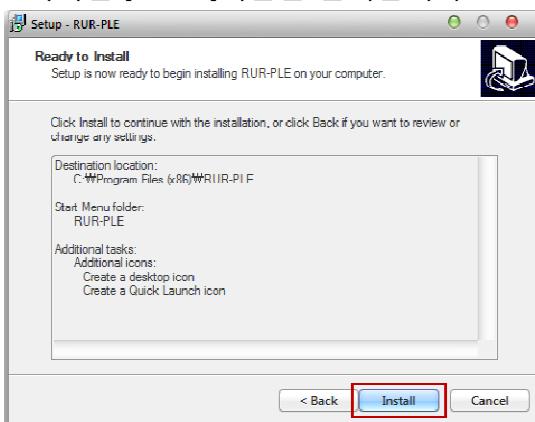
⑤ 시작메뉴폴더를 선택하는 대화상자에서 [Next >]버튼을 클릭합니다.



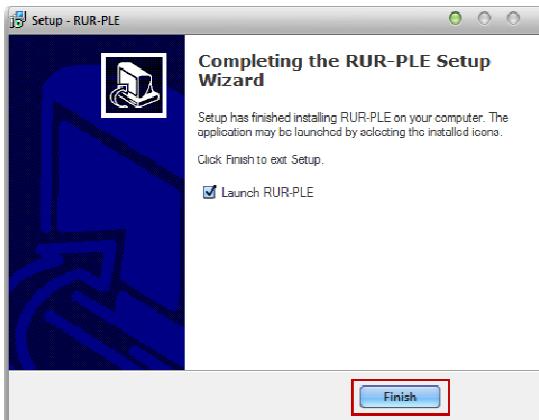
⑥ 바탕화면에 아이콘을 추가여부를 선택하는 대화상자에서 'Create a desktop icon'를 체크하고 'Create a Quick Lanch icon'를 체크합니다. [Next >]버튼을 클릭합니다.



⑦ 설치한 내용이 표시되면 [Install]버튼을 클릭합니다.

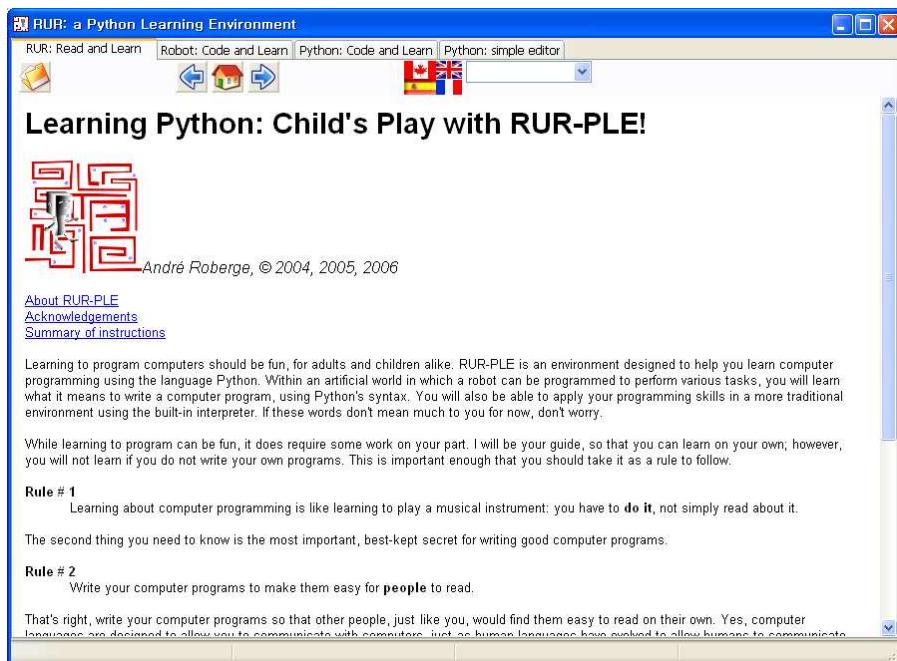


⑧ 설치가 완료되었다는 메시지가 표시되면 [Finish]버튼을 클릭합니다.



(4) 프로그램 설치 후 주요 화면 구성

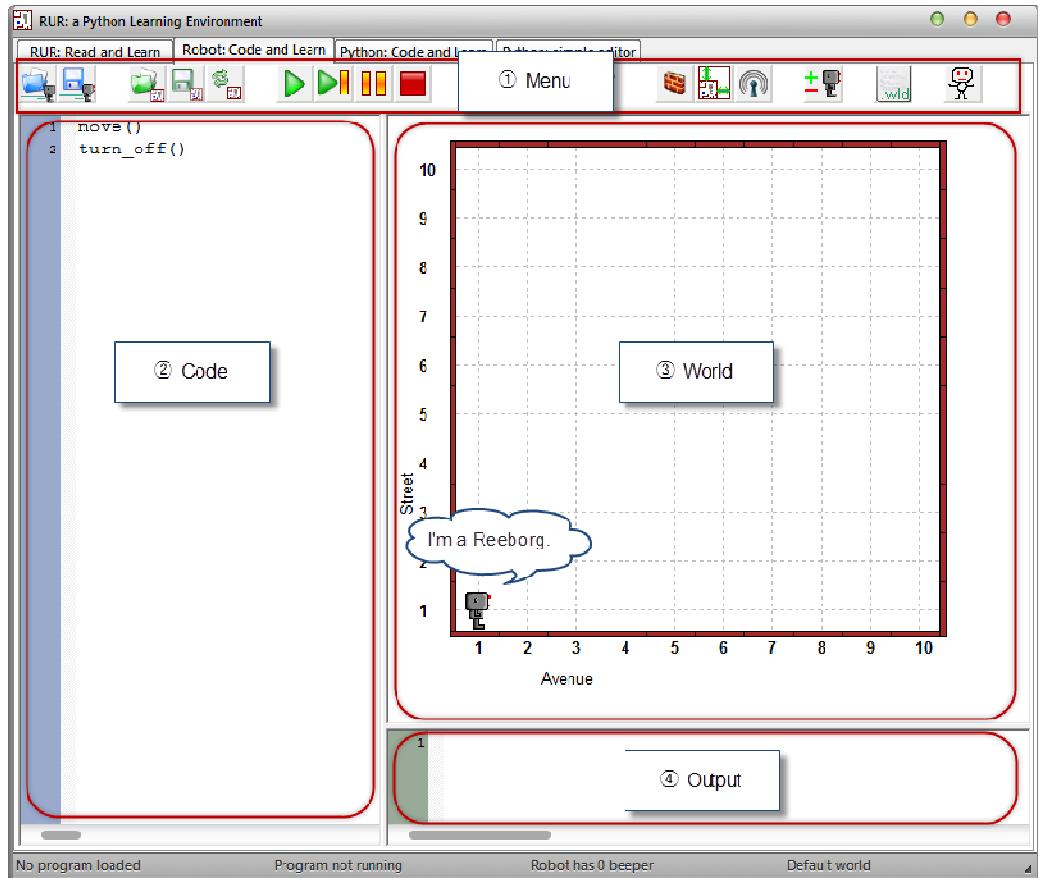
러플(RUR-PLE) 설치 후 4개의 탭으로 구분되고 각 탭 내부에 주요 화면이 구성되어 있습니다.



본 학습에서는 4가지 탭에서 2번째 탭(로봇을 이용한 프로그래밍[Robot: Code and Learn])에 중점을 둡니다.

(5) [Robot: Code and Learn] 탭의 화면구성

1) 화면구성



- ① Menu : 코드파일 및 월드파일 열기, 저장, 실행 등의 메뉴버튼
- ② Code : 리보그가 동작할 명령을 입력하는 부분
- ③ World : 리보그가 움직일 가상의 공간
- ④ Output: 데이터를 출력할 수 있는 부분

2) 메뉴구성



제 II 부

학습 목록

브라우저(About the Browser) 소개

러플(RUR-PLE)은 노트북으로 구성되어 있습니다. 첫 페이지(RUR : Read and Learn)는 지금 러플(RUR-PLE) 웹페이지와 동일한 페이지입니다. 이 페이지는 간단한 웹브라우저입니다. 사용하기는 간단하지만, 기능은 다소 제약이 있습니다.

이 브라우저는 여러분이 정보를 탐색하는데 도움이 되도록 다음 5 개 버튼을 제공합니다. 유용한 기준으로 위에서부터 순서가 되고 역할은 다음과 같습니다.

-  : 항상 처음 시작 페이지로 돌아갈 수 있게 합니다.
-  : 앞에서 본 페이지가 있다면, 뒷쪽 페이지로 갈 수 있게 합니다.
-  : 앞쪽 페이지로 볼 수 있게 이동합니다. (뒤로 돌아가는 버튼 동작의 반대)
-  : 브라우저 창에서 로컬파일(하드디스크, USB 등)에 저장된 것을 열게 합니다.
-  : 웹 정보 (URL, 예로 <http://www.python.org>) 열어서 브라우저에 정보를 보여줍니다.
-  러플(RUR-PLE) 번역 언어를 선택합니다. 현재 영어와 불어만 제공되고, 불어 제공은 현재까지 완벽하지는 않습니다.

현재 페이지 하단에 두 개의 링크가 있습니다. 준비가 되었으면, “리보고 명령하기” 링크를 클릭하여 다음 학습을 진행하세요.

--  -“리보고 명령하기” 

월드를 탐색하는 리보그

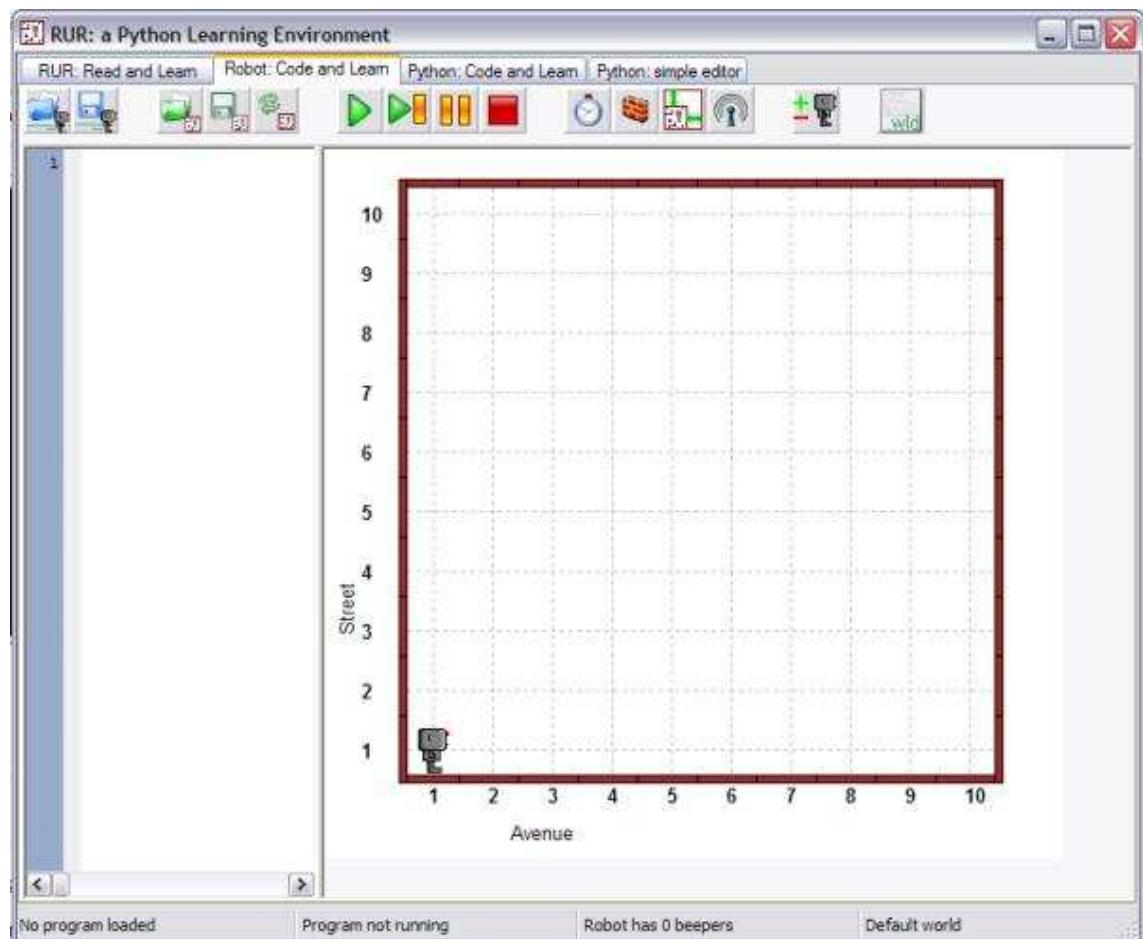
리보그는 4 가지 기본 작업을 할 수 있습니다. (앞으로 한칸 이동, 왼쪽 돌기, 비퍼를 줍기, 비퍼를 내려 놓기) 프로그래밍이라는 마술을 통해서 리보그가 4 가지 기본 작업을 조합해서 매우 복잡하고 어려운 작업을 수행하는 것을 배울 것입니다. 4 가지 기본작업으로 복잡한 작업을 수행하기 전에, 리보그 월드가 어디 있는지, 그리고 어떻게 생겼는지를 탐험하도록 여러분을 안내할 것입니다.

준비가 되면, 아래처럼 노트북의 두번째 페이지[RUR: Code and Learn]를 클릭하면 다음 화면이 보일 것입니다.



클릭하여 정상적으로 작동되면 아래와 같은 화면이 보일 것입니다.

왼쪽에 “프로그램창(Program Window)”이 있고, “로봇 월드(Robot World)”가 오른쪽에 있습니다.



리보그의 월드는 가로 세로 길로 구성되어 있습니다. 리보그의 위치를 식별하는데 편리한 이름을 다음과 같이 지었습니다. 북쪽(화면 위쪽으로), 동쪽(화면 오른쪽으로), 남쪽(화면 아래쪽으로), 서쪽(화면 왼쪽으로). 리보그는 4 가지 방향으로 움직일 수 있습니다. 리보그의 처음 위치는 가로 길의 첫번째, 세로길의 첫번째 위치해 있고 동쪽을 바라보고 있으며, 동쪽으로 이동할 준비를 마쳤습니다.

여러분 차례

리보그를 클릭해서 잠에서 깨우세요. 리보그가 반응하길 기대하지는 마세요. 단지, 리보그는 단지 준비가 되었습니다. 이제, 키보드의 위쪽 화살표()를 사용해서 리보그를 움직이세요. 위쪽 화살표()는 리보그를 한칸 앞으로 이동시키고, 좌측 화살표()는 리보그를 좌측으로 돌게 합니다.

실험!

리보그를 경계를 넘어 월드 밖으로 이동시키면 무슨 일이 생기나요?



첫 번째 프로그램

앞서 언급한 바와 같이, 리보그와 상호 작용하는 데 사용하는 두 개의 주요한 창이 있습니다. 왼쪽에 프로그램 창(Program window): 리보그가 따라야 할 명령어를 작성하는 곳입니다. 오른쪽에 그래픽 월드 (Graphic World): 명령을 받은 리보그가 움직이는 곳입니다.

프로그램 열기 버튼()을 클릭하고, “move1.rur”을 선택합니다. 다음의 컴퓨터 코드(computer code), 줄여서 코드(code)가 프로그램 창에 보일 것입니다.

```
move()  
turn_off()
```

코드(code)는 프로그램 텍스트(program text)와 동의어입니다. 프로그램은 일련의 명령, 지시입니다. 이 경우 코드는 두 개의 지시문으로 구성되었습니다.

- move(): 리보그에 한칸 앞으로 전진하도록 지시합니다.
- turn_off(): 리보그에서 더 이상의 지시할 것이 없다고 알려줘서 에너지를 절약하도록 리보그의 전원을 끕니다.

명령과 괄호’()’로 구성된 함수는 리보그가 반드시 지켜야 하는 명령문으로 인식합니다.

이제, 실행버튼()을 클릭하여 리보그가 ‘리보그 월드’에서 움직이는 것을 지켜보세요.

여러분 차례

아래와 같이 두 번째 함수 move() 을 추가하여 프로그램을 바꾸세요.

```
move()  
move()  
turn_off()
```

저장버튼()을 클릭하고, “move2”로 이름을 주세요. 프로그램에 자동으로 확장자(“.rur”)가 추가됩니다. 축하합니다. 여러분은 방금 전 여러분의 프로그램을 작성하셨습니다. 이제 실행버튼()을 클릭하여 리보그가 프로그램을 실행하도록 하세요.

실험!

리보그에게 두 개 이상의 작업을 수행하도록 해보세요. 만약 리보그가 너무 많은 작업을 수행해서 월드의 경계를 넘어가게 되면 무슨 일이 발생할까요? 특히, 프로그램 창을 보세요. 리보그가 명령을 수행하게 될 때 해당 명령문이 강조(하이라이트)되는 것을 보실 수 있습니다.



오류(eRRoRs) 다루기

리보그는 문자 그대로 명령을 따릅니다. 이것은 좋기도 하구, 나쁘기도 합니다.

예를 들어, 다음과 같이 프로그램을 작성합니다

```
Move()  
turn_off()
```

“wrong_move.rur”로 저장하고 실행합니다. 리보그는 자신만의 특별한 방식에 불평합니다. 왜냐하면 Egrebor에게 M과 m(대문자, 소문자)가 다르기 때문입니다.

여러분 차례

정확하게 무슨 일이 일어나는지를 확인하세요. 앞으로 Egrebor가 한 이런 종류의 불평을 보게 된다면, 여러분이 작성한 프로그램에서 무엇이 잘못된 것인지를 알 수 있을 것입니다. 다른 말로, 어떻게 버그(bug)이지를 파악하는 것을 배울 것입니다. 잠시 후에 버그에 대해서 더 배우게 될 것입니다. 여러분이 버그가 있는 프로그램을 작성한다면, 조만간 배우게 될 것입니다.

퀴즈

여러분이 작성할 수 있는 가장 짧은 유효한 프로그램은 무엇입니까? 유효한 프로그램이란... 리보그가 어떠한 오류 메시지도 없이 프로그램을 수행하는 것을 의미합니다. 간단한 프로그램이 예상한 대로 작동하는지를 확인하기 위해서 시도해 보고 확인해 보세요.

◀ 첫 번째 프로그램 -  - 주석(Comments) ▶

주석(Comments)

여러분이 작성한 첫 번째 프로그램을 여세요. 여러분은 저장을 했고 제가 여러분에게 부탁 드린 대로 데로 가지고 있습니다. 만약 그렇지 않다면 기다리는 동안 다시 작성하세요.

좋습니다. 여러분이 작성한 프로그램은 프로그램 창에 표시되어야 합니다.
이제, 제가 프로그램을 작성할 때 여러분에게 말씀 드린 가장 중요한 것을 기억하십니까? 만약 기억하지 못한다면, 다시 제일 처음으로 돌아가서 읽어보세요.

기다리겠습니다.....

이해하시죠... 그렇습니다. 가장 중요한 것은 다른 사람이 여러분의 프로그램을 읽기 쉽게 만들어야 됩니다. 많은 연습과 생각이 요구됩니다. 하지만, 프로그램 안에 컴퓨터가 아닌 다른 사람을 위한 메모를 적는 요령이 있습니다. 메모나 노트를 주석(Comments)이라고 합니다. 프로그램에 주석을 적는 몇 가지 방법이 있습니다. 파이썬에서 사용하는 가장 간단한 방법을 가르쳐 드리겠습니다. 프로그램의 첫 줄에 다음 텍스트를 추가하세요.

나의 첫 프로그램 (My first program)

프로그램은 다음과 같을 것입니다.

```
# My first program
move()
move()
turn_off()
```

여러분이 색맹이 아니라면 #으로 시작하는 첫 줄이 녹색으로 보일 것입니다. 리보그(혹은 파이썬)는 # 기호로 시작하는 나머지 부분 주석이다라는 것을 나타냅니다. 주석의 텍스트가 녹색이라 다른 명령문과 식별하는데 도움을 줍니다. 왜 녹색으로 했는지는 특별한 이유가 없습니다. 단지 러플(RUR-PLE) 프로그램을 작성할 때 녹색을 선택했고, 그래서 주석이 녹색으로 보입니다.

여러분 차례

명령문 시작 줄에 # 기호를 붙이면 무슨 일이 일어날까요? #을 붙여보고, 프로그램을 저장하고 실행버튼()을 클릭하여 리보그가 무슨 일을 하는지 지켜보세요.

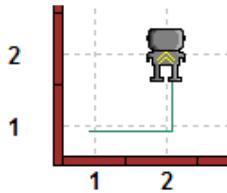
 오류(eRRoR) 다루기 -  - 왼쪽으로 돌기 

왼쪽으로 돌기

앞에서 언급했듯이, 리보그는 왼쪽방향으로, 왼쪽방향의 90 도로 어느 방향이든지 돌 수 있습니다. 리보그를 만든 이가 이보다 더 좋은 체계를 구현할 수 없어서 그렇습니다. 리보그가 왼쪽으로 돌게 하려면, `turn_left()` 라고 작성하면 됩니다.

예를 들어, 리보그가 처음 시작 위치에서 출발한다면, 아래 간단한 명령문은 다음 화면을 뿌려줍니다.

```
move()  
turn_left()  
move()  
turn_left()
```



여러분 차례

다음 프로그램을 사용하여 리보그로 하여금 간단한 사각형을 따라 움직이게 하세요.

```
move()  
turn_left()  
move()  
turn_left()  
move()  
turn_left()  
move()  
turn_left()  
turn_left()  
turn_off()
```

필히, 프로그램을 실행하기 전에 작성한 프로그램을 저장하는 것을 잊지 마세요.

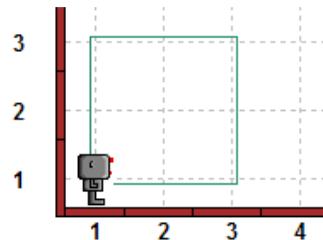
시도해 보세요.

영어가 모국어가 아니라면, 먼저 정의한 후에, 여러분의 모국어의 동의어를 만들어서 작성할 수 있습니다. 하지만, 동의어는 필히 영어 알파벳이어야 하고,

강세 등의 표식이 없어야 합니다. 예를 들어, 프랑스어로, vire_a_gauche = turn_left 정의하고, vire_a_gauche() 사용하여 로봇이 왼쪽으로 돌게 명령할 수 있습니다.

다시 여러분 차례

사각형을 따라서 반시계 방향으로 리보그가 움직이는 프로그램을 작성하세요.
앞으로 2칸 이동하고, 왼쪽으로 돌기를 반복하여 처음 시작위치로 돌아와서
아래와 같이 동쪽을 바라보는 모습으로 서 있어야 합니다.



실험!

리보그 월드에서 다양한 경로를 따라 리보그가 이동하는 다른 프로그램을
자유롭게 만들어 보세요.

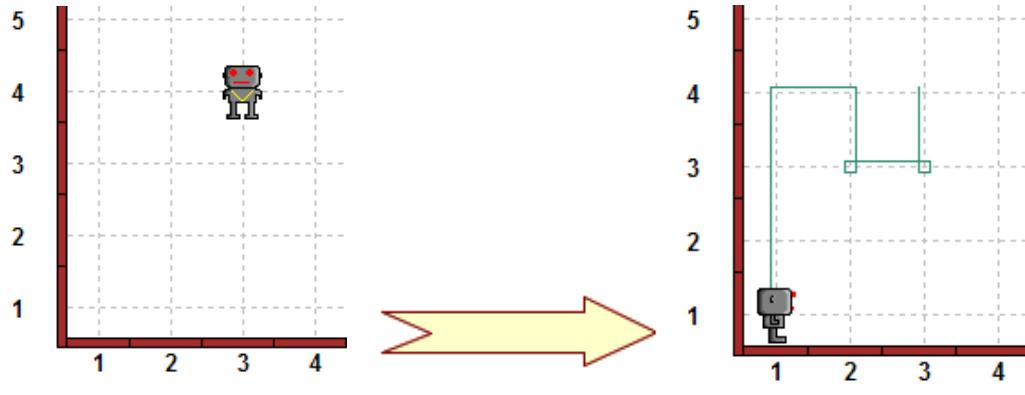
집으로

리보그 월드에 들어갈 때, 리보그는 통상 왼쪽 아래 모퉁이에서 있고 동쪽을 향해 있습니다. 오른쪽 끝쪽에 있는 “Show/Hide world file” 버튼 (wld)을 클릭하세요. 오른쪽 옆에 텍스트가 나타나며 화면에 변화를 볼 수 있습니다. 특히, `robot = (1, 1, 'E', 0)` 줄을 주목해서 보세요. 이 줄이 월드에서 리보그의 위치를 나타냅니다. 첫 번째 숫자는 세로 길(열)의 위치, 두 번째 숫자는 가로 길(행)의 위치, 인용부호 한의 문자는 리보그가 향하는 방향(E=East(동쪽), N=North(북쪽), W=West(서쪽), S=South(남쪽)), 동쪽은 오른쪽을 향하고, 북쪽은 위쪽을 향하고 등등입니다. 마지막 숫자 (0)는 리보그가 갖고 있는 비퍼(beeper) 숫자입니다. 비퍼에 대해서는 조금 후에 학습할 것입니다.

앞에서 보았듯이, 컴퓨터 키보드의 방향키를 사용하여 리보그를 조종할 수 있습니다. 리보그가 키보드로 움직일 때는 어떤 훈적도 남기지 않습니다. 하지만, 리보그가 움직일 때, `robot = (...)` 괄호안의 텍스트가 월드에서 새로운 위치를 나타내기 위해 바뀌는 것을 알 수 있습니다.

여러분 차례

키보드를 사용하여 아래 그림과 같이 3 번째 열, 4 번째 열에 리보그가 남쪽을 향하도록 움직여보세요. 리보그가 처음의 위치로 돌아가는, 즉 첫 번째 열, 첫 번째 행 동쪽을 향한 초기 상태의 리보그가 되도록 프로그램을 작성하세요. 아래 보인 것보다 더 적은 프로그램 단계가 걸리도록 프로그램을 작성하세요.



◀ 주석(Comment) - - 비퍼(Beepers) ▶

비퍼(Beepers)

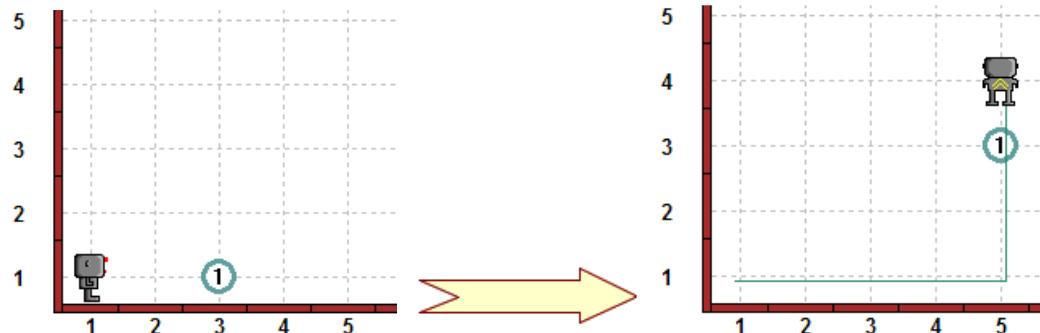
1. 빵! 빵!

리보그의 월드는 비퍼를 포함합니다. 비퍼는 커질 때 작은 소리를 만드는 작은 물체입니다. 화면상에는 꽤 크게 그려집니다. 리보그는 비퍼 바로 옆에 서 있을 때 비퍼 소리를 들을 수 있습니다. 리보그는 비퍼를 주울 수 있고, 주머니에 꺼진 상태로 가지고 다니거나, 자리에 놓을 수 있는데 이 경우 자동으로 꺼집니다.

리보그로 하여금 pick_beeper() 명령문으로 비퍼를 줍게 하거나, put_beeper() 명령문으로 자리에 놓게 할 수 있습니다. 비퍼가 없는 곳에서 비퍼를 줍게 하거나, 비퍼가 주머니에 없는데 자리에 놓게 한다면, 리보그는 불평하고, 바로 종료합니다.

여러분 차례

왼쪽 아래 그림 같은 beepers1.wld 파일을 여세요. 월드 열기버튼(을 클릭하여 월드(world) 파일을 여세요. 비퍼를 줍고, 아래와 같이 가지고 이동한 후에 내려놓는 프로그램을 작성하세요.



2. 비퍼 변경

비퍼를 추가해서 리보그의 월드를 쉽게 바꿀 수 있습니다. 이를 위해서, 가로 줄과 세로 열이 만나는 교차점에서 우클릭하여 0~20 사이의 비퍼 숫자를 선택합니다. 단, 0은 비퍼를 제거하기 위해서도 사용됩니다.

여러분 차례

beepers1.wld 파일을 열고, 1에서 5로 1행 3열에 위치한 비퍼의 숫자를 증가시키세요. Beepers2.wld라는 이름으로 월드 저장버튼()을 눌러 저장하세요. 리보그가 모든 비퍼를 주워서 한 칸 앞 앞으로 이동하는 프로그램을 작성하세요.



이놈의 버그!

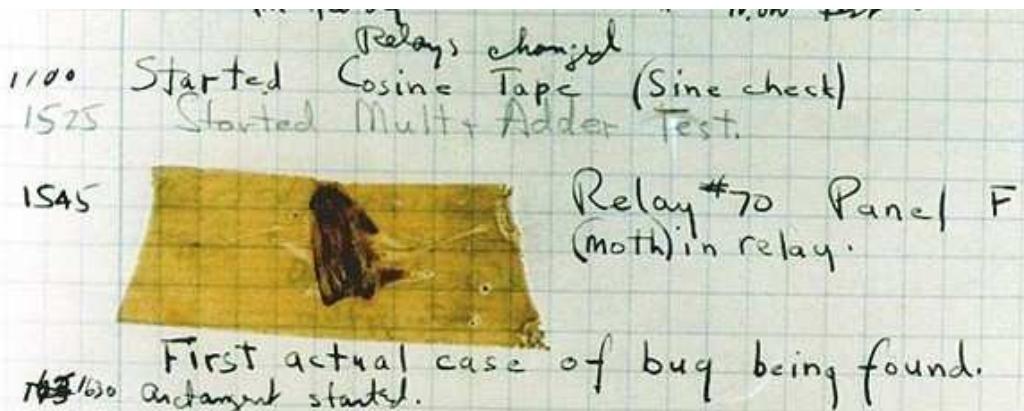


누구도 컴퓨터 버그에 관해서 얘기하는 것을 좋아하지 않아요. 그래서 이번 학습은 매우 짧습니다. 제가 여러분께 부탁 드리는 것은 단지 읽어만 보세요. 여러분이 버그있는 프로그램을 작성하지는 않습니다.

1. 버그(Bug)란 무엇인가?

컴퓨터 용어 버그(Bug)의 어원은 하버드 대학 마크 II 컴퓨터 내부에서 나방이 발견된 실제 사건에 기원하는데, 분명히 나방이 컴퓨터가 멈추게 하는 원인을 제공했습니다. 컴퓨터 프로그래밍의 컴파일러 언어 개념을 발명한 저명한 컴퓨터 과학자이자 수학자이며 해군 장교인 Grace Murray Hopper 가 이끄는 팀에서 발견되었습니다. Grace Hopper 박사는 종국에는 미 해국 소장까지 진급했습니다.

이 나방(Moth)은 하퍼의 기록지에 테이프되어 보관되었다. 흥미롭게도 기록지에는 다음과 같은 메모가 적혀있습니다. “버그가 발견된 첫 실제 사례(First actual case of bug being found.)”



사실, 기술적인 맥락에서 버그라는 단어는 옥스퍼드 영어 사전에는 토마스 에디슨에 기인한다고 적혀있습니다. 옥스퍼드 영어 사전에 따르면, 1889년 3월 11일 Pall Mall Gazette 판에는 다음과 같은 텍스트가 적혀있습니다.

“제가 듣기로는 에디슨씨가 측음기에 ‘버그’가 발견되어 두 밤을 쌌습니다. 버그는 풀기 어려운 문제를 해결하는 표현으로 어떤 가상의 벌레가 은밀하게 내부로 들어가서 모든 문제를 일으키는 것입니다.”

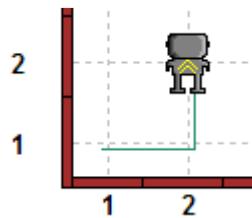
원 “버그”는 실제로 벌레일지 모르지만, 사실 가상으로 보입니다. 불행하게도, 컴퓨터 버그는 가상이지는 않습니다.

2. 버그 다루기

컴퓨터 전문용어인 버그는 프로그램이 예기치 않은 방식으로 작동하는 오류입니다. 컴퓨터 프로그램을 작성한다면, 모든 사람이 그렇듯이 조만간 버그를 가질 것입니다. 좋은 프로그래머는 버그를 제거하고, 프로그램이 예기치 않게 작동한다는 것을 알자마자 프로그램을 수정합니다.

러플(RUR-PLE)은 버그를 잘 찾도록 설계되었습니다.

1. 로봇의 기름이 누출되어 흔적을 남기는데 로봇에 의해 수행된 명령문을 추적하여 작성된 프로그램을 확인할 수 있습니다.

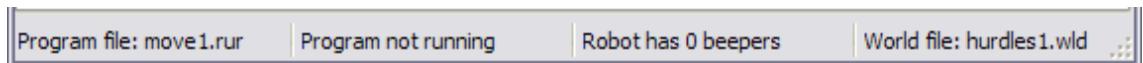


2. 각 명령문 (예로, ‘pick_beeper()’ 처럼)은 로봇에 의해서 수행될 때 프로그램 윈도우에서 눈에 띄게 됩니다.

```
3 move ()  
4 pick_beeper ()  
5 move ()  
6 put_beeper ()
```

3. 프로그램이 실행될 때, 정지 버튼(■)을 클릭하여 프로그램을 정지시킬 수 있습니다. 컴퓨터 프로그램에서 “정지점(breakpoint)”을 설정하는 것과 유사합니다.
4. 명령실행 그리고 멈추거나 단계별로 실행버튼(▶)을 클릭하여 명령어를 하나 실행하고 멈추고(Step Through)를 통해 한 스텝씩 프로그램을 수행할 수 있습니다.
5. 오래 실행되는 프로그램에 대해서는 스피드 버튼(⌚)을 클릭하여 빨리 실행시킬 수 있습니다. 빨리 프로그램을 실행하고, 예기치 못한 행동을 보이는 부문(‘버그’)에서는 잠시 멈추고, 한번에 한 명령어 실행을 통해서 자세히 들여다 봅니다.

6. 프로그램을 시작한 후에, 끝까지 가기 전에 프로그램을 멈추고 싶다면 빨간색 정지 버튼()을 클릭하여 프로그램을 어느 때고 정지시킵니다.
7. 화면 하단에 상태표시 막대가 있습니다. 1) 어느 프로그램이 로드되어 있는지, 2) 그 프로그램의 상태는 어떤지, 3) 로봇이 주머니에 가지고 있는 비퍼의 숫자는 얼마인지, 4) 어떤 월드 파일이 로드되어 있는지, 1) ~ 4)를 상태 표시 막대는 표시합니다.



비퍼와 월드 파일에 대해서는 나중에 좀 더 살펴볼 것입니다.

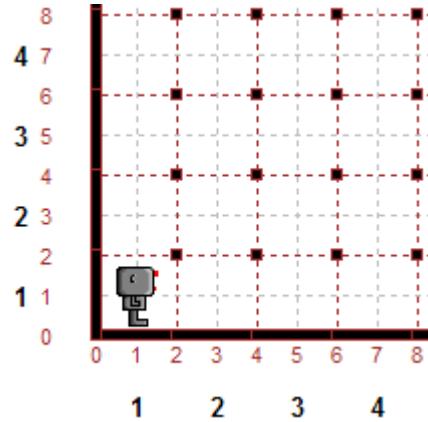
8. 처음 월드파일을 열어, 로봇이 처음 시작 지점으로 위치하도록 재시작 하고자 한다면, 월드 리셋 버튼()을 클릭합니다. (이부분에 대해서은 아직 자세하게 설명하지 않았습니다.)

지금 충분한 것보다 많을 수 있습니다. 나중에 버그를 찾고, 고치는 다른 요령을 보여줄 것입니다.

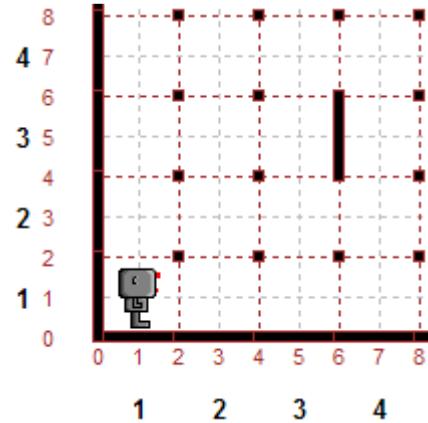


벽 만들기

벽을 추가해서 리보그 월드를 쉽게 바꿀 수 있습니다. 벽버튼()을 클릭하면 화면이 바뀌어서 아래처럼 보입니다.



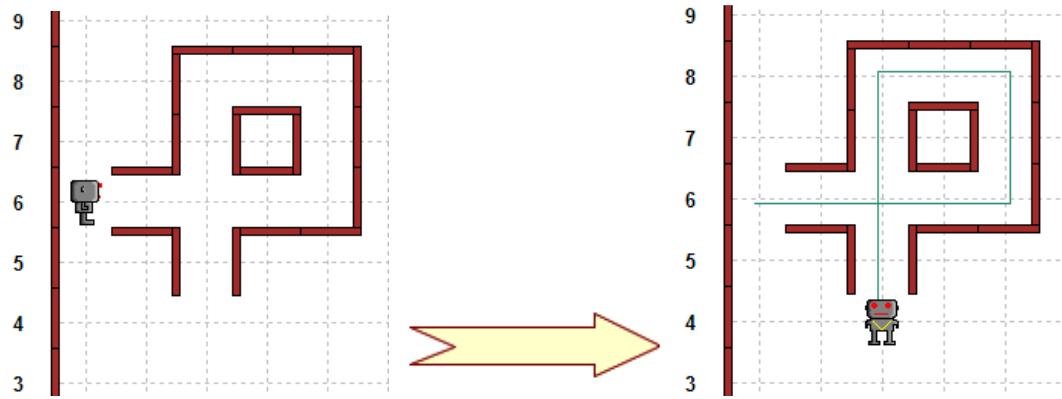
두 기둥(까만 점은 위에서 내려다 본 기둥) 사이에 빨간 점선을 클릭해서 벽을 추가하고 삭제합니다. 하지만, 가장자리 벽은 제거할 없습니다.



다시 벽버튼()을 클릭하면 기둥과 빨간 점들이 사라지고, 새로 만든 벽만 남게 됩니다.

여러분 차례

아래에 표시된 것과 같은 월드를 만들고 저장하세요. 표시된 선을 따라 간단한 미로를 빠져 나와 하단에 나타나도록 리보그를 안내하는 프로그램을 작성하세요.

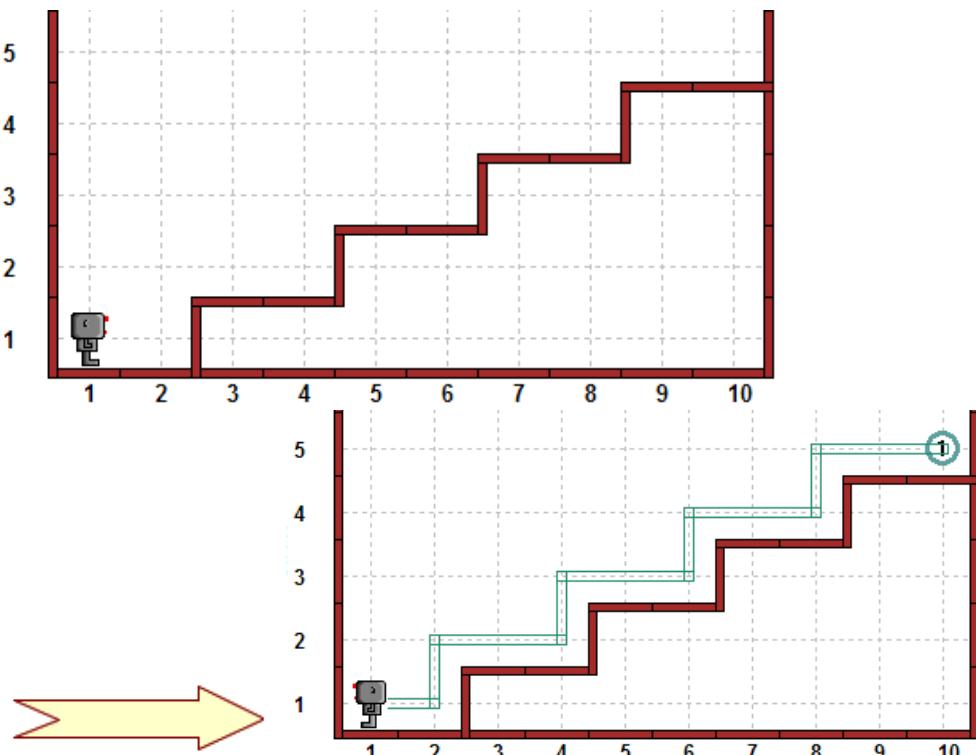


도전

연습을 몇 번하고 이번 학습을 마무리 합니다. 이번 학습의 연습문제는 너무 어렵지 않습니다. 사실 이번 단계에서 약간 지루하게 느껴질 수 있지만, 다음 학습에는 정말 도움이 많이 될 것입니다.

신문 배달

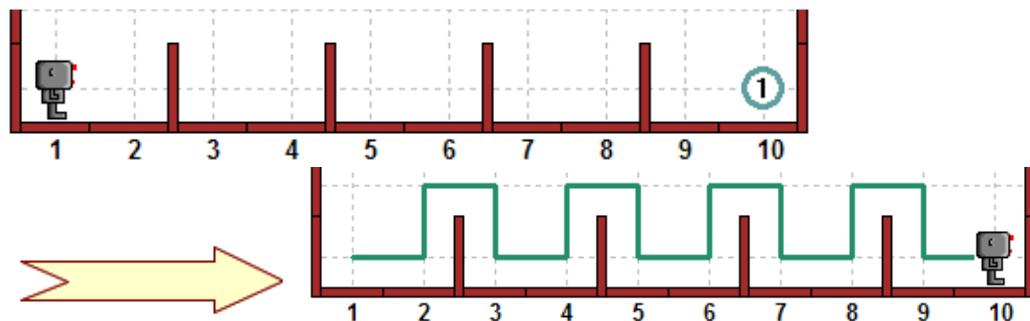
리보그가 지역 근처에 신문을 배달합니다. 집 앞에 계단을 올라가서, 신문을 마지막 계단에 놓고 (비퍼로 표현), 다시 아래와 같이 처음 시작지점으로 돌아오는 것입니다. 월드 파일은 newspaper.wld 입니다.



이 작업을 수행하기 위해서 리보그가 50 이상의 명령문을 수행해야 한다는 것을 알게 되고, 꽤 많은 키보드 입력을 해야 합니다. 더 많은 타이핑을 할수록 더 많은 오류를 만들 수 있습니다. 다음 학습에서, 어떻게 파이썬을 사용하여 이 문제의 해답으로 간략화할 수 있는지 살펴볼 것입니다.

장애물 넘기

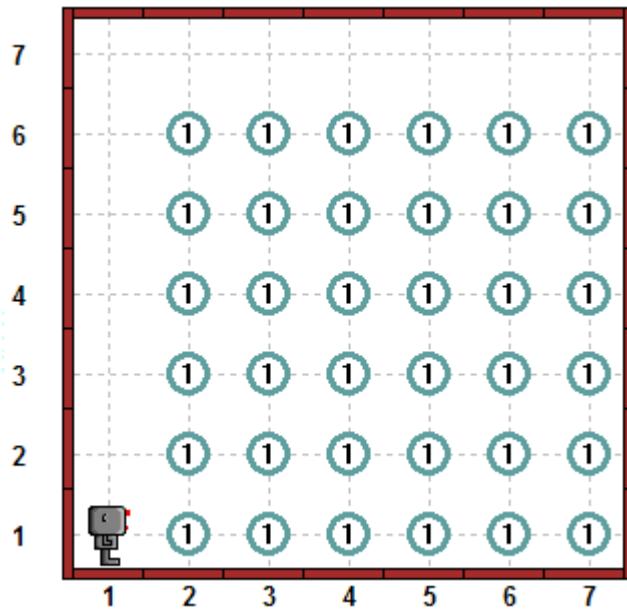
리보그가 장애물 넘기 경주에 참가합니다. 아래 보여진 경로를 따라 결승선에 리보그가 도착하도록 프로그램을 작성하세요. 월드 파일은 hurdles1.wld 입니다.



다음 학습에서, 같은 작업을 수행하는 짧은 프로그램을 어떻게 작성하는 것을 보게 됩니다. 그 후에는 리보그를 지도해서 자동으로 다른 높이의 다른 넓이를 가진 장애물을 넘게 하는지를 배울 것입니다.

수확 시기

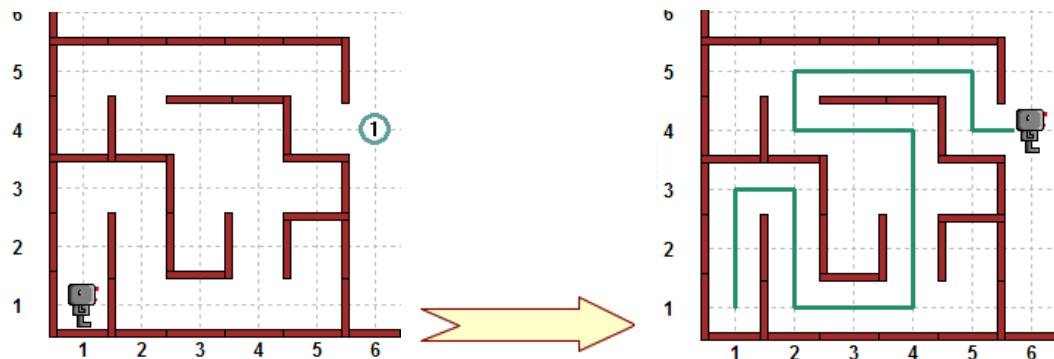
수확의 계절입니다. 정원의 모든 당근(비퍼로 표현)을 리보그가 수확합니다. 월드 파일은 harvest1.wld 입니다.



나중에, 동일한 작업을 수행하는데 좀더 짧은 프로그램을 작성하는 것을 배울 것입니다. 그리고, 리보그에게 어떻게 정원의 임의의 장소에 당근이 빠진 장소를 처리하는 방법을 가르칠 것입니다.

놀라운 미로

리보그가 미로에 빠졌습니다. 리보그가 미로를 빠져 나오도록 도와주세요. 가장 짧은 경로가 아래 표시되어 있습니다. 월드 파일은 maze1.wld 입니다.



나중 수업에서, 많은 다른 종류의 미로를 빠져 나오는 동일한 프로그램을 사용해서, 리보그가 어떻게 미로를 스스로 빠져 나오는지를 볼 것입니다.

이놈의 버그 - - 확실히 반복 피하기

확실히 반복 피하기

이번 학습에서 새로운 로봇 명령문을 어떻게 정의하는지 배울 것입니다.

컴퓨터 프로그램을 작성할 때, 3 번째 유용한 규칙을 보게 됩니다.

규칙 3.

프로그램을 작성할 때, 자신을 반복하지 마세요.

다시 말씀드립니다. **자신을 반복하지 마세요.**

1. 원쪽으로 세 번 도는 것은 오른쪽 한번 돌게 만들니다.

곰곰이 생각한다면, 리보그가 원쪽으로 세 번 돌게 하는 것은 오른쪽으로 한번 도는 것과 같은 결과를 줍니다. 종이 위에 그려서, 컴퓨터 없이 다음 프로그램이 리보그가 수행하는 것을 이해해 보세요.

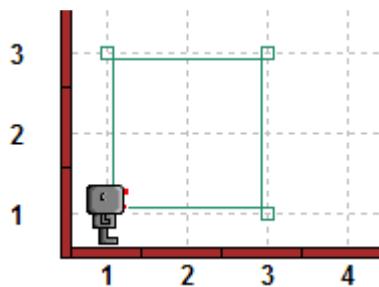
```
turn_left()  
move()  
turn_left()  
turn_left()  
turn_left()  
move()  
move()  
turn_left()  
turn_left()  
turn_left()  
move()  
turn_left()  
turn_left()  
turn_left()  
move()  
move()  
turn_left()  
turn_left()  
turn_left()
```

여러분 차례

위 프로그램을 작성하고 저장하세요. 리보그가 여러분이 기대하는 것을 리보그가 수행하는지 살펴보세요.

다시 여러분 차례

방금 전에 저장한 프로그램을 수정하여, 아래 사각형에서 보이듯이 리보그가 시계방향으로 선을 따라 돌도록 만드세요.



2. 오른쪽 돌기 정의

앞에서 3 번의 왼쪽 돌기를 조합해서 리보그가 어떻게 오른쪽으로 한번 도는지를 살펴봤습니다. 일련의 오른쪽 돌기를 만든다면, 작성된 결과 코드를 작성하고 읽는 것은 무척이나 지루한 일이 될 것입니다. 왜냐하면, 반복을 계속하기 때문입니다. 다르게 말씀 드려, 동일한 일련의 명령문이 프로그램에서 다른 곳에 여러 번 반복하여 나타난다는 것입니다. 이런 중복을 피하기 위해서, 파이썬에 프로그램된 리보그의 능력은 매우 도움이 됩니다.

파이썬에서, 일련의 명령문에 간단한 이름을 부여할 수 있습니다.

예를 들어, 아래와 같이 리보그가 오른쪽 회전 명령문을 정의할 수 있습니다.

```
1 #define it.
2 def turn_right():
3     turn_left()
4     turn_left()
5     turn_left()
6
7 #use it!
8 turn_right()
9
```

4 가지 중요한 점에 주목하세요.

- 앞에서 보았듯이, 녹색의 # 기호는 같은 줄의 나머지 모두 리보그(혹은 파이썬)에서 무시합니다. # 기호 텍스트를 주석이라고 하고 다른 프로그래머에게 설명하거나, 다음의 프로그램 명령문이 무슨 역할을 하는지에 대해서 우리 자신에게 상기시키는 용도로 사용됩니다. 녹색으로 보여, 명령문들과 구별되도록 도움을 줍니다. 파이썬 혹은 리보그는 주석을 무시합니다.

- 둘째로, 정의는 에디터에서 청색으로 색깔 칠해진 파이썬 키워드 ‘def’입니다. 파이썬 키워드(keyword)는 파이썬 자신이 정의한 자신만의 의미를 가진 단어입니다. def 키워드는 새로운 명령어 이름, 괄호, 그리고 콜론으로 구성됩니다.
- 셋째, 새로운 정의의 부분들의 각 명령문들은 동일하게 들여쓰기가 됩니다. 들여쓰기에 문제가 있다면, 파이썬이 불평을 하거나 우리가 기대하는 것을 수행하지 않을 수 있습니다. 들여쓰기는 시작 줄에 빈 공백을 두는 것입니다. 주어진 코드 블록에 4 개의 빈 공백을 파이썬에서 관습적으로 사용합니다.
- 넷째, 명령문을 정의하는 것은 앞에서 동의어를 새로 만드는 것과는 다른 것입니다. (동의어를 새로 만들 때, 동의어 사이에 ‘=’ 같음 기호를 사용했고 괄호가 없는 것은 명령어가 아니라는 것을 의미합니다.)

한번에 너무 많은 정보를 보여준 것처럼 보입니다. 이 새로운 키워드를 얼마나 잘 이해하고 있는지 확인하는 좋은 시간입니다.

여러분 차례

아래 1), 2)를 만족하는 프로그램을 작성하세요.

- 1) 오른쪽으로 회전하는 새로운 명령문을 정의하세요.
- 2) 새로운 명령문을 이용하여 앞에서와 동일하게 리보그가 시계방향 사각형 선을 따라서 움직이게 하세요.

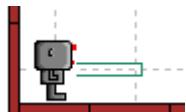
최종 프로그램이 처음 프로그램보다 짧고, 리보그가 움직이는 선을 이해하기 쉽다는 것을 알게 됩니다.

다시 여러분 차례

step_back() 명령문을 정의하세요.

```
# step_back() defined up here
move()
step_back()
turn_off()
```

`step_back()` 명령문은 리보그가 앞으로 한 칸 전진하고 다시 처음 출발자리로 돌아와서 아래와 같이 처음과 같이 같은 방향을 응시합니다.



힌트: 새로운 정의의 일부분인 명령문을 들여쓰기 하는 것을 잊지 마세요.

또 다시 여러분 차례

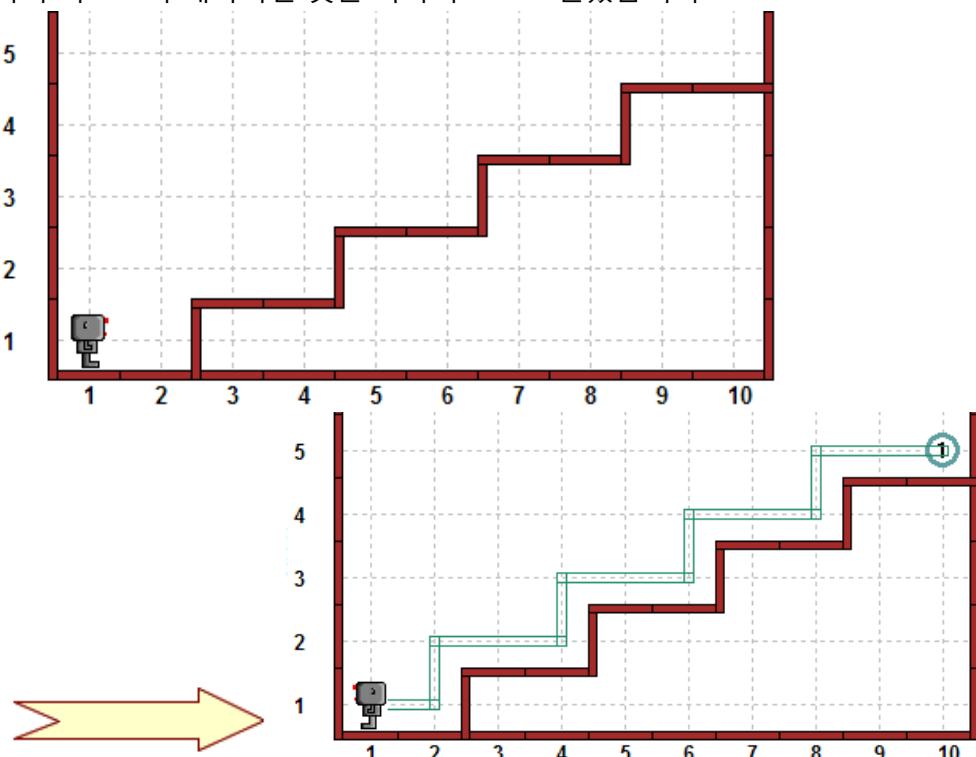
`turn_around()` 명령문을 정의해서, 다음 새로운 명령문이 여러분이 기대하는 것을 리보그가 수행하는지 확인하세요.

```
def step_back():
    turn_around()
    move()
    turn_around()

def turn_right():
    turn_around()
    turn_left()
```

3. 신문 배달, 다시

앞 선행학습에서 여러분이 작성한 마지막 연습문제 중에 하나가 리보그가 신문 배달하는 프로그램을 작성하는 것입니다. 신문배달 프로그램 상기하기 위해서 여기 리보그가 해야하는 것을 시각적으로 표현했습니다.



신문배달 문제의 해답은 아마 아래와 같을 것입니다.

```
move()  
# climb step  
turn_left()  
move()  
turn_left()  
turn_left()  
turn_left()  
move()  
move()  
# climb step  
turn_left()  
move()  
turn_left()  
turn_left()  
turn_left()  
move()  
move()  
# climb step  
turn_left()  
move()
```

```
turn_left()  
turn_left()  
turn_left()  
move()  
move()  
# climb step  
turn_left()  
move()  
turn_left()  
turn_left()  
turn_left()  
move()  
move()  
# put down newspaper and turn around  
put_beeper()  
turn_left()  
turn_left()  
# step down  
move()  
move()  
turn_left()  
move()  
turn_left()  
turn_left()  
turn_left()  
# step down  
move()  
move()  
turn_left()  
move()  
turn_left()  
turn_left()  
turn_left()  
# step down  
move()  
move()  
turn_left()  
move()  
turn_left()  
turn_left()  
turn_left()  
# step down  
move()  
move()  
turn_left()  
move()  
turn_left()  
turn_left()  
turn_left()  
# move away and stop  
move()  
turn_off()
```

타이핑이 정말 많고, 반복이 정말 많습니다. 프로그램 마지막에 도달했을 때는 프로그램의 시작을 화면에서 볼 수 없습니다. 프로그램에서 어디에 있는지를 추적하는 것을 돋도록 제가 몇 개의 주석을 넣은 것을 알 수 있습니다. 프로그램 프로그램 해결책 개요가 나올 때, 주석은 생각하는 블록에 가까이 위치합니다.

- 4 계단 오르기
- 신문 내려놓기
- 돌기
- 4 계단 내려오기

파이썬 방식(Pythonic form)으로 프로그램 개요를 적어봅시다.

```
climb_up_four_stairs()  
put_beeper()  
turn_around()  
climb_down_four_stairs()
```

신문배달 프로그램의 완벽한 해결책은 아닙니다. [예를 들어, turn_off() 명령문이 빠져있습니다.] 하지만, 전과 비교하여 읽기 훨씬 쉽고 다음의 새로운 명령문이 정의된다면 최종 프로그램에 좀 더 가까이 다가갔습니다. 여기 필요한 몇몇 정의 명령문이 있습니다.

```
def turn_around():  
    turn_left()  
    turn_left()  
  
def turn_right():  
    turn_left()  
    turn_left()  
    turn_left()  
  
def climb_up_one_stair():  
    turn_left()  
    move()  
    turn_right()  
    move()  
    move()  
  
def climb_up_four_stairs():  
    climb_up_one_stair()  
    climb_up_one_stair()  
    climb_up_one_stair()  
    climb_up_one_stair()
```

여러분 차례

빠진 정의 명령문을 추가하여 최종 프로그램이 파이썬 방식 버전처럼 보이게 만들어 보세요. `turn_off()` 포함해서, 몇 개 간단한 명령문을 추가할 필요가 있습니다. 프로그램을 저장하고, 원래 프로그램과 다른 이름을 사용하는 것을 잊지 마세요.

다시 여러분 차례

신문 배달 프로그램의 최초 버전과 가장 마지막 버전과 비교하는 시간을 가져보세요. 어느 것이 읽기 쉬운가요?

4. 읽기 도전

잘 지은 이름은 프로그램이 무엇을 수행하는 이해하는데 정말 도움을 줍니다. 마찬가지로, 잘못 지은 이름은 프로그램을 이해하는데 어려움을 줍니다.
[규칙 4를 보세요] 컴퓨터를 실행하지 말고, 다음 프로그램이 무엇을 하는지 이해해 보세요.

```
def a():
    turn_left()
    turn_left()

def b():
    turn_left()
    a()

def c():
    move()
    move()

def d():
    c()
    b()

def e():
    d()
    d()
    d()
    d()

    turn_left()
    e()
    b()
    turn_off()
```

a(), b(), c(), d(), e() 명령문에 좀더 서술적인 이름이 유용하다는 것을 알 수 있습니다.

◀ 벽 만들기 -  - 다시 반복 피하기 ▶

다시 반복 피하기

앞에서 본 규칙 3은 너무 중요해서 여러분이 잘 기억하도록 다시 반복하고 싶습니다.

규칙 3.

프로그램을 작성할 때, 자신을 반복하지 마세요.

다시 말씀드립니다. **자신을 반복하지 마세요.**

1. 반복 !

신문 배달 연습문제의 마지막 해결 프로그램에 여전히 반복이 남아 있습니다. 예를 들어 turn_left() 명령문이 turn_right() 정의 명령문에 3 번 나타납니다. 동일하게, climb_up_one_stair()는 climb_up_one_stairs() 정의 명령문에 4 번 나타납니다. 자신을 반복하지 말자는 규칙에 반하는 것처럼 보입니다. 반복을 피하는 한가지 방법은 리보그가 특별한 명령문을 통해서 명령문을 반복하는 것입니다.

리보그가 명령문을 반복하도록 repeat() 명령문을 아래와 같이 사용합니다.

repeat(명령문 이름, 반복 횟수)

명령문의 이름 끝에 괄호, ‘()’, 가 없다는 것을 주목하세요. 예를 들어 오른쪽으로 회전하기를 다음과 같이 작성할 수 있습니다.

```
def turn_right():
    repeat(turn_left, 3)
```

여러분 차례

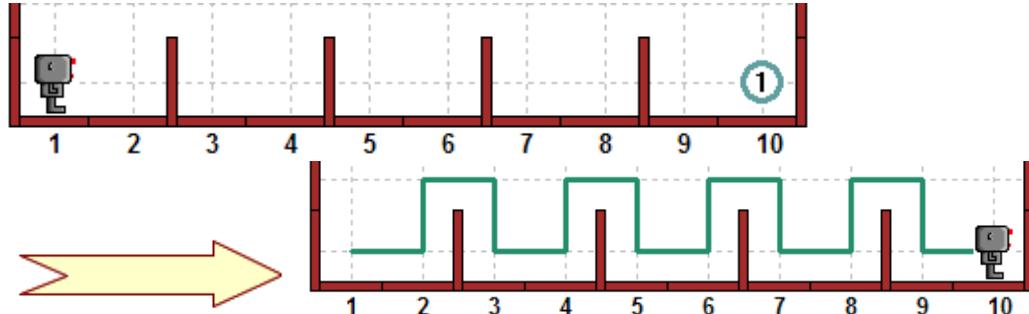
repeat 명령문을 사용해서 신문 배달 프로그램을 짧게 작성할 수 있습니다. 새로운 프로그램이 기대한 대로 잘 작성하는지 확인하세요.

2. 도전

연습문제를 몇 개 더 작성하고 이번 학습을 마무리 합니다. 이번 학습의 주 학습 내용을 존중해서, 첫번째와 두번째 연습문제는 비퍼 학습의 끝에 수행했던 연습문제를 다시 작성하는 것입니다. 이 문제를 풀기 위해서, 이번 학습 및 이전 학습에서 배운 새로운 개념 [def, repeat()]을 사용해서 프로그램을 작성해야 합니다.

장애물 넘기

리보그가 장애물 넘기 경주에 참가합니다. 아래 보여진 경로를 따라 결승선에 리보그가 도착하도록 프로그램을 작성하세요. 월드 파일은 hurdles1.wld 입니다.



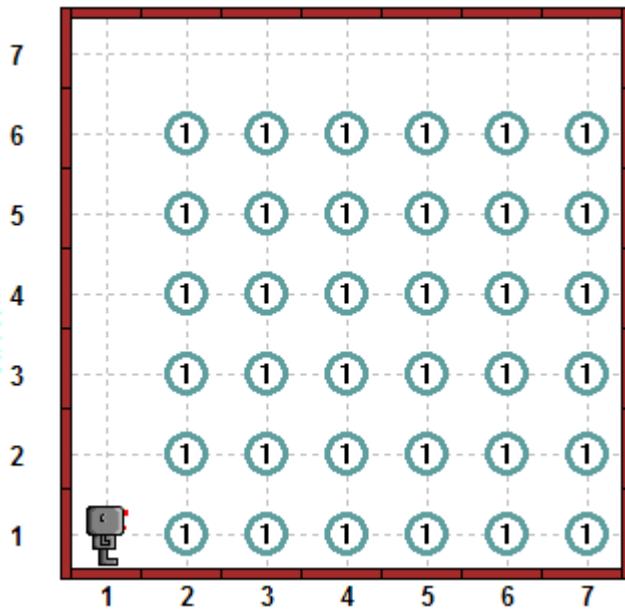
다음 경로에 상응하는 새로운 명령어 jump_hurdle()을 정의하여 유용해 보입니다.



새로운 프로그램 해결책을 이전의 것과 비교하여 보세요. [프로그램을 저장했습니까?, 저장했지요?]

수확 시기

수확의 계절입니다. 정원의 모든 당근(비퍼로 표현)을 리보그가 수확합니다. 월드 파일은 harvest1.wld 입니다.



여러분이 작성한 프로그램은 다음 명령문 함수를 정의해야 합니다.

```
move_to_first_row()
harvest_two_rows()
```

이 명령문 함수를 좀더 분해하길 원할지도 모릅니다. 예를 들어 여러분은 다음과 같이 작성할 수도 있습니다.

```
def harvest_two_rows():
    harvest_one_row()
    move_to_next_row()
    harvest_one_row()
    move_to_next_row()
```

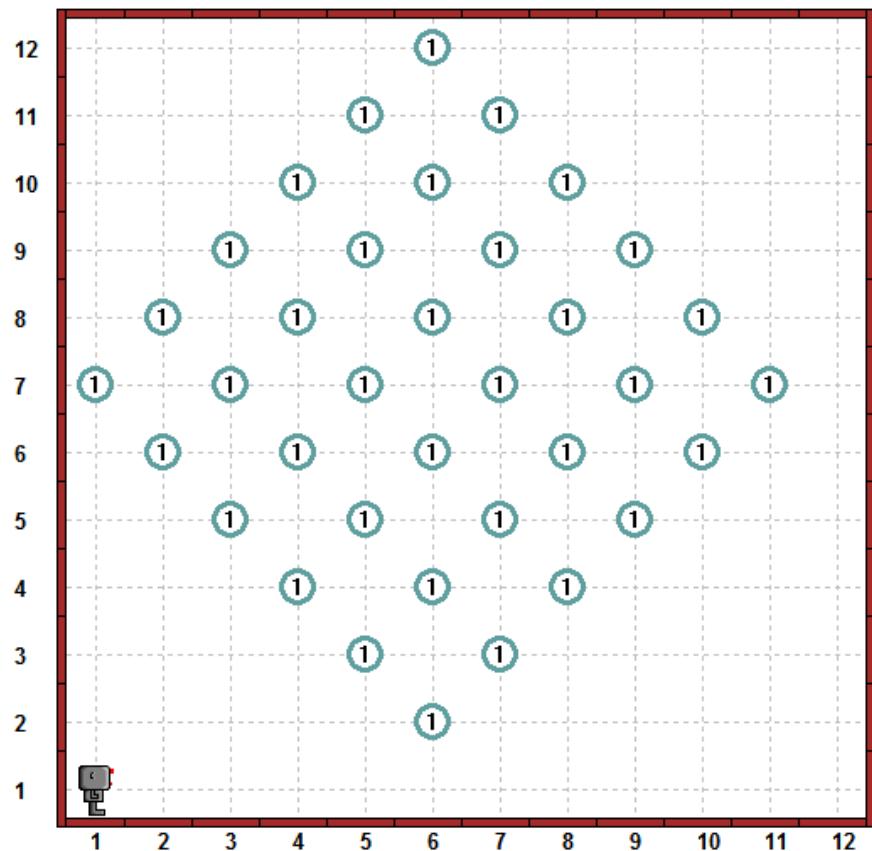
하지만, 두개의 필수 명령문을 분해하는 자신만의 방식을 선택할 수도 있습니다. 두 필수 명령문을 정의하고, 프로그램은 다음과 같이 작성될 수 있습니다.

```
move_to_first_row()
repeat(harvest_two_rows, 3)
turn_off()
```

다시, 수학 문제에서 앞서 작성한 프로그램과 새로운 프로그램을 비교하여 보세요.

다시 수확 시기

다시, 수확의 계절입니다. 하지만, 정원의 행이 대각선으로 정렬되어 있습니다. 리보그로 하여금 아래 정원의 모든 당근(비퍼로 표현)을 수확하도록 만드세요. 월드 파일은 harvest4.wld 입니다.



앞선 예제처럼, 여러분의 프로그램은 다음 명령문을 정의해야 합니다.

```
move_to_first_row()  
harvest_two_rows()
```

이 명령문들은 앞서와 같은 방식으로 정의되지 않을 것입니다. 하지만, 이 명령문 함수들을 정의하면, 앞선 예제 프로그램과 같은 방식으로 여러분의 프로그램을 작성해야 합니다.

```
move_to_first_row()  
repeat(harvest_two_rows, 3)  
turn_off()
```

벽 가장자리에 부딪히지 않도록 주의하세요. 수확을 어디서 시작해서, 어느 방향으로 수확을 할 것인지를 생각해 보세요. 종이 위에 스케치를 하여 경로를 그려보는 것이 도움이 될 수 있습니다.

◀ 확실히 반복 피하기 -  - 리보그 스스로 결정을 할 수 있다면 ➤

리보그 스스로 결정을 할 수 있다면

만약, 만약, 만약...

잠시만요!!! 리보그는 스스로 몇 가지 결정을 할 수 있습니다. 제가 말씀을 드리지 않았나요?

1. 첫 번째 결정

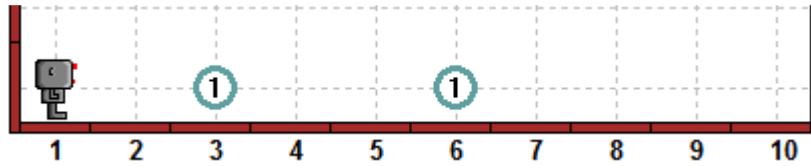
음... 사실대로 말씀드려, 리보그가 스스로 결정을 하기 위해서 약간의 도움이 필요합니다. 리보그에게 결정할 몇 가지 선택사항을 주어야 합니다. 예를 들어, 리보그가 비퍼 옆에 있을 때, 무엇을 해야 할지에 대해서 몇 가지 선택지를 줄 수 있습니다. 예를 들어, 리보그가 아래처럼 비퍼를 집으라고 할 수 있습니다.

```
if next_to_a_beeper():
    pick_beeper()
```

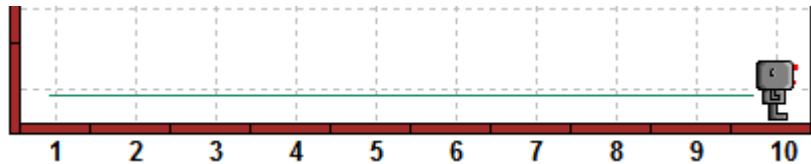
위의 코드 의미를 살펴봅시다.

- 파이썬 키워드 if 는 리보그에게 조건의 값이 참(True)이나 거짓(False)에 따라 행동을 취하게 합니다.
- next_to_a_beeper()은 조건 혹은 시험으로 리보그가 비퍼 옆에 있다면 (스크린에 같은 위치에 한다면)조건은 사실이 되고 그렇지 않다면 거짓입니다.
- 코론(:)은 조건이 참(True)이라면 리보그가 따라야 하는 명령문 앞에 선행합니다.
- 조건이 참인 경우 일련의 명령문은 정의 함수의 경우와 마찬가지로 들여쓰기를 합니다.

설명이 처음 읽을 때 복잡하게 보일 수 있지만, if 문을 사용하는 것을 사실 매우 간단합니다. 간단한 예제로 사용된 if 문을 살펴보세요. 리보그가 9 칸을 이동하면서, 길 중간에 있는 어떤 비퍼라도 줍는다고 가정합시다. 예를 들어, 시작 위치는 다음과 같이 보일 것입니다.



그리고, 최종 위치는 다음과 같을 것입니다.



리보그가 다음을 수행하기를 원할 것입니다.

- 앞으로 한 칸 전진하세요.
- 비퍼가 있는지 확인하세요.
- 비퍼가 하나 있다면 줍고, 그렇지 않다면 무시하고 다음 칸으로 이동하세요.

다음 명령문을 9 번 반복합니다. 만약 리보그에게 비퍼가 없는데 주우라고 명령을 한다면 리보그는 불평하고 꺼집니다. 여기에 어떻게 수행하는지 코드가 있습니다.

```
def move_and_pick():
    move()
    if next_to_a_beeper():
        pick_beeper()

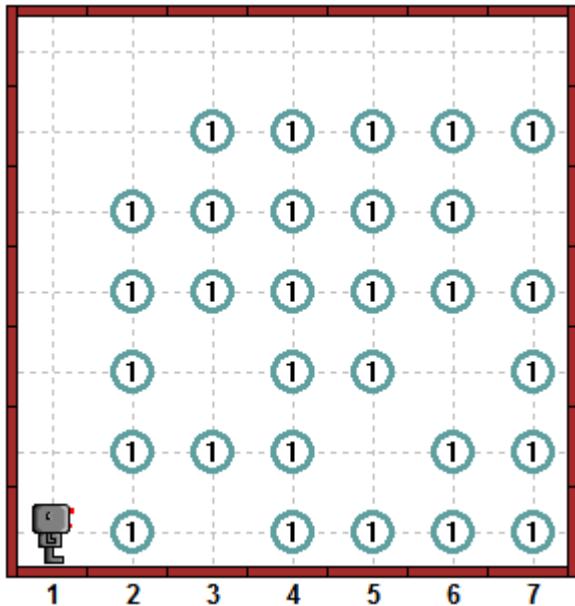
repeat(move_and_pick, 9)
turn_off()
```

시작해 보세요!

다시 수학 시기

다시 수학시기가 돌아왔습니다. 하지만, 이번에는 모든 당근 씨앗이 발아하지 않아서, 당근 몇 개가 정원에 비워있습니다. 정원에 있는 모든 당근 (비퍼로 표현)을 리보그가 수확하게 만드세요. 월드 파일은

harvest3.wld 입니다. 지난 학습시간에 했던 마지막에서 두 번째 수확 연습문제를 살펴보세요. 여러분이 해야 할 일은 아마도 harvest_one_row() 명령 함수를 수정해서 위의 move_and_pick() 명령 함수와 비슷할 것입니다.



여러분이 작성한 새로운 프로그램은 전에 사용한 harvest1.wld 월드 파일에서와 마찬가지로 작동해야 합니다.

◀ 다시 반복 피하기 -  - 제 말을 들으세요... 그렇지 않다면 ➤

제 말을 들으세요... 그렇지 않다면

프로그램을 작성하는 것을 배우는 것이 재미있을지 모르지만, 컴퓨터 앞에서 많은 시간을 보내고 싶지는 않을 것입니다. 만약 비가 내린다면, 책을 읽고, 그렇지 않다면 밖으로 나가서 노세요. 예심지어 할아버지도 말입니다.

1. 선택하기

if로 시작하는 문장으로 재미있게 놀아봅시다.

If it rains,
... keep reading,
otherwise,
... go outside and play!

위 문장은 마치 작은 컴퓨터 프로그램처럼 보입니다. 파이썬 프로그램처럼 다시 작성해 봅시다.

```
if it_rains():
    keep_reading()
else:
    go_outside_and_play()
```

방금 전에 여러분은 파이썬의 else 키워드를 배웠습니다. 그래서, 만약 비가 온다면, 책을 읽고, 그렇지 않다면 여러분은 무엇을 해야 할지 알고 있습니다.

2. 리보그가 “이제 보여요...”라고 말합니다.

하나 혹은 그 이상의 비퍼 옆에 리보그가 서 있는지를 알 수 있는 것에 더해서, 리보그는 앞에 벽이 있는지도 알 수 있어요. 리보그는 머리를 왼쪽 혹은 오른쪽으로 돌려서 벽이 있는지를 알 수 있어요. 리보그에게 다음 테스트를 요청할 수 있습니다.

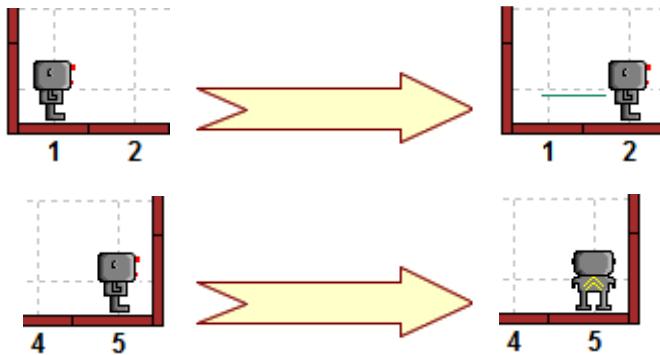
```
front_is_clear() # 만약 전방(앞)에 벽이 없다면 참(True), 그렇지 않으면 거짓(False)
left_is_clear()
right_is_clear()
```

리보그가 월드를 탐색하는데 첫 번째 명령문을 사용합시다. 만약 벽이 없다면 계속 전진하고, 벽이 있다면 왼쪽으로 회전을 합으로서 리보그가 월드의 경계를 따라 움직이게 할 수 있습니다. 간단한 다음 프로그램이 필요한 기초입니다.

```
if front_is_clear():
    move()
```

```
else:  
    turn_left()  
  
turn_off()
```

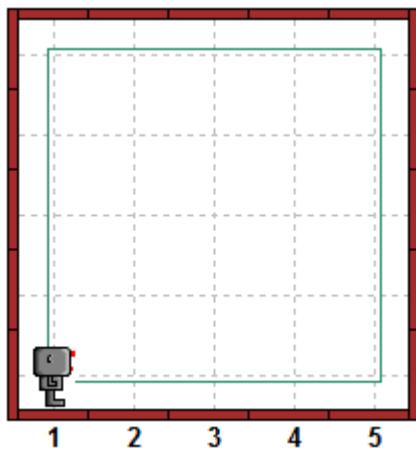
아래는 두 가지 다른 상황에서 간단한 프로그램을 실행시킨 결과입니다.



이제 간단한 조건 명령문을 많이 반복 실행해서 리보그가 월드를 돌아다니게 해봅시다.

```
def move_or_turn():  
    if front_is_clear():  
        move()  
    else:  
        turn_left()  
  
repeat(move_or_turn, 20)  
turn_off()
```

작은 월드에서 프로그램을 실행하면 다음의 결과를 줄 것입니다.



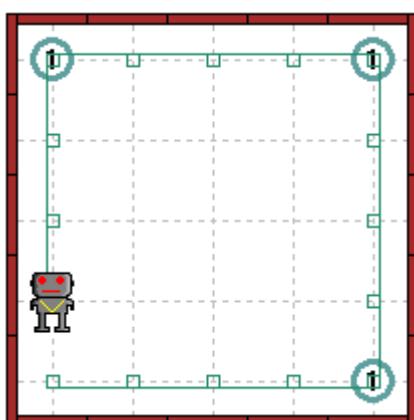
리보그가 앞으로 이동하고 방향을 바꾸면 비퍼를 내려 놓는다면, 리보그가 춤을 추게 함으로써 벽을 따라 이동하는 것을 좀더 흥미롭게 만들 수 있습니다.

이 작업을 수행할 수 있도록 충분한 비퍼를 리보그가 가지고 다니도록 확인하세요.
다음 프로그램은 리보그가 월드의 일부를 돌아다니며 작업을 수행하는 일부입니다.

```
def dance():
    repeat(turn_left, 4)
def move_or_turn():
    if front_is_clear():
        dance()
        move()
    else:
        turn_left()
        put_beeper()

repeat(move_or_turn, 18)
turn_off()
```

dance()와 move() 명령문이 if 문에 들여쓰기가 되어 있어서, 동일한 명령문 블록에 속해서 정렬되어 있는 것을 주목하세요. else 문에 속해서 들여쓰기 되어 있는 turn_left()와 put_beeper() 명령문도 비슷하게 정렬되어 있습니다. 프로그램 실행 결과는 아래 표시되어 있습니다.



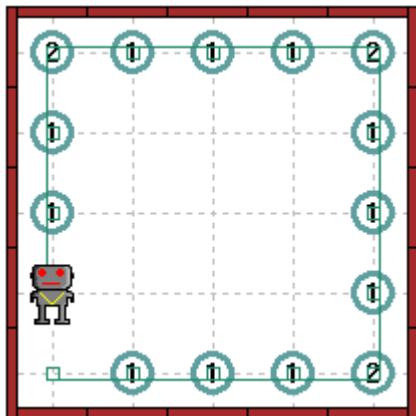
이제 put_beeper()와 turn_left()가 정렬되어 있지 않고, 아래와 같이 대신에 else 문과 같이 정렬한다면 무슨 일이 생길까요?

```
def dance():
    repeat(turn_left, 4)
def move_or_turn():
    if front_is_clear():
        dance()
        move()
    else:
        turn_left()
        put_beeper()

repeat(move_or_turn, 18)
```

```
turn_off()
```

이제 move_or_turn() 명령 함수는 매번 put_beeper() 명령문을 실행하고, 춤을 추고 앞으로 가든지 왼쪽으로 회전하는 if/else 선택을 포함합니다. 이 프로그램의 실행 결과는 아래 나타나 있습니다.



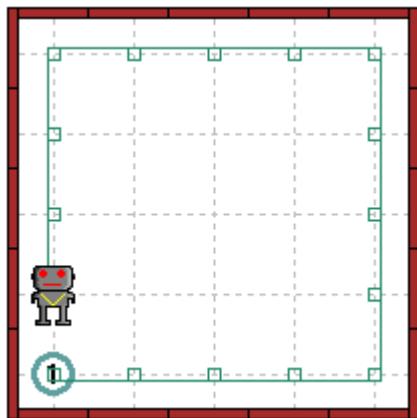
보시면, 매번 앞으로 리보그가 전진할 때, 비퍼를 자리에 놓습니다. 각 모퉁이에는 두 개의 비퍼가 있습니다. 하나는 모퉁이에 도달하는 앞선 이동에서 나온 비퍼고, 다른 하나는 모퉁이에 도착한 후에 왼쪽으로 회전하면서 나온 비퍼입니다.

이제 put_beeper() 명령문을 아래에 나타나 있듯이 def 문과 정렬하면 어떻게 될까요?

```
def dance():
    repeat(turn_left, 4)
def move_or_turn():
    if front_is_clear():
        dance()
        move()
    else:
        turn_left()
put_beeper()

repeat(move_or_turn, 18)
turn_off()
```

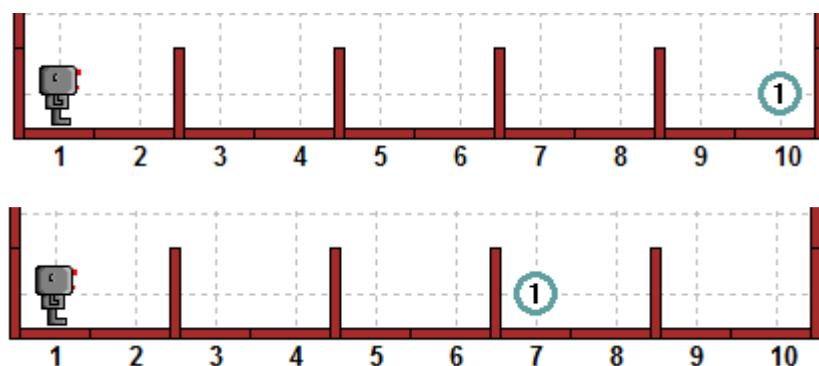
이제 put_beeper()는 정의 명령 함수문에 포함되어 있지 않아서 다른 정의 명령문과 정렬되어 들여쓰기가 되어 있지 않습니다. Put_beeper()는 단일 명령문입니다. move_or_turn() 명령 함수문을 18 번 실행하기 전에 리보그가 수행해야 하는 첫 번째 명령문입니다. 실행 결과는 다음과 같습니다.



지금까지 보시다시피, 빈 공백(즉, 블록 안에 명령문 들여쓰기)을 통해서 많은 정보가 리보그에게 전해집니다. 연습을 통해서, 빈 공백 들여쓰기를 잘 활용하는 방법을 배울 것이고, 명령문을 들여쓰기 함으로써 파이썬에서 매우 읽기 좋은 코드를 작성할 수 있다는 것도 학습할 것입니다.

장애물 넘기

리보그는 장애물 넘기를 잘 합니다. 리보그가 다른 길이의 경주(짧은 스프린트, 장거리 레이스)에 참가합니다. 리보그는 비퍼가 옆에 있을 때 결승선에 도착했다는 것을 인지합니다. 아래, 두 개의 경기 코스가 있습니다. 월드 파일은 hurdles1.wld, hurdles2.wld 입니다.



20 단위보다 더 긴 경주는 없다고 가정합시다. 아래와 같은 명령문 함수를 정의합시다.

```
def move_jump_or_finish():
    if next_to_beeper(): # 경주 끝
        turn_off()
    else:
        if front_is_clear(): # 경주가 끝나지 않았고, 넘을 장애물이 없다.
            move()
```

```
else:  
    jump_one_hurdle()
```

jump_one_hurdle() 명령 함수문으로, 다른 함수 정의문없이, 리보그가 수행할 필요가 있는 하나의 명령문은 다음과 같습니다.

```
repeat(move_jump_or_finish, 20)
```

위의 정의에서, 코드는 테스트를 추가함에 따라 점점 더 들여 쓰게 됩니다.

↳ 리보그 스스로 결정할 수 있다면 -  - if, else, if, else, ... ↳

if, else, if, else, ...

앞선 장애물 연습 프로그램은 if/else 문을 다른 if/else 문 내부에 작성하도록 했습니다. 왜냐하면 리보그가 3 가지 선택(끝내거나, 앞으로 움직이거나, 장애물을 넘든가)을 해야 하기 때문입니다. 이런 상황은 점점 더 코드를 들여 써야만 한다는 것을 알아차리셨을 것입니다. 리보그가 10 개의 상호 배타적인 선택을 한다면 무슨 일이 생길지 상상해 보세요. 프로그램 코드의 가독성은 떨어질 것입니다. 이런 상황을 타개하고자, 파이썬을 만든 Guido Van Rossum 은 if 절에 else 문과 조합을 표현하는 키워드, elif 를 만들었습니다. elif 는 여러분이 생각하듯이 else if 의 축약어입니다. 이 새로운 키워드로, 프로그램 코드는 다음과 같이 다시 작성될 수 있습니다.

```
def move_jump_or_finish():
    if next_to_beeper():
        turn_off()
    elif front_is_clear():
        move()
    else:
        jump_one_hurdle()
```

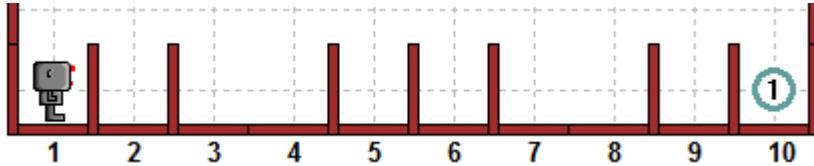
프로그램 코드가 동일한 방식으로 들여쓰기가 되어, 3 가지 가능한 경우의 수가 있다는 것을 이제 좀더 잘 볼 수 있습니다. else 조건은 앞의 조건이 모두 거짓인 경우에만 수행되어, else 와 관계된 다른 조건은 없습니다. 3 개 이상의 선택지를 필요하다면, 다른 elif 문을 추가하기만 하면 됩니다.

```
def move_jump_or_finish():
    if next_to_beeper():
        turn_off()
    elif front_is_clear():
        move()
    elif right_is_clear(): # 항상 거짓(False)
        pass
    else:
        jump_one_hurdle()
```

리보그가 바닥의 벽을 따라 움직여서, right_is_clear() 명령문은 항상 거짓이여서, pass 명령문(파이썬에서 아무것도 하지 않는 것과 동일)은 항상 무시됩니다. Left_is_clear() 명령문이 대신 사용한다면, 리보그는 첫 번째 장애물에 도달하자마자 영원히 움직이지 못하게 될 것입니다. **여러분이 직접 해보세요!**

여러분 차례

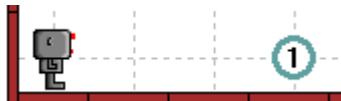
If, elif, else 키워드를 사용하여 앞 두 장애물 경주 코스 예제를 수행하는 프로그램과 다음 hurdles3.wld 파일의 경주 코스 예제도 수행하는 프로그램을 작성해 보세요.



◀ 제 말을 들으세요... 그렇지 않다면 - 🏠 - 참이 아닙니까? ➡

참이 아닙니까?

비퍼를 발견하고 꺼질 때까지 리보그가 계속 이동하길 원한다고 가정합시다. 출발 위치에서 사례는 아래와 같습니다.



의사 코드(pseudo code)로 작업을 수행하는 방법은 다음과 같을 것입니다.

```
If next to beeper,  
... stop;  
otherwise,  
... keep moving.
```

지금까지 공부한 것을 사용해서, 의사 코드를 아래처럼 번역할 수 있습니다.

```
if next_to_beeper():  
    turn_off()  
else:  
    move()
```

아직, 리보그에게 반복해서 진행하라고 명령하지 않아서 단지 프로그램의 일부입니다. 잠시 이것은 무시합시다. 대신에 이것에 상응하는 다른 의사 코드를 생각해 봅시다.

```
If not next to beeper,  
...keep moving;  
otherwise,  
...stop.
```

keep moving 과 stop 의 순서를 바꾸었지만, 여러분은 두 가지 방식이 동일하다는 것에 동의하실 것입니다. 핵심 내용은 두 번째 경우에 if 문 뒤에 부정 “not” 키워드의 사용입니다. 파이썬은 아래와 같이 테스트에서 부정을 사용할 수 있습니다.

```
if not next_to_beeper():  
    move()  
else:  
    turn_off()
```

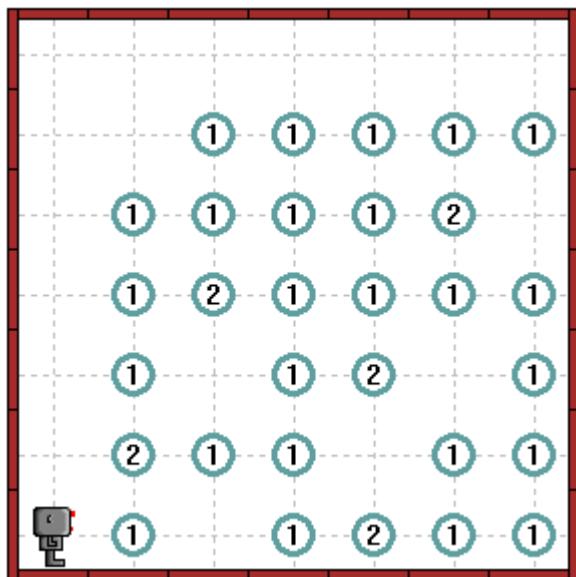
지금까지 여러분이 배운 것을 사용해서, 완전한 프로그램으로 바꿔봅시다. def 함수 명령문을 사용해서 명령문을 정의하고, 리보그가 작업을 완수할 수 있을 만큼 충분히 반복해야 합니다.

여러분 차례

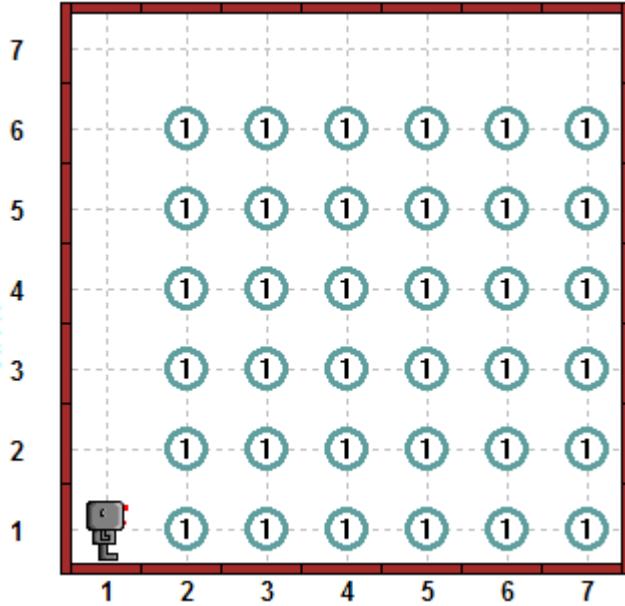
신규 파이썬 키워드 not 를 사용해서 장애물 넘기 프로그램을 다시 작성하세요.

김을 매고 씨 뿌리는 시기

봄이 왔습니다. 리보그의 아버지는 가을 추수를 위해서 정원에 씨앗을 심었습니다. 몇몇 곳에서 씨앗 두 개가 발아를 했고, 다른 곳에서는 발아를 하지 않았습니다. 전형적인 상황은 아래 보여지고 있습니다. (파일: harvest4.wld)



리보그가 정원에서 김을 매는 것을 도와주세요. 두 개의 당근(비퍼)이 있는 곳이 있거나, 당근이 하나도 없는 곳이 있으면 안 됩니다. 최종 상황은 아래와 같아야 합니다.



여기서 코드의 일부분에 사용될 수 있는 제안이 있습니다.

```
# 문제에 관련 어휘를 연결시키기
next_to_a_carrot = next_to_a_beeper
plant_carrot = put_beeper
pick_carrot = pick_beeper

def one_carrot_only():
    if not next_to_a_carrot():
        plant_carrot()    # 씨앗이 없는 것을 대체하기
    else:
        pick_carrot()
    if not next_to_a_carrot(): # 이런!(Oops!)
        plant_carrot() # 단지 하나만 제거해야 합니다.
```

현실에서 모든 씨앗을 제거하고 당장 다시 심는 것은 좋은 생각이 아닙니다.

if, else, if, else, ... - - while 문

While 문

어떤 조건이 만족될 때까지 명령문을 반복하기 원할 때, 파이썬에서 좀 더 간단한 방식으로 프로그램을 작성할 수 있는 새로운 키워드(while)를 사용할 수 있습니다. 앞에서 살펴본 예제를 의사 코드를 사용하여 이것이 어떤 모습인지 살펴봅시다.

While not next to beeper,
... keep moving;
otherwise,
... stop.

이것이 앞에서처럼 같은 생각을 표현한다는데 동의하실 것입니다. 파이썬 코드를 사용해서, 여기 어떻게 프로그램을 작성하는지 살펴봅시다.

```
while not next_to_a_beeper():
    move()
    turn_off()
```

반복을 할 필요가 전혀 없습니다. **시도해 보세요!**

여러분 차례

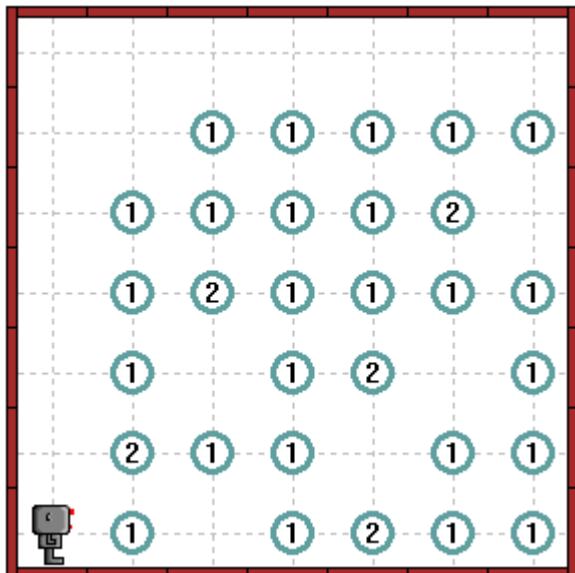
while 과 **not** 키워드를 사용해서 장애물 넘기 프로그램을 다시 작성하세요. 반복문의 임의의 숫자를 사용할 필요가 없습니다. 다른 말로, 프로그램의 핵심부분은 다음과 같을 것입니다.

```
while not next_to_a_beeper():
    move_or_jump()
    turn_off()
```

정상적으로 작동하는지 확인해보세요!

김을 매고 씨 뿌리는 시기

다시 봄이 돌아왔습니다. 리보그의 아버지는 가을 추수를 위해서 정원에 씨를 뿌렸습니다. 지난번과 마찬가지로 두 개의 씨가 발아한 곳이 있는 반면, 전혀 씨가 발아하지 않은 곳도 있습니다. 전형적인 상황은 아래 보여지고 있습니다.
(월드파일: harvest4.wld)



리보그가 정원에서 김을 매는 것을 도와주세요. 두 개의 당근(비퍼)이 있는 곳이 있거나, 당근이 하나도 없는 곳이 있으면 안됩니다.

여기 while 키워드를 사용한 프로그램 코드의 일부가 제안으로 있습니다.

```
# 문제에 관련 어휘를 연결시키기
next_to_a_carrot = next_to_a_beeper
plant_carrot = put_beeper
pick_carrot = pick_beeper

def one_carrot_only():
    while next_to_a_carrot():
        pick_carrot()      # 모두를 뽑아냅니다.

    plant_carrot()      # 하나만 다시 파종합니다.
```

여기 프로그램 코드의 일부분은 앞의 것보다 약간 짧습니다. (6 줄 대신에 def 함수 정의 명령문에 3 줄이 있습니다.) 더욱이, 한 장소에 2 개 이상의 씨앗이 발아한 경우에도 이 프로그램은 작동이 됩니다. 시도해 보세요~

현실에서 모든 씨앗을 제거하고 당장 다시 심는 것은 좋은 생각이 아닙니다.



놀라운 Part 1

여기 부분(Multi-part) 수업에서는 증가하는 복잡성을 가진 문제를 해결하는 완벽한 프로그램을 작성할 것입니다.

1. 놀라운 솔루션: 첫 걸음

간단한 프로그램으로 출발합시다. 리보그가 월드 주위를 한 바퀴를 도는데, 중간에 어떠한 장애물도 없다고 가정합시다. `front_is_clear()` 테스트를 소개했을 때, 전에 동일한 프로그램을 작성했습니다. 여기 출발점에서 비퍼 하나를 가지고 다니는 리버그 프로그램의 대략적인 개요가 있습니다.

- I. 출발점에서 비퍼를 내려 놓습니다.
- II. 벽을 마주칠 때까지 계속 전진합니다.
- III. 벽에 마주칠 때 왼쪽으로 회전을 합니다.
- IV. 내려놓은 비터를 발견할 때까지 II, III 번 단계를 반복합니다.
- V. 비퍼를 발견하면 리보그가 꺼집니다.

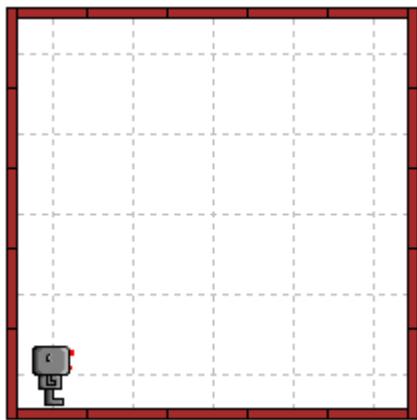
중요한 프로그램의 단계는 IV로 반복 명령문이 있습니다. 반복 명령문은 아래와 같이 작성됩니다.

`while not next_to_a_beeper():`

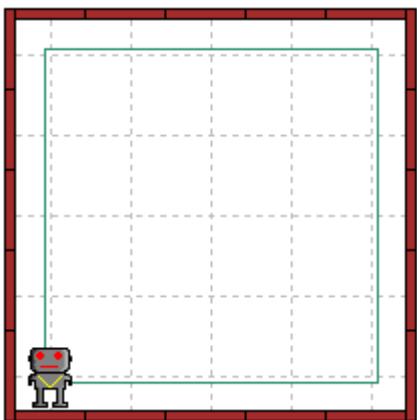
그 다음으로 중요한 프로그램의 단계는 II, III가 됩니다. 글로 작성한 코드를 온전한 프로그램 코드로 전환하여 봅시다.

```
put_beeper()  
while not next_to_a_beeper():  
    if front_is_clear():  
        move()  
    else:  
        turn_left()  
  
turn_off()
```

만약 출발 위치가 아래 보여진 것과 같이 되어 있다면, 시간을 가지고 위 프로그램이 리보그에게 무슨 일을 명령하는지 생각해 봅시다.



아래의 경우는 원하는 결과를 얻지 못합니다. 왜 그런지 설명할 수 있나요? 만약 설명이 되지 않는다면, 뒤로 돌아가서, 다시 생각해 보세요.



◀ while 문 - - 놀라운 Part 2 ▶

놀라운 Part 2

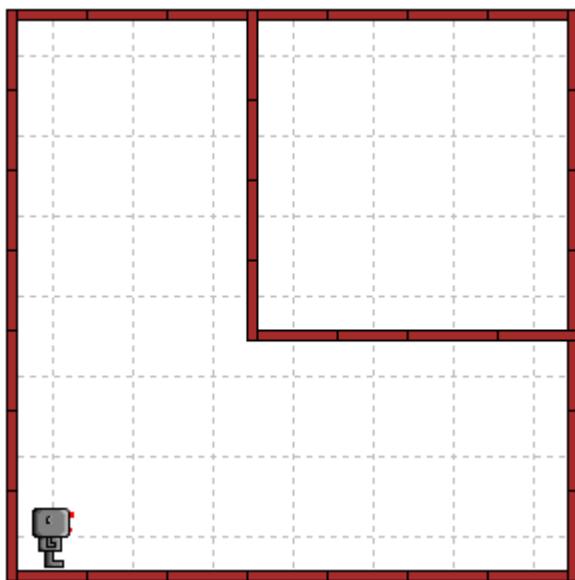
Part 1 의 마지막에 우리가 직면한 문제의 원인이 여기 있습니다. 비퍼를 내려놓습니다. 앞으로 이동하기 전에 비퍼가 옆에 있지 않은지를 확인하는 테스트를 합니다. while 반복 명령문에 들어갈 기회가 전혀 없습니다. 반복 명령문을 실행하기 전에 move() 명령문을 아래와 같이 추가하여 수정합니다.

```
put_beeper()  
move()  
while not next_to_a_beeper():  
    if front_is_clear():  
        move()  
    else:  
        turn_left()  
  
turn_off()
```

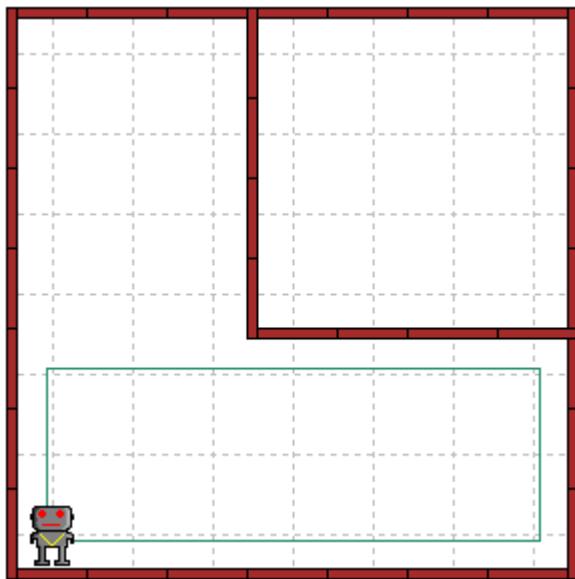
성공! 시도해 보세요.

2. 그다지 간단하지 않은 월드

작성한 프로그램을 조금 더 복잡한 월드에서 실행해 봅시다. 아래 월드는 쉽게 다시 만들 수 있는 월드입니다.



결과는 우리가 원하는 것은 정확히 아닙니다. 리보그가 지름길을 택해서, 벽을 따라 움직이지는 않았습니다.



문제는 리보그가 벽을 따라 움직일 때 단지 앞으로 전진만 하거나, 왼쪽으로만 회전하는 것을 가정했습니다. 리보그가 오른쪽으로 회전하는 상황에 대해서는 고려를 전혀 하지 않았습니다. 리보그에게 필요한 것은 첫째 오른쪽에 여전히 벽이 있는지를 확인하는 것입니다. 만약 벽이 없다면, 리보그가 오른쪽으로 회전을 해야만 합니다. 오른쪽으로 회전하는 수정한 버전의 프로그램이 있습니다.

```
def turn_right():
    repeat(turn_left, 3)
    put_beeper()
    move()
    While not next_to_a_beeper():
        if right_is_clear():
            turn_right()
        elif front_is_clear():
            move()
        else:
            turn_left()
turn_off()
```

정상적으로 잘 동작합니까? 결정을 내리기 위해서 주의 깊게 읽어보세요. 여러분의 의견을 확인하기 위해 직접 시도해 보세요.



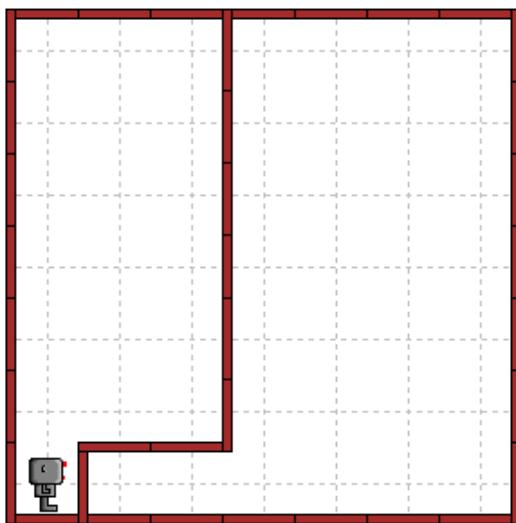
놀라운 Part 3

알아채셨겠지만, 프로그램은 작동을 하지 않습니다. 리보그 주변에 벽이 없는 상태에서 리보그가 무한 반복상태에 놓여지게 됩니다. 아래와 같이 오른쪽을 회전을 한 후에 리보그가 move() 명령문으로 한 칸 앞으로 이동해야 합니다.

```
def turn_right():
    repeat(turn_left, 3)
    put_beeper()
    move()
    While not next_to_a_beeper():
        If right_is_clear():
            turn_right()
            move()
        elif front_is_clear():
            move()
        else:
            turn_left()
    turn_off()
```

3. 좀더 복잡한 월드

이제 아래 보여지는 월드를 생각해봅시다. 여러분이 작성한 프로그램이 동작할까요?



불행하게도, 정답은 ‘아니오’입니다. 더 이상의 학습을 진행하기 전에 왜 그런지 알아봅시다.

◀ 놀라운 Part 2 -  놀라운 Part 4 ▶

놀라운 Part 4

여러분 대부분 알아내셨듯이, 비퍼를 내려 놓은 후에 너무나도 서둘러서 리보그에게 앞으로 전진하게 명령을 한 것입니다. 앞으로 움직이기 전에 벽이 있는지를 확인하는 단계가 필요합니다. 여기 해답이 있습니다.

```
def turn_right():
    repeat(turn_left, 3)
put_beeper()
# 프로그램 수정 시작 지점
if not front_is_clear():
    turn_left()
# 프로그램 수정 끝 지점
move()
while not next_to_a_beeper():
    if right_is_clear():
        turn_right()
        move()
    elif front_is_clear():
        move()
    else:
        turn_left()
turn_off()
```

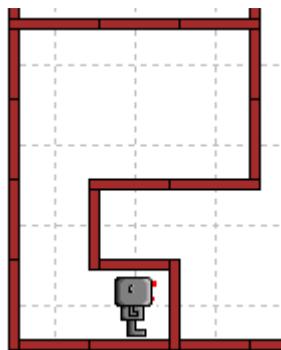
지금 당장 테스트해보고 정상적으로 작동하는지 확인하세요. 이 프로그램이 동작할 것 같지 않는 상황을 상상할 수 있나요?



놀라운 Part 5

5. 좀더 복잡함

다음 월드를 생각해봅시다.



앞서 작성한 프로그램이 작동합니까? 시도해 보세요.

아마도 생각하듯이, 이 프로그램은 작동하지 않습니다. 정상적으로 작동시키기 위해서, 방금 전에 추가한 while 명령문을 if 명령문으로 바꿀 필요가 있습니다. 시도해 보세요! 저장하는 것을 확인하세요.

6. 의도를 명확히 하기

리보그가 마주칠 것 같은 거의 모든 상황에서 작동하는 프로그램을 작성한 듯 보입니다. 이 프로그램은 리보그가 벽을 따라 월드를 한번 탐색하도록 작성되었습니다. 프로그램이 다소 짧고 구조가 깔끔하게 보일 수 있지만, 처음으로 프로그램을 읽는 사람에게는 명확하게 보이지 않을 수 있습니다. 주석을 추가하든지 좀더 의미가 있는 단어를 사용하는 것이 좋은 생각입니다. 생각하는 것 보다 다소 단어나 말이 길어지는 것처럼 보일 수 있지만, 주석을 추가해 봅시다.

```
# 유용한 명령문 함수를 정의합니다.  
def turn_right():  
    repeat(turn_left, 3)  
# 비퍼를 내려놓아서 시작 지점을 표시합니다.  
put_beeper()  
# 그리고 나서, 나아갈 방향을 찾고, 앞으로 움직여 갑니다.  
while not front_is_clear():  
    turn_left()  
move()
```

```

# 비퍼를 내려 놓은 장소로 돌아왔을 때,
# 월드를 벽을 따라 한 바퀴 돈 것을 알 수 있습니다.
While not next_to_a_beeper():
    if right_is_clear(): # keep to the right
        turn_right()
        move()

    elif front_is_clear(): # 오른쪽 벽을 따라 움직입니다.
        move()

    else: # 왼쪽 벽을 따라 움직입니다.
        turn_left()
turn_off()

```

이 방법이 각 명령문에 대한 의도를 명확히 하지만, 문제를 해결하는데 사용되는 알고리즘(Algorithm)으로 알려진 방법을 요약하는 데는 그다지 도움이 되지 않습니다. 따라서, 이렇게 주석을 다는 것은 여러분이 희망하는 다른 프로그램을 읽는 분들에게는 도움이 되지 않을지도 모릅니다. 주석을 읽으면, 작성된 프로그램이 두 개의 부분으로 구성된 것을 주목바랍니다.

- 1) 첫 시작 지점을 표시합니다.
- 2) 처음 시작 지점으로 돌아올 때까지 오른쪽 벽을 따라 계속 이동합니다.

프로그램을 다시 작성해서, 두 부분이 좀더 명확해 지도록 주석을 다르게 작성합시다.

```

# 이 프로그램은 리보그가 반시계 방향으로 벽을 따라 돌아
# 처음 시작한 지점으로 다시 돌아와서 멈추는 프로그램입니다.
def turn_right():
    repeat(turn_left, 3)

def mark_starting_point_and_move():
    put_beeper()
    while not front_is_clear():
        turn_left()
        move()

def follow_right_wall():
    if right_is_clear():
        turn_right()
        move()
    elif front_is_clear():
        move()
    else:
        turn_left()

found_starting_point = next_to_a_beeper # 비퍼가 초기 시작지점을 표시합니다.

```

```
==== 함수 명령문 정의 끝, 솔루션(해법) 시작
```

```
mark_starting_point_and_move()
```

```
While not found_starting_point():
```

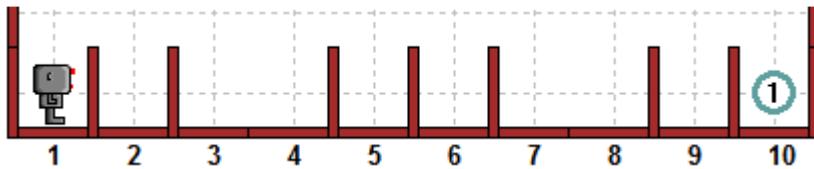
```
    follow_right_wall()
```

```
turn_off()
```

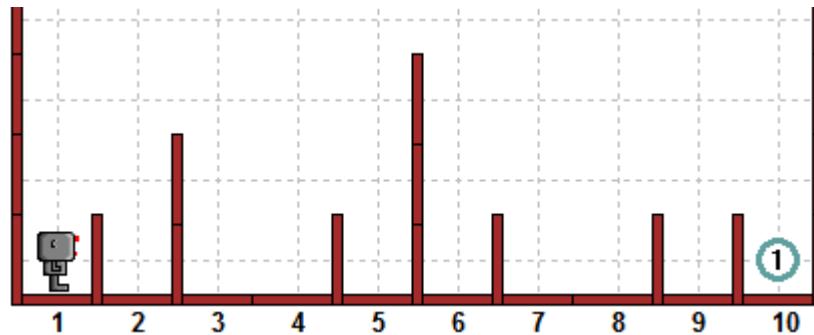
이것이 좀더 명확하지 않습니까? 이번에는 각 모퉁이의 시작 지점에 비퍼가 있다고 가정합시다. 그 다음, 비퍼를 제거하는 것을 통해서 시작 지점을 선택할 수 있습니다. 함수 명령문 정의 부분을 약간 수정할 필요가 있지만, 솔루션(해법) 부분을 수정할 필요는 없습니다.

7. 첫 번째 기적

방금 전에 작성한 프로그램을 다음 장애물 경주 (파일: hurdles3.wld)에 저장하여 실행하여 보세요.



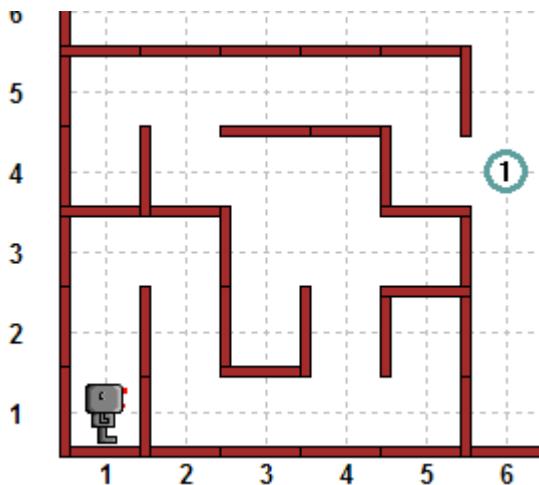
기적! 특이한 방향으로 결승선을 통과하고 결승선에서 우승한 후에 관람객에게 인사하는 것을 제외하고, 방금 전에 작성한 프로그램은 장애물 경주 문제도 해결할 수 있습니다. 이 프로그램은 아래 그림의 울퉁불퉁한 장애물 경주(파일: hurdles3.wld)에도 작동합니다. 장애물을 넘기 위해서 전에 작성한 프로그램은 해결할 수 없었던 것입니다.



이 프로그램의 경우 mark_starting_point_and_move()도 필요하지 않다는 것을 알아 채셨을 것입니다. 프로그램을 깔끔하게 구조화했기 때문에, 명령 함수문을 제거하고, 좀더 서술적인 다른 이름(racing_hurdles 등) 저장하기가 쉬울 것입니다.

8. 놀라운 기적

앞에서 소개한 아래 보여지는 미로 문제에 동일한 프로그램을 실행하여 봅시다.
(월드파일: maze1.wld)



실행해 보면, 여러분이 작성한 간단한 프로그램은 미로도 탈출할 수 있습니다.
놀랍죠! 다시 한번, 여러분이 작성한 프로그램은 불필요한 명령문 함수가 포함되어 있음을 주목하세요.

9. 결론

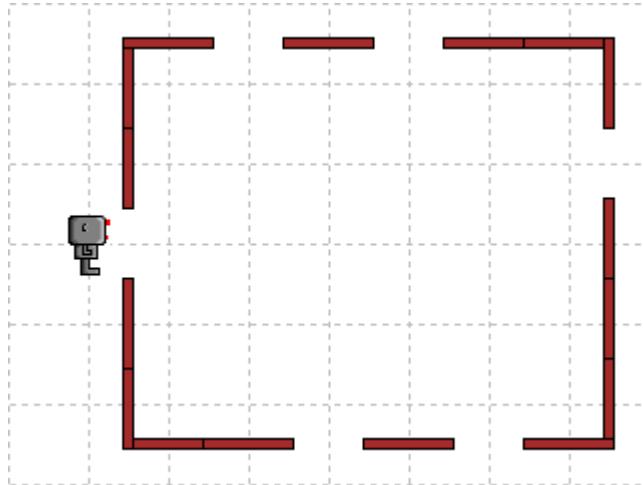
사각형 월드의 벽을 따라 움직이는 간단한 문제를 해결하는 것에서 시작해서 조금씩 조금씩 프로그램을 향상시키는 점진적 상세화(stepwise refinement)를 통해서 많은 연관되지 않을 것 같은 문제를 해결하는데 사용할 수 있는 프로그램을 작성했습니다. 매 단계마다 수정사항을 최소화하고, 좀더 복잡한 문제를 고려하기 전에 제대로 작동하는 솔루션 프로그램을 개발했습니다. 또한, 서술적인 이름을 알고리즘의 부분에 부여해서 읽기 쉽고, 이해하기 좋은 프로그램을 작성했습니다. 이것이 여러분이 자신만의 프로그램을 작성할 때 사용할 전략입니다.

- 간단하고, 작게 시작합니다.
- 한번에 작은 수정사항, 변경을 합니다.
- 각 단계의 수정 변경 사항이 전에 작성한 것과 잘 작동하는지 확인하세요.
- 각 명령문이 수행하는 것을 단순 반복하지 않는 적절한 주석을 추가하세요.
- 서술적인 이름을 사용하세요.

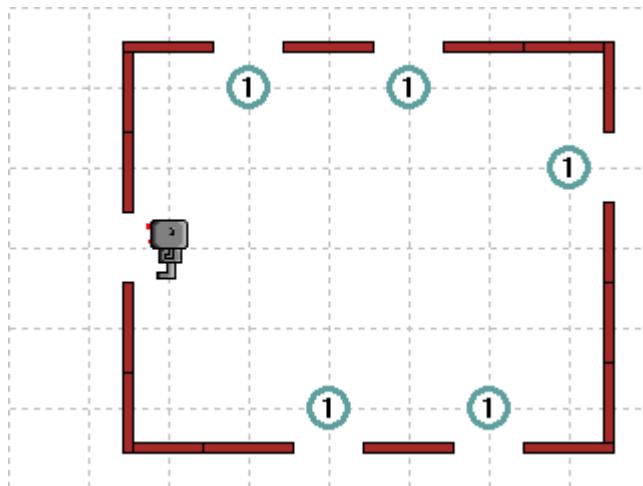
◀ 놀라운 Part 4 -  - 비가 와요 ▶

비가 와요

아름답도록 햇살이 가득한 날입니다. 리보그가 친구와 밖에서 놀고 있어요. 갑자기, 비가 내렸는데 리보그가 집의 모든 창문을 열어 놓고 나온 것을 기억해 냈습니다. 그래서 리보그는 황급히 집으로 돌아와서 집 앞 문턱에 멈췄는데, 어떻게 창문을 닫아야 할지 확신할 수 없었습니다.



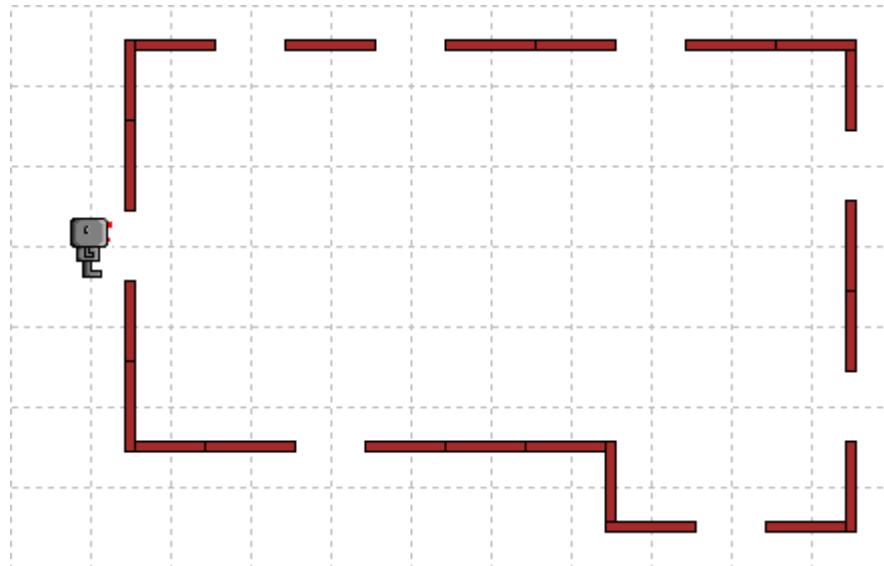
리보그가 집의 모든 창문을 닫도록 도와주세요. 리보그 월드에서 닫힌 창문은 창 앞에 비퍼가 하나 놓여 있습니다. 리보그가 작업을 마쳤을 때, 다시 문 앞에 서서 다시 돌아가서 밖에서 놀기 전에 비가 그치기를 기다리면서, 비 내리는 것을 바라보고 있을 것입니다.



월드 파일은 ‘rain1.wld’ 파일이며, 리보그는 작업을 수행하기 위해 충분한 비퍼를 가지고 다닙니다.

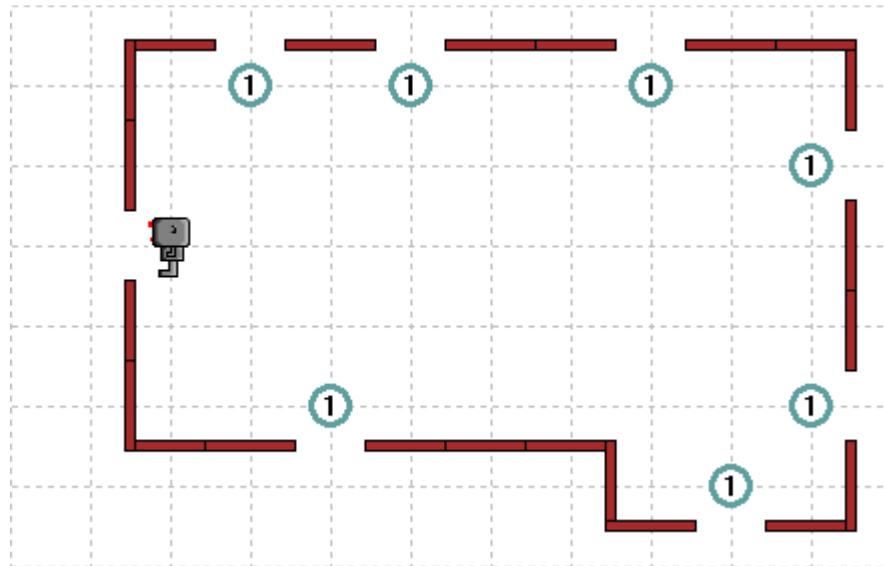
리보그 친구 차례

리보그 친구 Erdna 는 좀더 큰 집에 살고 있습니다. 비가 오기 시작했을 때, Erdna 는 리보그와 함께 밖에서 놀고 있었습니다. Erdna 가 집의 창문을 모두 닫도록 도와주세요. 시작 위치는 다음과 같습니다.



위 문제에 해당하는 월드 파일은 rain2.wld 입니다.

Erdna 가 모든 창문을 닫았을 때, 다음에 보여주듯이 집 문 앞에 서 있을 것입니다.



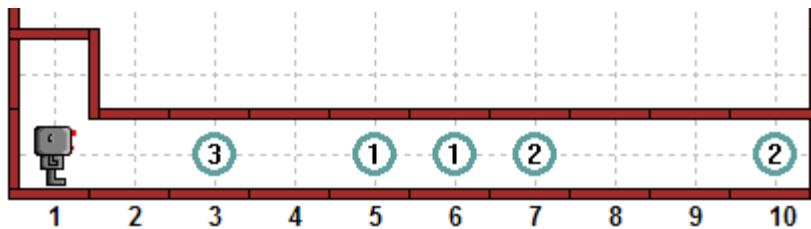
프로그램 한 개로 충분해요.

Erdna 를 도와서 창문을 닫는 프로그램이 동일하게 리보그의 집의 창문을 닫는데도 사용될 수 있는지 검증해 보세요. 만약 사용할 수 없다면, 프로그램을 수정해서 Erdna 와 리보그 집의 창문을 모두 닫는데 사용될 수 있도록 해보세요.

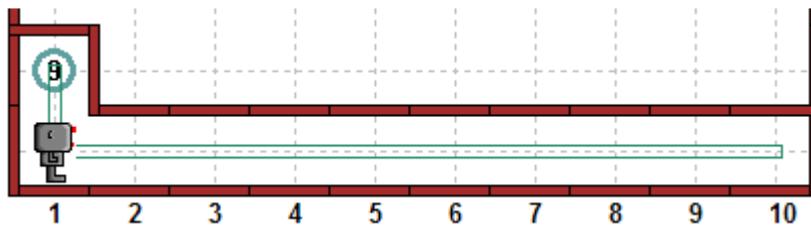
◀ 놀라운 Part 5 -  - 폭풍이 지나간 후에 ▶

폭풍이 지나간 후에

지난밤에 바람이 정말 심하게 불었습니다. 리보그 집 밖에 쓰레기가 여기저기 널려 있습니다. 리보그 부모님이 리보그에게 차도뿐만 아니라 장외시장으로 가는 길도 말끔하게 청소하라고 했습니다. 두 길 모두 직선이지만 다음에 보여지듯이 쓰레기가 아무 곳에나 널려있습니다.



리보그는 모든 쓰레기를 모아서 시작 지점의 북쪽에 있는 쓰레기통에 담아 놓아야 합니다. 최종 상황은 다음처럼 보일 것입니다.

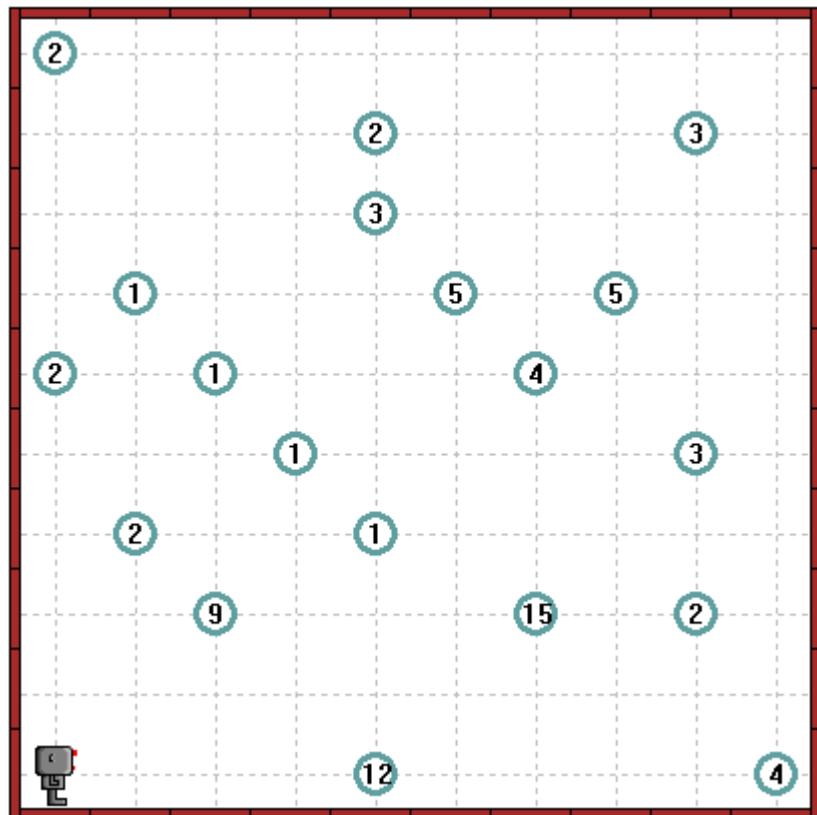


여러분이 작성한 프로그램이 두 가지 상황(월드 파일: rash1.wld, trash2.wld)에 모두 작동하는가를 확인하세요.

마당 청소 좀더 하기

리보그의 부모님은 아들의 처리한 일을 자랑스러워 하며, 지난 밤 폭풍에 날아간 뒷마당 쓰레기를 모두 주워 담으라고 했습니다. 리보그는 모든 쓰레기를 주워서 처음 위치로 가져다 놓아야 합니다. 길가 청소를 위해서 작성한 프로그램을 일반화해 보세요.

다음에 보여지는 상황에 상응하는 자신만의 월드 파일을 만들어 보세요. 여러분이 작성한 프로그램은 쓰레기가 위치한 정확한 장소에 의존하거나, 마당 크기에 좌우되어서는 안됩니다.



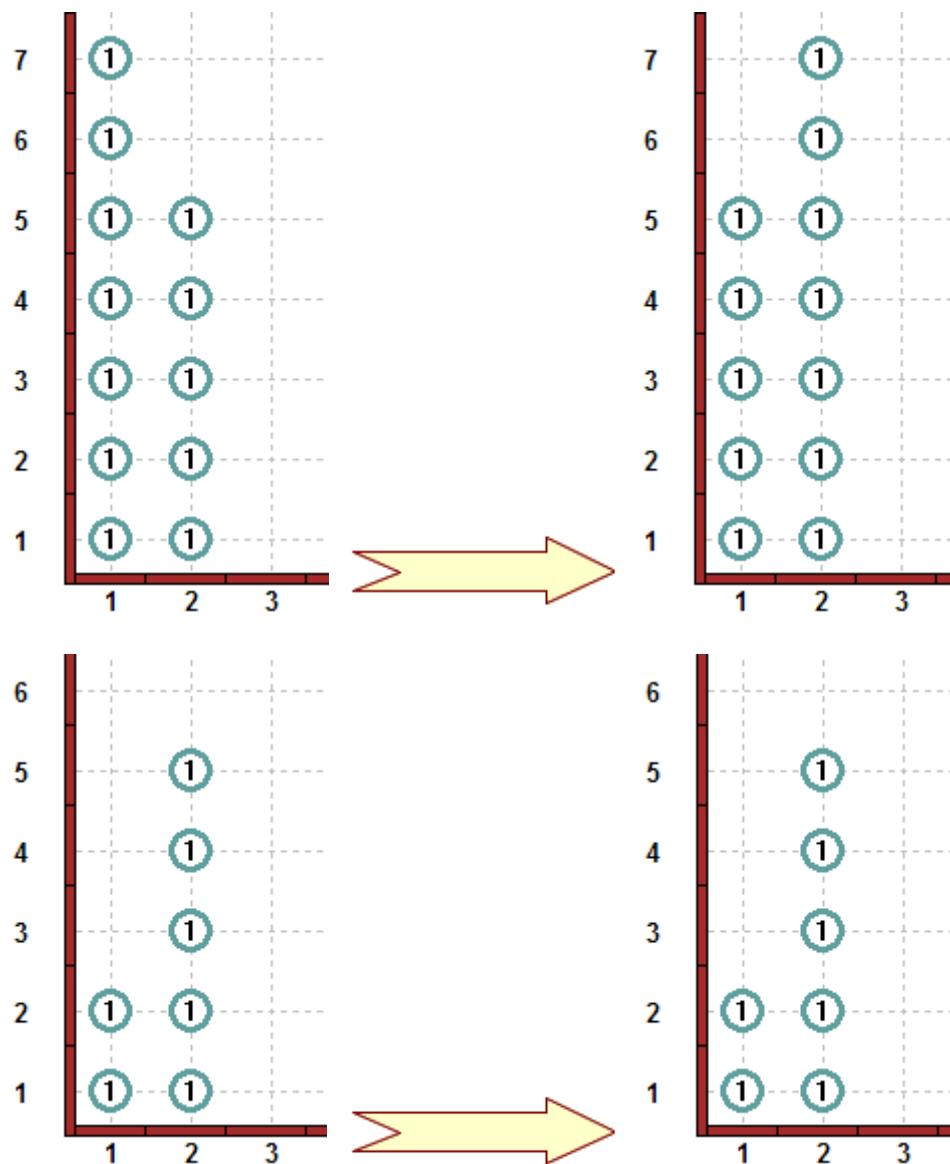
◀ 비가 와요 - 🏠 - 정렬하기 ▶

정렬하기

정렬은 큰 프로그램을 작성할 때 종종 필요한 것으로 오름차순 혹은 내림차순으로 개체의 집합을 순서대로 내려놓는 것입니다. 이번 학습에서, 리보그에게 몇 가지 정렬 기술을 가르칠 것입니다.

두 열 정렬하기

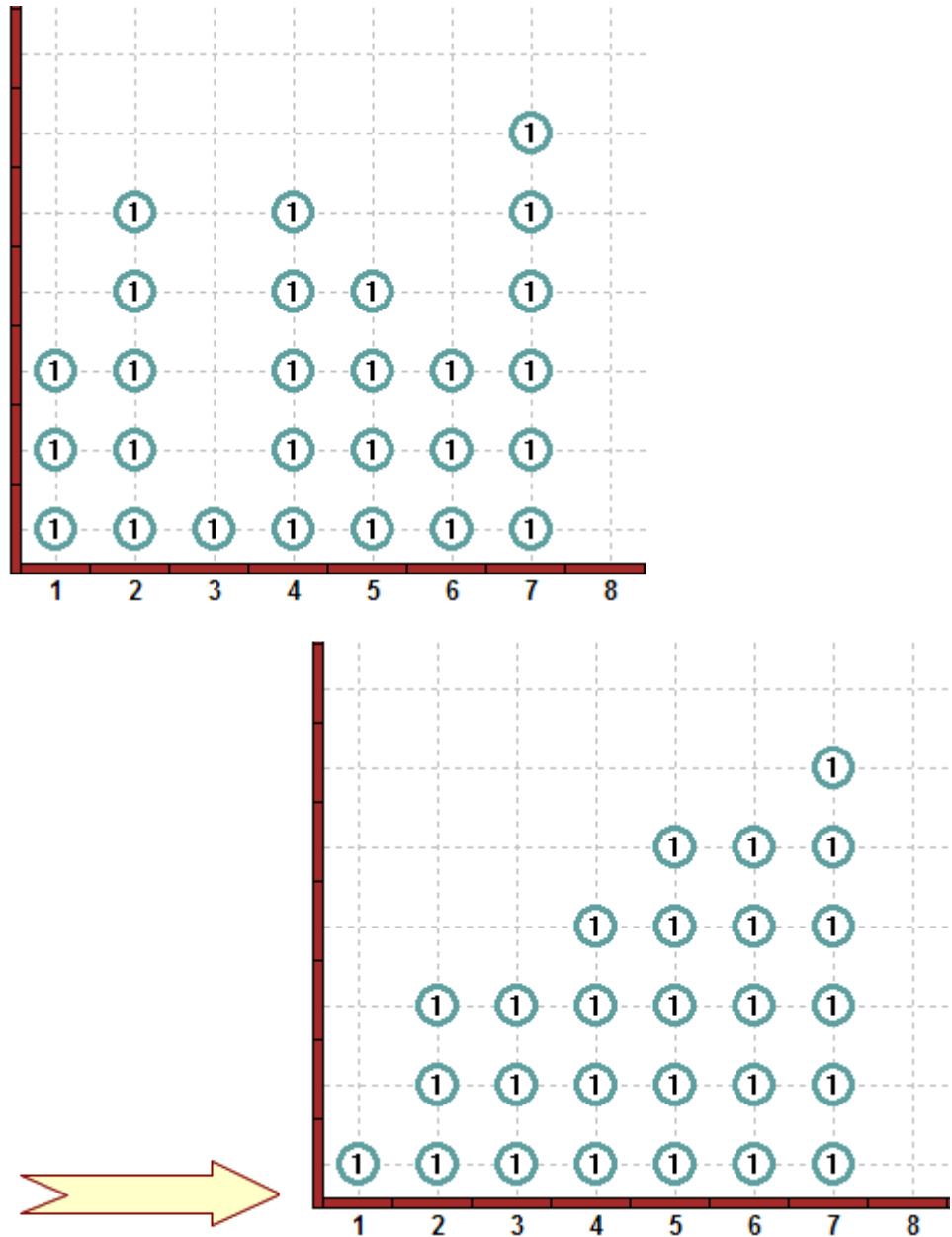
워밍업 연습으로, 두 줄로 쌓여 있는 비퍼를 순서대로 정렬하는 프로그램을 작성하세요. 여기 두가지 전형적인 상황이 있습니다.



내림차순으로 비퍼 더미를 내림차순으로 정렬하도록 프로그램을 수정하는 것은 얼마나 어렵나요?

많은 열을 정렬하기

다음 연습문제는 다소 도전이 될 수 있습니다. 프로그램을 작성해서 리보그가 파악이 되지 않는 모든 열의 비퍼들을 오름차순으로 정렬하세요. 첫 빈 비퍼들은 정렬할 필요가 있는 마지막 비퍼들을 의미합니다. 여기 일반적인 예제가 있습니다.



내림차순으로 비퍼들을 내림차순으로 정렬하도록 프로그램을 수정하는 것은 얼마나 어렵나요?

◀ 폭풍이 지나간 후에 -  - 반복 피하기 – 중요한 것 ▶

반복 피하기 - 가져오는 것

반복을 회피하는 규칙을 기억하세요.

규칙 3.

프로그램을 작성할 때, 자신을 반복하지 마세요.

다시 말씀드립니다. **자신을 반복하지 마세요.**

여기서 학습할 내용은 “바퀴를 재발명하는 것(reinventing the wheel)”을 회피하는 방법입니다.

1. 3 번의 좌회전은 한번의 우회전을 기억하세요.

turn_right() 명령문을 기억해 보세요.

```
def turn_right():
    turn_left()
    turn_left()
    turn_left()
```

이 명령문 함수를 작성한 것은 리보그가 우회전을 할 때마다 3 번 turn_left() 명령어를 타이핑하는 것을 회피하기 위해서입니다. 하지만, 여전히 반복이 많습니다. 왜냐하면, 거의 매번 새로운 프로그램을 작성할 때마다, turn_right() 명령문 함수를 정의해야 하기 때문입니다. 하지만, 이러한 반복을 회피하는 방법이 있습니다. 다음 프로그램을 작성해서 실행해 보세요.

```
from useful import turn_right
```

```
turn_right()
turn_off()
```

2. 유용한 것에 관해서 (About useful)

새로 정의한 명령문 함수를 포함하는 “useful”이라는 모듈을 여러분을 위해 제가 만들었습니다. 이 새로운 명령문이 수행하는 것을 설명하기보다, 다음 화면에 상응하는 간단한 프로그램이 있습니다.

```
from useful import *
```

```
turn_left()
move()
set_trace_style(1, 'red')
turn_right()
set_trace_style(2, 'blue')
move()
set_trace_style(1, 'green')
climb_up_east()
set_trace_style(2, 'blue')
move()
set_trace_style(1, 'orange')
turn_around()
set_trace_style(2, 'sea green')
climb_up_west()
set_trace_style(1, 'red')
turn_around()
set_trace_style(0, 'blue') # 0 = invisible
repeat(move, 4)
set_trace_style(1, 'green')
climb_down_east()
set_trace_style(2, 'blue')
move()
set_trace_style(1, 'black')
turn_around()
set_trace_style(2, 'red')
climb_down_west()
turn_off()
```



첫 번째 `turn_right()` 예제에서 했던 것처럼 모든 새로운 명령문들을 하나씩
하나씩 가져오는(`import`) 대신에 모든 명령 함수 정의문을 의미하는 ‘*’
표기를 사용합니다. 좀더 앞으로 나가기 전에 몇 가지 코멘트를 드립니다.

- I. 파이썬 모듈을 아시는 분들에게, “useful”이 실제 모듈이 아니라 리보그 월드에서만 작동하는 가상의 모듈입니다. 여러분은 실제

모듈을 가져와서 러플(RUR-PLE) 프로그램상에서 사용할 수 없습니다. 이유는 안전과 관련된 사유 때문에 그렇습니다.

- II. 새로운 명령문 함수 `set_trace_style()`가 사용되었습니다. 일반적인 프로그래밍 전문용어가 아니어서, 리보그 월드에서 좀더 의미 있는 `set_leaking_oil_colour_and_quantity()`로 작명을 하자는 못했습니다. 크게 말씀 드려, 프로그램 `trace` 는 프로그램이 수행될 때 일련의 명령문을 따라갑니다. 일반적으로 이런 명령문은 특별한 쓸모는 없지만, 버그를 추적하는 데나 지금 사용된 사례에서는 유용합니다.
- III. `from` 과 `import` 는 프로그램을 작성 시 부여되는 색깔로 알 수 있듯이 파이썬 키워드입니다.
- IV. “useful” 팩키지에 정의된 몇몇 명령문 함수들은 로봇이 시작하는 예상 방향으로 위치하지 않는다면, 예기치 못한 결과를 가져올 수 있습니다. 실제 파이썬 모듈을 가져올 때 비슷한 일이 종종 생길 수 있습니다. 모듈 적절하게 사용하기 위한 사전 점검사항에 대해서 충분히 인지해야 합니다. “useful” 모듈에 대해서 이러한 사전 점검사항에 대해서는 말하지 않을 것입니다. 스스로 몇 개의 프로그램을 작성하다 보면 알게 될 것입니다.

3. 동일한 이름을 사용하고자 한다면

“useful” 내부에 정의된 것과는 별개로 다른 의미를 가진 “turn_right” 이름을 가진 명령문 함수를 정의하고 있다고 가정합시다. 이를 위해서 몇 가지 방법이 있습니다.

- I. `import useful` 문을 사용할 수 있습니다. 이를 위해서, `useful` 을 각 명령문 앞에 추가할 필요가 있습니다. 예를 들어,
`useful.turn_right()`, `turn_right()` 은 각각 다른 명령문입니다. 첫 번째는 `useful` 모듈 내부에 정의된 것이고, 두 번째는 다른 곳에서 여러분이 정의한 명령문 함수입니다. `useful.`을 각 명령문 앞에 붙여서, 독자는 이 명령문이 정의된 된 것을 바로 알 수 있습니다.

II. 비슷한 것을 수행하는 두 번째 방법은 다음과 같습니다.

```
# 다음은 불어 사용자에게 유용합니다.  
from useful import turn_right as vire_a_droite  
  
# 아래와 같이 사용하세요.  
vire_a_droite()  
turn_off()
```

as 단어는 거의 파이썬 키워드입니다. 파이썬의 다음 버전에서는 키워드가 될 것입니다. 변수(variable)을 논의할 때 이 스테이트먼트(statement)가 의미하는 바를 설명할 것입니다.

4. 아마도 결국 그렇게 유용하지 않을 듯

자신만의 프로그램을 작성할 때, “useful”모듈의 진정한 유용성을 생각한다면, 결국 너무나도 제한되어 있어서 유용하게 사용하지 못할 것이라 생각할 수 있습니다. 다음 프로그램을 작성하는 것이 이를 반영할 수 있습니다.

```
# 모듈 가져오기  
import useful as not_so_useful_after_all
```

```
# 다양한 명령문 함수 사용하기  
not_so_useful_after_all.turn_right()  
not_so_useful_after_all.turn_around()  
turn_off()
```

실제에서, import 스테이트먼트의 정상적인 사용의 모습은 다음 줄을 더 잘 보여줍니다.

```
# 모듈 가져오기  
import module_with_very_very_very_long_name as short_name
```

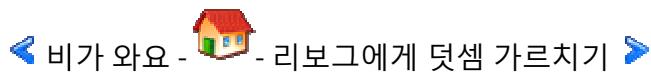
```
# 다양한 명령문 함수 사용하기  
short_name.instruction1()  
short_name.instruction2()
```

5. 왜 Import 를?

파이썬은 배터리가 포함돼서 온다고들 합니다. 즉, 파이썬은 매우 똑똑한 프로그래머들이 작성한 많은 유용한 모듈을 포함하고 있습니다. 이들 모듈은 버그가 없고 매우 효율적으로 작동하도록 철저하게 테스트가 되었습니다. 파이썬에 대해서 더 많이 배우게 되고, 자신만의 좀더 복잡한

프로그램을 작성 하다면, 이미 관련된 모듈이 존재하는지, 도움이 될 수 있는 모듈을 무엇인지를 먼저 확인할 필요가 정말 있습니다. 다음 학습에 몇 가지 유용한 모듈을 소개드릴 것입니다.

선생님께 중요한 노트: 위에 언급되었듯이, import 스테이트먼트 사용을 제한하는 이유는 누군가 안전하지 못한 스크립트를 가지고, 이를 사용해서 희생자가 되는 것을 피하기 위함입니다. 파이썬 편집 창에서 가능하다는 사실을 주지하십시오. 학생들도 실제 실행하기 전에 친구에게서 받은 프로그램을 확인하도록 주지시키십시오. 임의의 import 스테이트먼트를 가져오도록 러플(RUR-PLE)에서 작은 수정사항으로 매우 쉽게 할 수 있습니다. 수업시간에 유용할 수도 있습니다. 만약 필요하시다면, 스스로 방법을 찾지 못하시다면, 저자에게 연락을 주십시오.

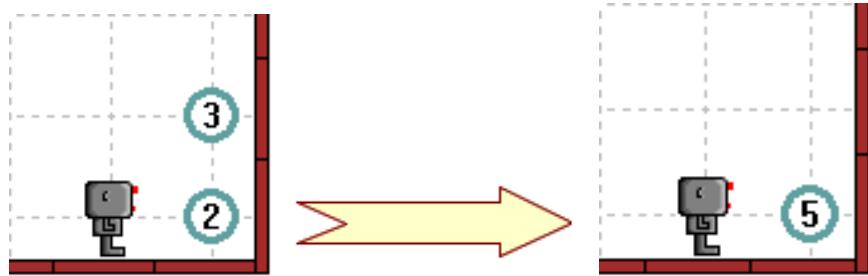


리보그에게 덧셈 가르치기

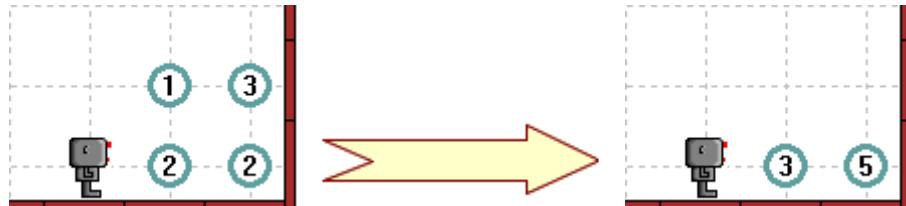
이번 학습에서, 리보그가 두 숫자를 더하는 프로그램을 작성하도록 여러분을 인도하겠습니다. 10 진수로 일반적인 방식으로 덧셈을 하지만, 다른 진법의 숫자를 이 프로그램을 사용하여 쉽게 할 수 있습니다.

1. 워밍업

- I. 다음 방식으로 리보그가 $3+2$ 로 더하기 하는 프로그램을 작성하세요.

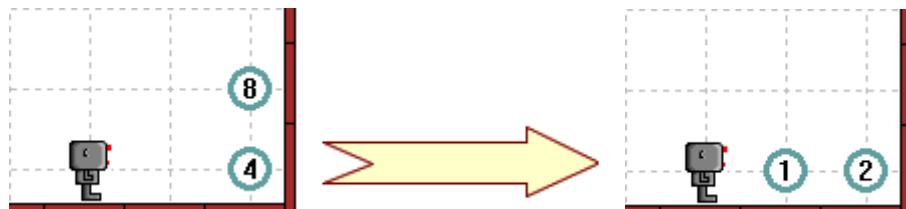


- II. 각 비피들은 숫자를 표현하는데, 리보그가 $13+22$ 를 다음 방식으로 더하도록 프로그램을 수정하세요.



여러분이 작성한 새로운 프로그램을 사용해서, 리보그가 $3+2$ 더하기를 할 수 있는 것을 확인하세요.

- III. 여러분이 작성한 프로그램을 사용하여, 리보그는 다음과 같이 $8+4$ 더하기를 할 수 있을까요?



더하기를 수행하도록, 프로그램을 수정하는 방법을 생각해 낼 수 있습니까? 더 읽기 전에 스스로 프로그램을 만들어서 시도해 보세요.

2. 덧셈 검토

전통적인 방식으로 오른쪽에서부터 왼쪽으로 두 숫자를 더해봅시다.

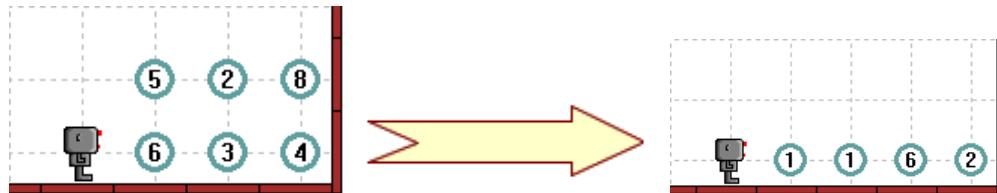
$$\begin{array}{r} 528 \\ + 634 \\ \hline \end{array}$$

12 # 마지막 숫자 두 개를 더하기 (8+4)

단자리에서 십자리 이월 시킬 1을 가지고 있습니다. 이월은 여러분이 작성한 프로그램이 문제를 가질 소지가 가장 큰 곳입니다. 일반적인 방식으로 다시 프로그램을 다시 작성해서 계속 갑시다.

$$\begin{array}{r} 1 \\ 528 \\ + 634 \\ \hline 1162 \end{array}$$

그럼, 조금 간략하지만, 여러분이 충분히 따라올 수 있다고 확신합니다. 리보그 월드에서, 덧셈을 아래 보이는 것처럼 보입니다.

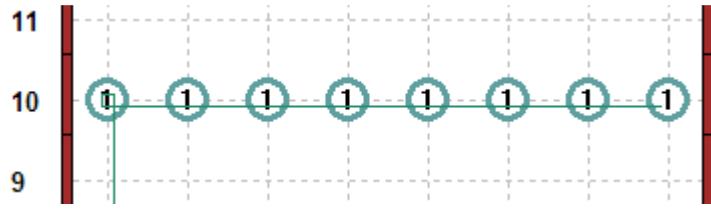


8+4 더하는 좀더 간단한 문제를 먼저 다뤄 봅시다.

3. 10 진수 8+4 더하기

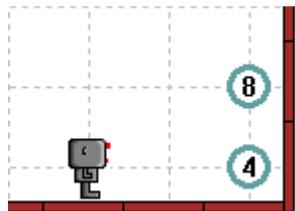
앞에서 언급했듯이, 각 비퍼들은 숫자를 나타내고 두 숫자를 더해서 합이 9보다 큰 경우 (십진수에서) 두 숫자를 더하기의 문제가 발생합니다. 두 숫자를 무엇을 더하든지 상관없이 이 마술 숫자(10)을 기억할 필요가 있습니다. 월드 파일(파일: adding_world.wld)을 생성했는데 10 진수(혹은 29 진수도 가능)로 7 자리까지 충분히 더할 수 있는 큰 월드입니다. 월드 파일을 살펴보세요. 적절하게 덧셈을 할 수 있는 프로그램을 작성할 수 있도록 제가 여러분을 인도하여 드리겠습니다.

월드 파일을 메모리에 올린 후에, 화면의 바닥을 살펴보면, 리보그가 8 개 비퍼를 가지고 있는 것을 알 수 있습니다. 다음에 보여지듯이 리보그가 10 번째 행에 비퍼를 쭉 내려 놓는 프로그램을 작성하세요.



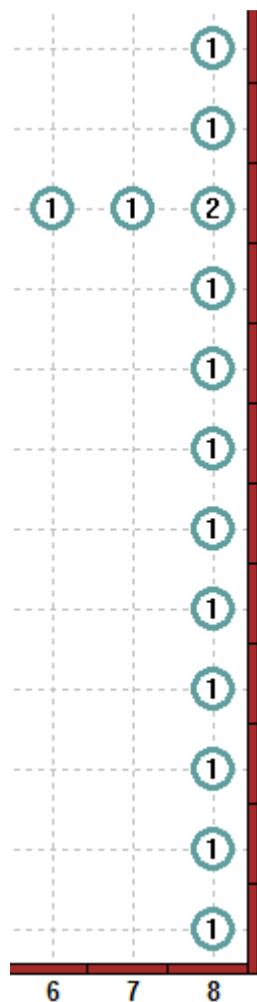
이제 좀더 진행하기 전에 이 프로그램을 저장하세요.

월드 파일을 열어 다시 메모리에 올리고, 오른쪽 하단 모퉁이에 비퍼를 추가해서 다음과 같이 보이도록 하세요. 하지만 리보그는 처음 시작위치 (좌측 하단 첫 번째 위치)에서 있습니다.



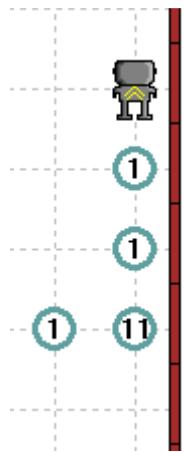
리보그는 다음을 수행합니다.

1. 앞에서 수행했듯이, 10 번째 행에 비퍼를 쭉 내려 놓습니다.
2. 화면의 바닥 오른쪽으로 가서, 두 개의 비퍼(각 8 개, 4 개)들을 줍습니다.
3. 아래 보여지듯이 12 개 비퍼를 수직 열로 뿌려 놓습니다. (중요: 설명을 드리지 않은 carries_beeper() 명령문을 사용할 필요가 있을 것입니다. while carries_beeper(): ... 처럼 시도해 보세요)



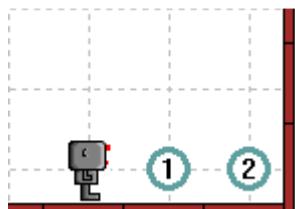
이제, 비퍼 수평선에는 두 개의 비퍼와 이월로 사용할 나머지 잉여 비퍼가 있습니다.
이제 여러분이 해야 할 것은 다음을 작성하는 것입니다.

1. 수평선 아래 9개 비퍼를 주워 버리는 것입니다. (아마도 수평선의 모든
비퍼를 주워서)
2. 아래 보여지듯이 리보그는 마지막 비퍼를 지나 계속 북쪽으로 이동합니다.



3. 돌아서서 비퍼 하나를 줍고 이동하게 합니다. 이를 리보그가 더 이상 비퍼가 없는 모퉁이에 도달할 때까지 반복합니다. 이 지점에서 리보그는 비퍼 3개를 가지고 있어야 합니다.
4. 리보그가 벽에 도착할 때까지, 리보그는 이 비퍼를 가지고 다닙니다.
5. 리보그가 모든 비퍼(3)를 내려 놓고, 하나(이월)를 집어서, 서쪽으로 이동하고, 이월 비퍼를 내려 놓습니다.

남은 마지막 일은 결과를 보여드리기 위해서 리보그를 길 밖으로 빼는 것입니다.



사실... 이 5 단계는 상당히 많은 프로그램 코드를 필요로 하고, 제대로 작동하게 만들기가 꽤나 어렵습니다. 하지만, 체계적으로 한다면 하실 수 있습니다. 시도해 보세요!

4. 3+5 더하기

그래서, 종국에 $8+4$ 를 계산하는 프로그램을 만들었습니다. 훌륭합니다. 이제 $3+5$ 를 시도해 봅시다. 작동하나요? 작동이 되지 않죠... 이 연산은 이월을 필요하지 않습니다. 다시 돌아가서 프로그램을 수정해서 두 숫자를 더하는 연산 $0+0$ 에서 $9+9$ 까지 잘 돌아가도록 하세요.

5. 복수 자리 숫자 더하기

다음은 단자리 덧셈을 복수자리 덧셈으로 일반화하는 것입니다. while 문 안의 코드의 핵심부분을 모든 열에 대해서 싸는 것이 제 제안입니다. 이것은 연습문제로 남겨 놓습니다.

6. 최종 도전

10 진수에 말고 다른 진법의 숫자를 더한다는 의미를 알고 있다면 프로그램을 변경해서 다른 진법의 숫자를 더하는 프로그램을 작성해 보세요.

7. 다음은?

지금까지, 리보그 월드안에서, 파이썬 키워드 def, elif, else, if, not, pass, while 를 살펴봤습니다. 다소 복잡한 프로그램을 작성해서, 리보그가 두 숫자를 더하는 것으로 마무리 했습니다. 이제 리보그 월드를 벗어나서 파이썬으로 좀더 쉽게 두 숫자를 더할 수 있는지 살펴봅시다. 리보그 월드로 돌아가기 전에 파이썬에 대해서 좀더 학습할 것입니다.

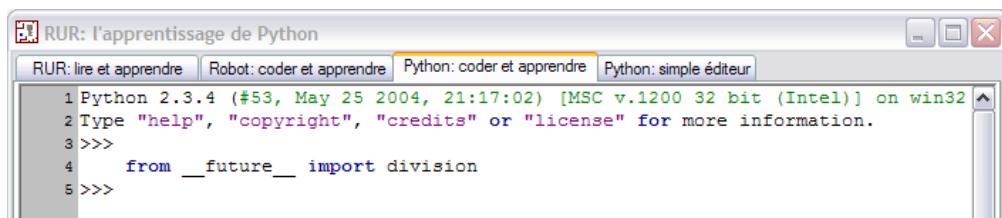
◀ 반복 피하기 – 가져오는 것 -  - 파이썬은 이미 더하기를 알고 있어요 ▶

파이썬은 이미 더하기를 알고 있어요

마지막 학습에서, 리보그를 지도하여 두 숫자를 어떻게 더하게 하는지 배웠습니다. 이번 학습에서는, 파이썬이 어떻게 숫자를 더하고 그 이상을 수행하는지 배울 것입니다.

1. 중요한 것부터 먼저

세 번째 탭을 클릭하여, 파이썬 인터프리터(Python Interpreter, Python: Code and Learn)을 선택합니다. 화면이 다음에 보이는 것과 같을 것입니다.



지금은 처음 4 줄을 무시하세요. 다섯 번째 줄에 파이썬 프롬프트입니다.

>>>

5 행에 커서를 놓고 “8+4”를 타이핑하고 “enter” 키를 누릅니다. 화면에 다음과 같이 보일 것입니다.

```
>>> 8+4
12
>>>
```

놀랍죠! 파이썬이 방금 두 숫자를 더했습니다. 두 숫자를 더하는 것은 리보그가 하기에는 너무 어려웠습니다. 여러분에게 프롬프트(Prompt, >>>)로 더 많은 명령어 주기를 재촉합니다. “오려 붙이기(cut and paste)”를 사용하여 인터프리터에서 여기로 옮긴 후에 아래처럼 수행하세요.

```
>>> 8+4
12
>>> 8-4
4
>>> 8*4
32
>>> 8/4
2.0
```

파이썬은 더하고, 빼고, 곱하고, 나누기를 어떻게 하는지 알고 있습니다. 마지막 사례는 설명이 필요합니다. 시간을 가지고 모두 잘 읽어보세요. 그다지 복잡하지 않습니다.

2. 나누기

“8/4”의 결과로 “2” 대신에 파이썬은 “2.0”을 출력합니다. 파이썬 나눗셈의 정상적인 결과값은 아닙니다. 통상 파이썬에서 정수 나누기 정수는 결과값으로 정수가 됩니다. 파이썬은 $8/4=2$ 를 준다는 의미입니다... 하지만, $8/3=2$ 를 줄 수 있습니다. 처음 나눗셈을 배울 때 8 나누기 3은 몫이 2이고 나머지가 2로 배웁니다. 다른 말로 $8/3 = 2+2/3$ 이 됩니다. 사실 여러분이 처음 정수 나눗셈을 배울 때, 8을 3으로 나눌 수 없다고 배웁니다. 후에 나눗셈의 나머지를 배우고, 그 다음에는 소수점에 대해서 배웁니다.

통상 파이썬은 여러분이 처음으로 배운 정수 나눗셈을 다룹니다. 파이썬에서 $1/2=0$ 이라는 조금 우스운 상황이 되기도 합니다. 하지만, 파이썬에게 두 소수점을 가진 숫자 혹은 하나는 정수, 다른 하나는 소수점이 있는 숫자로 나눗셈을 한다면 기대한 답($1/2=0.5$)을 얻을 수 있습니다.

파이썬의 향후 버전에서는 다르게 동작합니다. 4 번째 줄에 보여지는 division 모듈의 future 를 가져오기로 수행을 한 것입니다. 정수 나눗셈을 수행하고자 한다면 아래처럼 이중 나눗셈을 사용하세요.

```
>>>8/3  
2  
>>> 8.0//3.  
2.0
```

단지 제 말씀을 받아들이지 말고, 직접 타이핑하여 시도해 보세요.

이중 나눗셈 부호는 부동 소수점 나눗셈으로 알려져 있습니다. 나눗셈의 몫을 취하고 가장 가까운 정수로 내림 합니다. 정수 두 개를 나눈다면, 결과는 정수입니다. 둘 중 하나 혹은 모두 소수점 숫자라면, 결과도 가장 가까운 정수에 내림한 소수점 숫자가 됩니다.

3. 더 많은 숫자

파이썬은 수학 연산자의 우선순위 규칙을 알고 있습니다. 많은 숫자와 연산자가 있는 경우, 왼쪽에서 오른쪽 순서로 곱셈과 나눗셈을 먼저 수행하고,

덧셈과 뺄셈을 그 다음 순서로 수행합니다. 글로 의미를 전달하는 것보다 몇 개의 예제가 더 좋은 설명이 될 것입니다.

```
>>> 2+3*5  
17  
>>> 2+ (3*5) # 여기서 공백에 대해서는 파이썬은 무시합니다.  
17  
>>> (2+3)*5  
25  
>>> 2*4/8  
1.0  
>>> 2*4//8  
1  
>>> 2+1-4  
-1
```

스스로 연습문제를 좀 더 해보세요.

4. 더 많은 수학 연산자

파이썬에서 기본 수학 연산자보다 많은 것을 사용할 수 있습니다. 여기 여러분이 시작해서 확장 가능한 몇 가지 예제가 더 있습니다.

```
>>> 3*3*3*3  
81  
>>> 3**4 # 제곱  
81  
>>> 7%3 # 나머지 연산: 7/3 = 2, 나머지 1  
1  
>>> 7.0 % 3  
1.0  
>>> 7.25 % 3  
1.25  
>>> 7.3 % 3  
1.299999999999998
```

마지막 결과는 여러분이 기대하는 거의 1.3과 같습니다. 1.3과 1.299999999999998의 차이는 미미하며, 컴퓨터 소수점 연산에서 기인합니다. 컴퓨터 프로그래밍 전반에 대해서 공부하게 되면, 추후 학습에서 어떻게 이런 결과가 나왔는지 설명할 것입니다. 실무에서, 정확한 결과값과 파이썬에서 계산된 값의 차이는 문제가 되지는 않습니다.

5. 매우 큰 수

다음을 시도해 보세요.

```
>>> n = 2147483646  
>>> n+1  
2147483647  
>>> n+2  
2147483648L
```

숫자 2147483646과 동의어로 “n”을 사용했습니다. 프랑스 독자를 위한 avance를 move로 번역할 때 동의어에 대해서 논의한 것을 기억하십시오. 파이썬은 로봇 명령어 말고 다른 동의어도 지원합니다. 컴퓨터 전문용어로 동의어 대신에 변수(variable)이라고 합니다. 그래서, n은 변수이고, 변수값은 2147483646입니다.

n+1은 우리가 기대한 것이지만, n+2는 끝에 흥미로운 문자 L이 붙어있습니다. 이것은 컴퓨터 메모리에 쉽게 표기될 수 없는 긴 정수(Long Integer)로 알려진 것을 나타냅니다. 32비트 중앙처리장치 칩을 가진 컴퓨터에서는 모든 양수가

31 비트의 다양한 조합으로 표현이 되고, 표현할 수 있는 가장 큰 정수는 $2^{31}-1=2147483647$ 입니다. 이 보다 큰 숫자에 대해서 수학적 연산을 파이썬에서 수행하려고 하면 고등 기법을 사용해야 합니다. 이런 연산은 속도가 느리며, 파이썬이 숫자의 마지막에 L을 붙여서 여러분에게 상기시켜 줍니다. 꼭 필요하지 않는다면, 매우 큰 수를 사용하는 것을 삼가세요.

6. 과학적 표기법

여러분은 거의 과학적 표기법에 친숙할 것입니다.

$$2000 = 2 \times 10^3$$

$$0.25 = 2.5 \times 10^{-1}$$

과학적 표기법은 매우 편리하게 매우 작거나 큰 수를 표기하는데 유용합니다. 파이썬으로 과학적 표기법을 사용할 수 있지만, $\times 10$ 을 나타내기 위해서 문자 E 혹은 e를 사용합니다.

7. 다른 형식의 숫자

파이썬을 사용해서 복소수, 8 진수, 16 진수를 다룰 수 있습니다. 이것들이 무엇인지 모른다고 걱정하지 마세요. 나중에 필요할 때 무엇인지 설명해 드리겠습니다.

◀ 반복 피하기 – 가져오는 것 -  - 파이썬은 이미 더하기를 알고 있어요 ▶

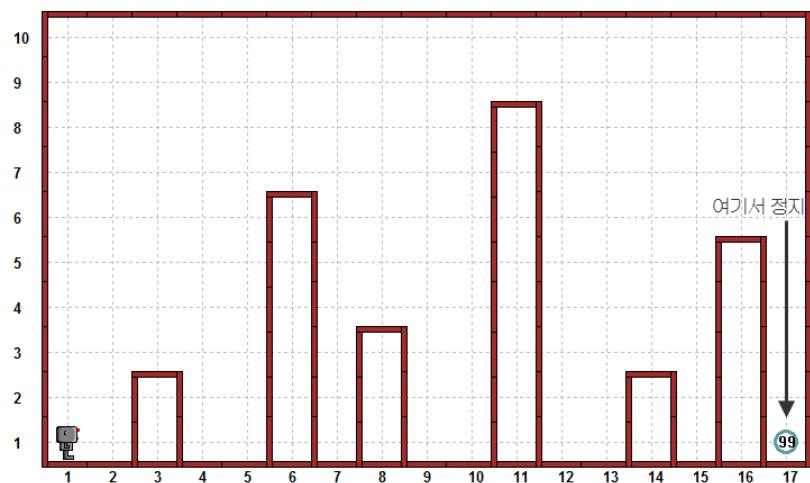
제 III 부

연습문제

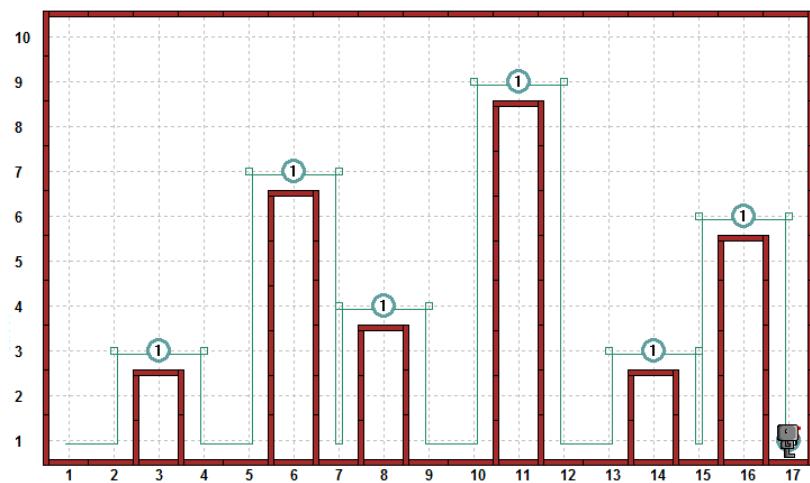
문제 1

빌딩 옥상에 안테나를 설치하는 리보그 제작을 의뢰 받았습니다. 아래와 같이 각 빌딩 옥상에 안테나(비퍼)를 설치하도록 코드를 작성하시오(단, 빌딩의 개수와 높이는 임의로 바뀔 수 있습니다).

[초기 상태 예시]



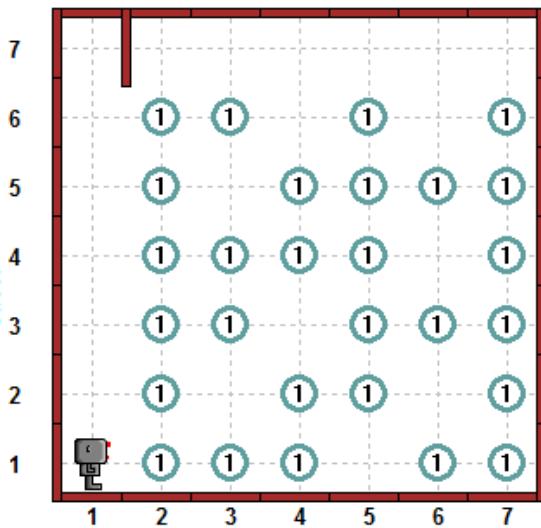
[실행 결과 예시]



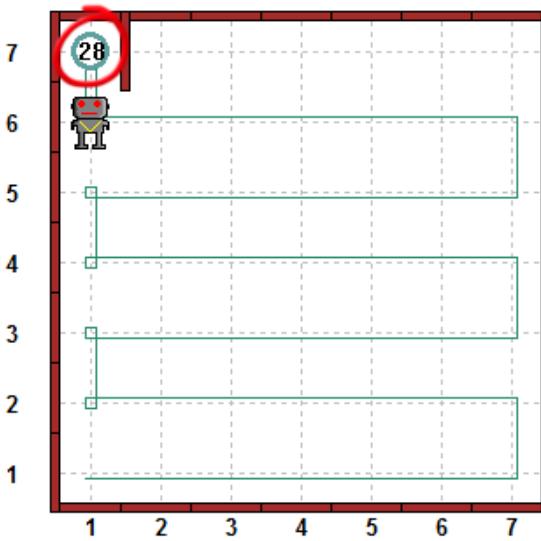
문제 2

농작물을 수확하는 리보그 제작을 의뢰 받았습니다. 농작물(비퍼)을 수확하여 창고에 모두 저장하는 코드를 작성하시오(단, 농작물의 개수는 임의로 바뀔 수 있습니다.)

[초기 상태 예시]



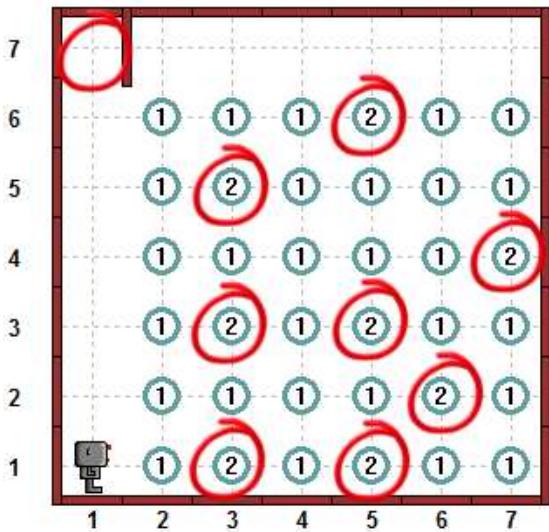
[실행 결과 예시]



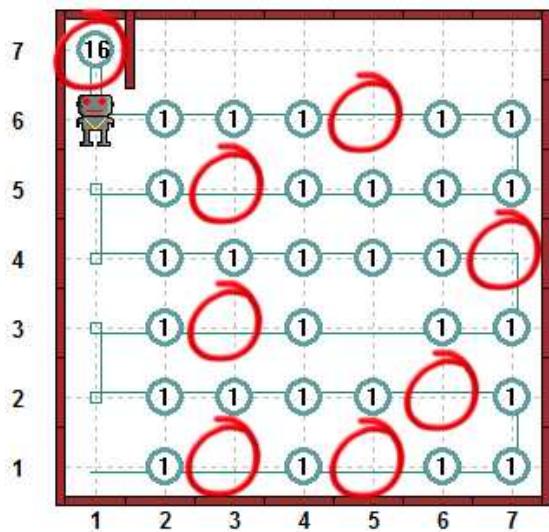
문제 3

딸기만(②) 수확하여 창고에 모두 저장하는 코드를 작성하시오(단, 딸기의 위치는 임의로 바뀔 수 있습니다.)

[초기 상태 예시]



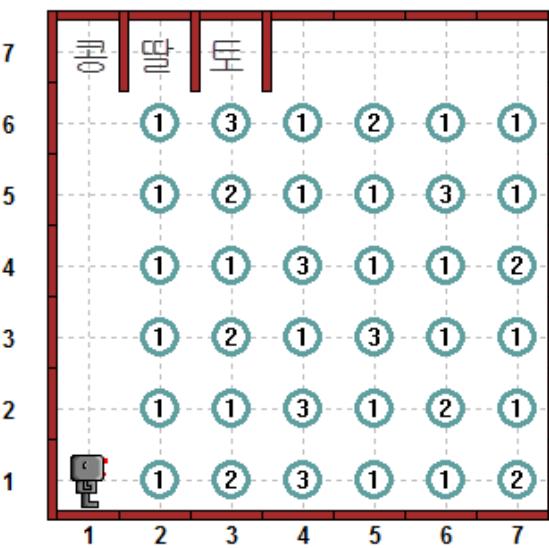
[실행 결과 예시]



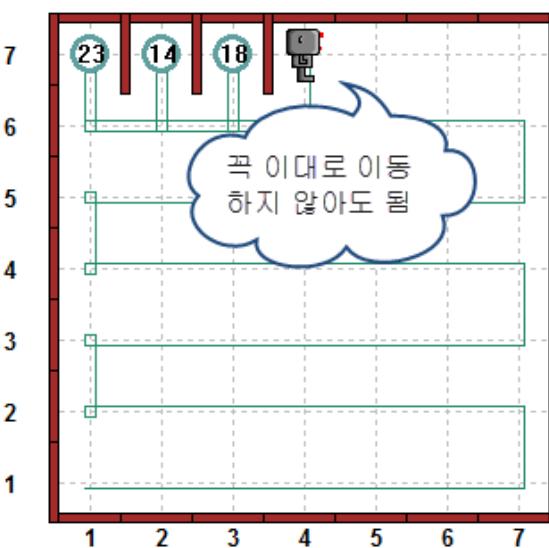
문제 4

콩(①), 딸기(②), 토마토(③)를 수확하여 각각 창고에 저장하는 코드를 작성하시오(단, 농작물의 위치와 개수는 임의로 바뀔 수 있습니다.)

[초기 상태 예시]



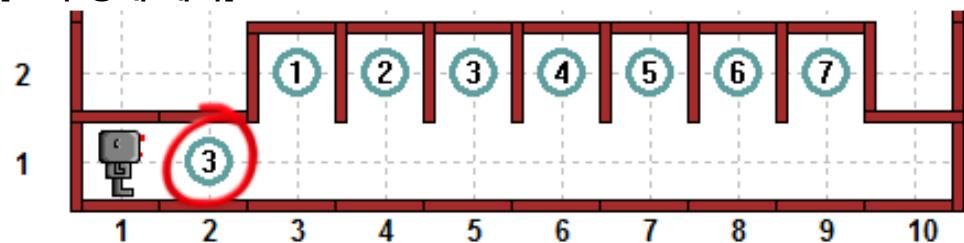
[실행 결과 예시]



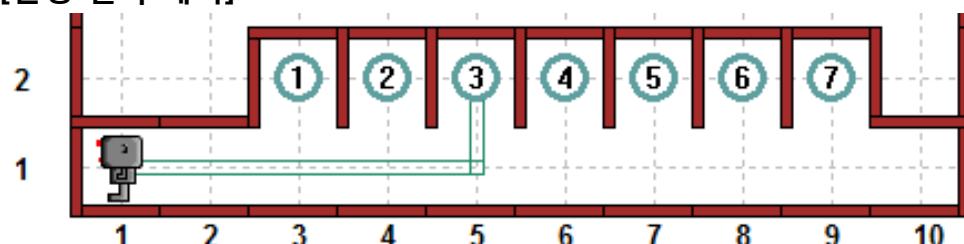
문제 5

식당에서 손님의 주문을 받는 리보그 제작을 의뢰 받았습니다. 아래와 같이 호출한 방의 번호를 확인하고, 해당 방으로 이동하여 주문을 받아 돌아오는 코드를 작성하시오. (단, 호출한 방의 번호는 임의로 바뀔 수 있습니다.)

[초기 상태 예시]



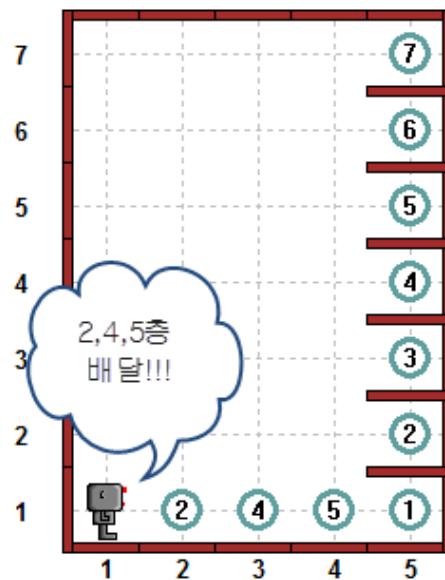
[실행 결과 예시]



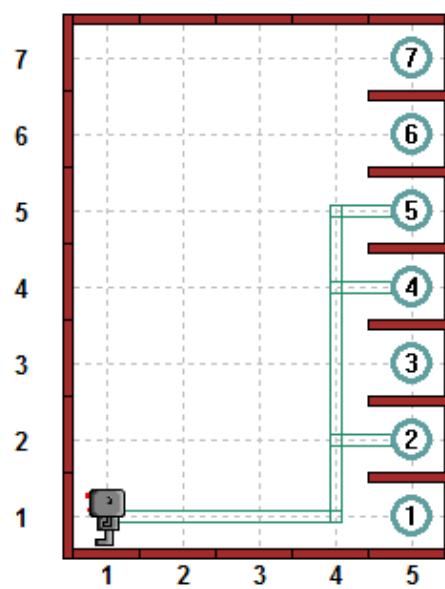
문제 6

피자를 배달하는 리보그 제작을 의뢰 받았습니다. 아래와 같이 주문한 층 번호를 확인하고 해당 층으로 이동하여 피자를 배달하고 돌아오는 코드를 작성하시오
(단, 배달하는 곳은 항상 3 곳이며, 임의로 층수가 바뀔 수 있습니다.)

[초기 상태 예시]



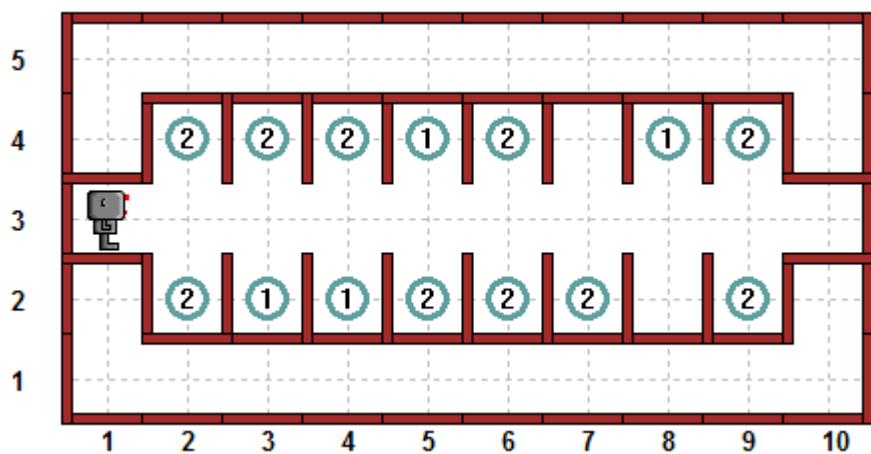
[실행 결과 예시]



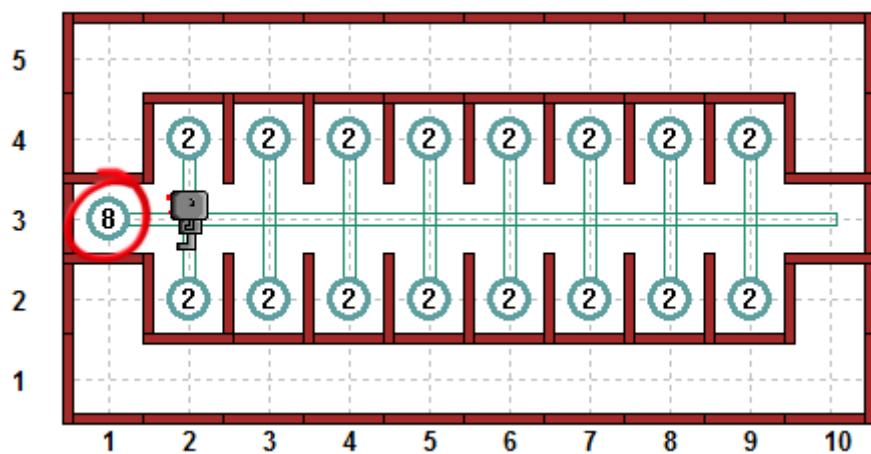
문제 7

호텔 룸서비스 용 리보그 제작을 의뢰받았습니다. 호텔 객실에는 수건(비퍼)이 2개씩 비치되어야 하는데, 가끔 일부 몰지각한 사람들이 수건을 가져가는 경우가 있습니다. 따라서 리보그는 아래와 같이 객실을 순회하며 분실된 수건을 채워 넣고, 분실된 수건이 몇 장인지 총지배인에게 알려주어야 합니다. 이와 같은 동작을 하는 코드를 작성하시오(단, 수건의 개수는 임의로 바꿀 수 있습니다.)

[초기 상태 예시]



[실행 결과 예시]



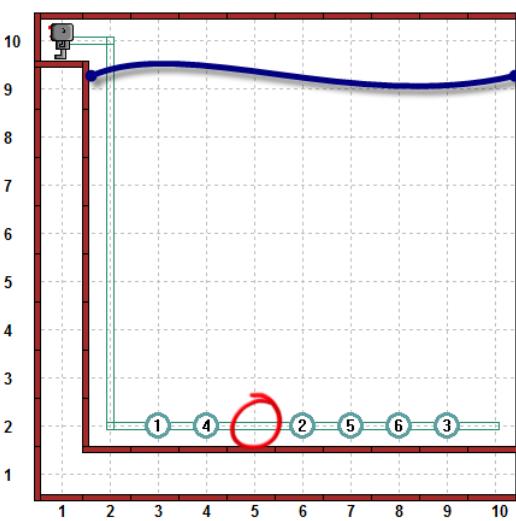
문제 8

해저 탐험 용 리보그 제작을 의뢰 받았습니다. 리보그는 바다 밑을 탐색하여 보물(비퍼)을 건져 와야 합니다. 그러나 안타깝게도 보물은 하나만 건져 올 수 있다고 합니다. 가장 가치가 높은 보물(비퍼의 숫자가 높을 수록 가치가 높은 보물입니다.)을 건져 오는 코드를 작성하시오.

[초기 상태 예시]



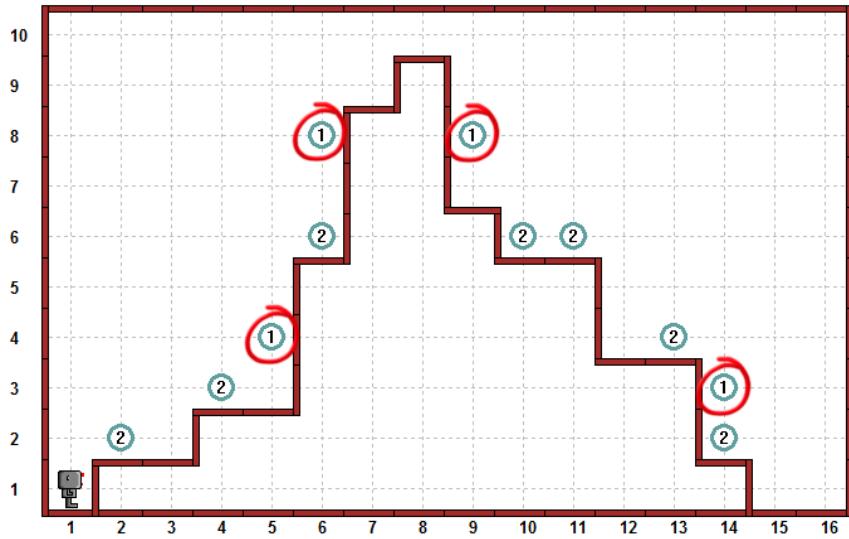
[실행 결과 예시]



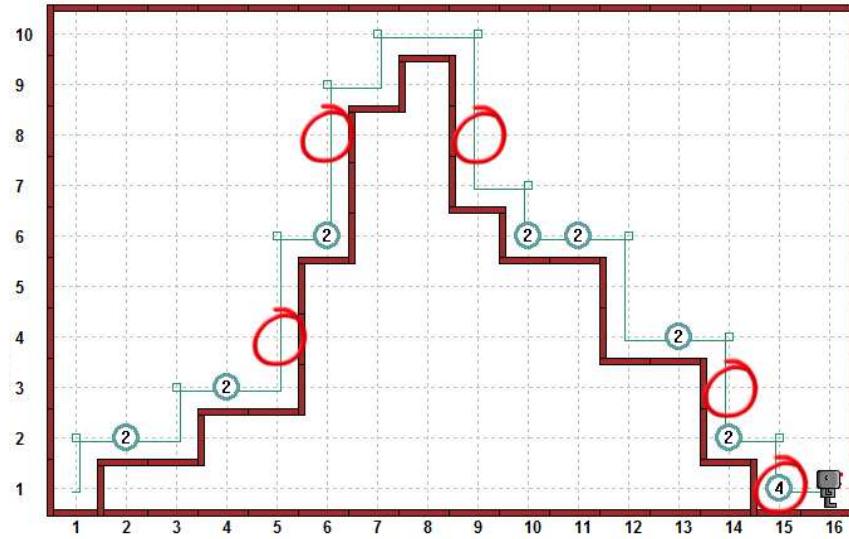
문제 9

산악 구조 용 리보그 제작을 의뢰받았습니다. 리보그가 산을 등반하면서 조난당한 등산객(①)을 구조하여 내려올 수 있도록 코드를 작성하시오. (단, 조난당한 등산객의 위치는 임의로 바뀔 수 있습니다.)

[초기 상태 예시]



[실행 결과 예시]



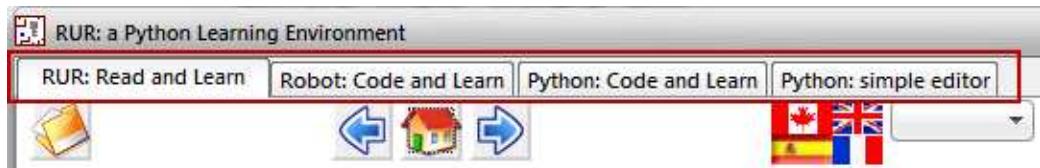
제 IV 부

러플(RUR-PLE) 사용법

프로그램 및 화면 구성

1. 프로그램 구성

러플(RUR-PLE)은 4 개의 탭으로 구성되어 있으며 각각의 탭은 다음과 같습니다.



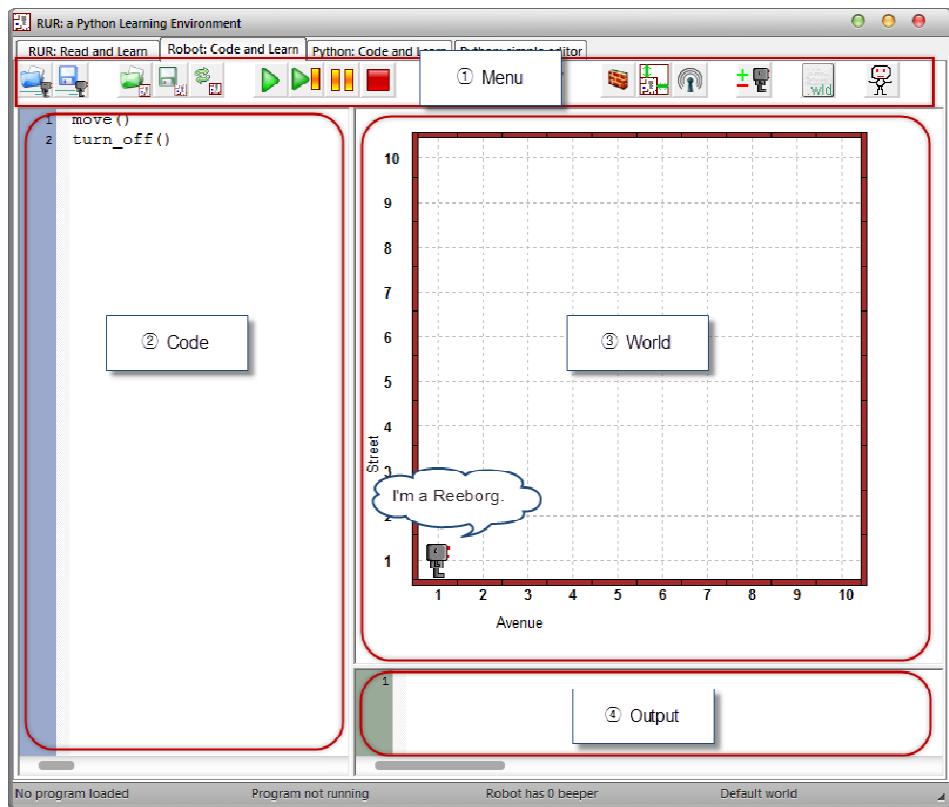
- RUR: Read and Learn(튜토리얼)
- Robot: Code and Learn(로봇을 이용한 프로그래밍 학습 환경)
- Python: Code and Learn(파이썬 템)
- Python: Simple editor(파이썬 에디터)

본 교재는 4 가지 탭에서 2 번째 탭인 Robot 을 이용한 프로그래밍 학습에 중점을 둔다.

2. 화면 구성

1) 화면 구성

- ① Menu : 코드파일 및 월드파일 열기, 저장, 실행 등의 메뉴버튼
- ② Code : 리보그가 동작할 명령을 입력하는 부분
- ③ World : 리보그가 움직일 가상의 공간
- ④ Output: 데이터를 출력할 수 있는 부분



2) 메뉴 구성



기본 명령

1. 수동 조작

키보드의 방향키를 이용하여 리보그를 이동시킬 수 있습니다.

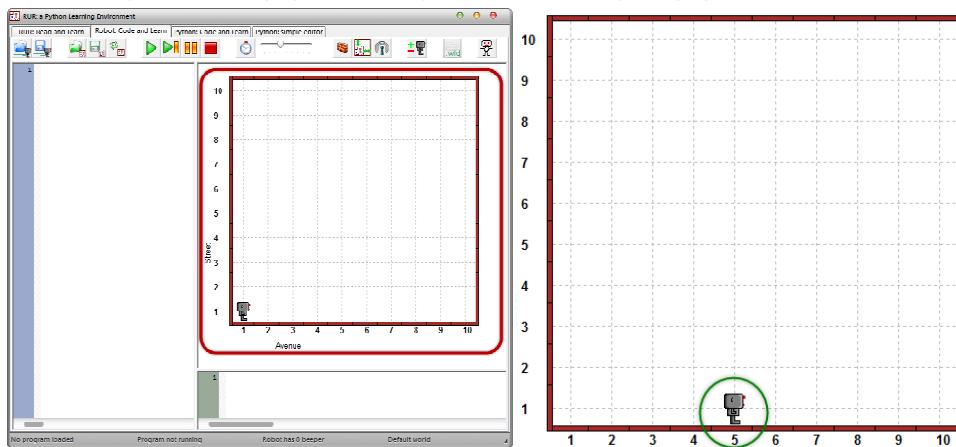
방향키는 ←, ↑ 만 사용 가능합니다.

↑ : 리보그가 바라보는 방향으로 한 칸 이동

← : 제자리에서 왼쪽으로 90 도 회전

[따라 하기]

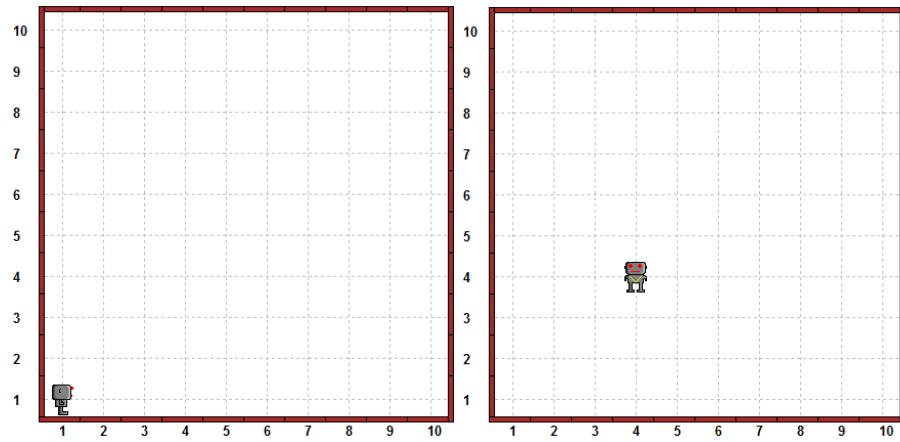
- 월드 부분을 클릭하고 방향키 ↑ 를 4 번 입력합니다.



- 결과를 확인하고 월드초기화버튼()를 클릭하면 리보그가 초기 위치로 돌아갑니다.

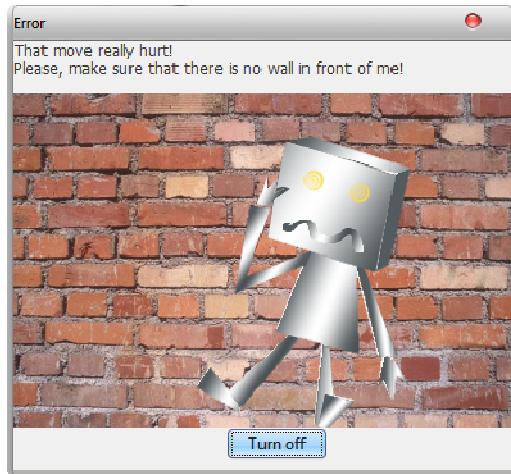
[스스로 하기 1]

리보그를 다음과 같이 이동 시키시오.



[스스로 하기 2]

리보그를 이동하다가 벽에 부딪쳐 보시오.



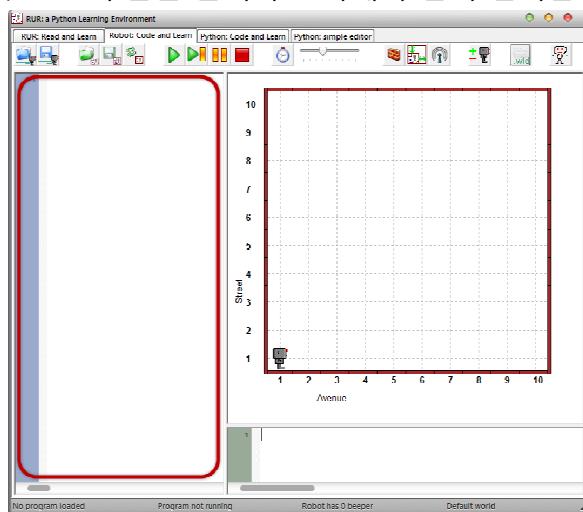
벽에 부딪치면 아래와 같은 에러창이 나타난다. 에러가 발생하면 리보그는 더 이상 움직이지 않기 때문에 에러가 발생하지 않도록 주의해야합니다.

2. move()

move()는 키보드의 방향키 ↑ 와 같은 역할을 하는 명령으로 리보그를 현재 방향으로 한 칸 앞으로 이동시킵니다.

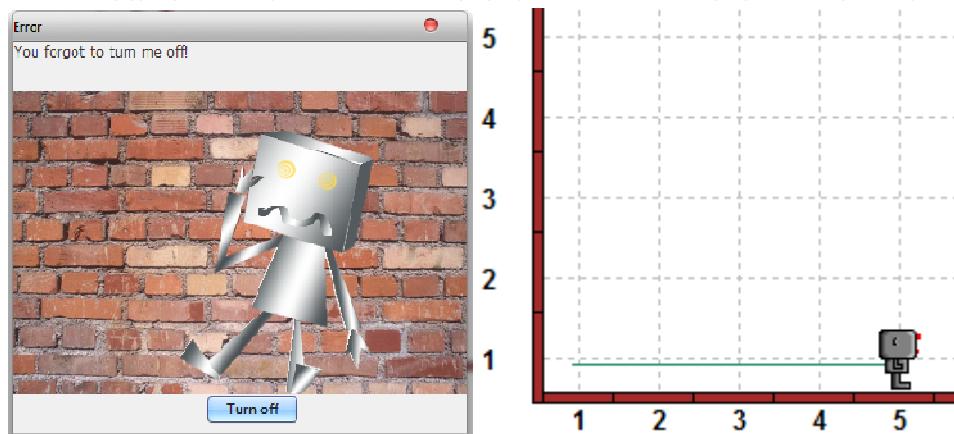
[따라 하기]

1) 코드 부분을 클릭하고 아래와 같이 입력합니다.



1 move ()
2 move ()
3 move ()
4 move ()

2) 상단 메뉴의 실행버튼()을 클릭하여 코드를 실행시키고 결과를 확인합니다.



전원을 끄지 않았다는 에러가 발생합니다. (리보그는 코드 마지막에 전원 끄기 명령을 주어야 합니다.) 일단 [Turn off] 버튼을 클릭하면 리보그가 4 칸 전진한 결과를 확인할 수 있습니다. 키보드로 움직인 것과 다르게 자신이 이동한 경로가 표시됩니다.

[스스로 하기 3]

아래와 같은 결과가 나오도록 코드를 작성하시오.

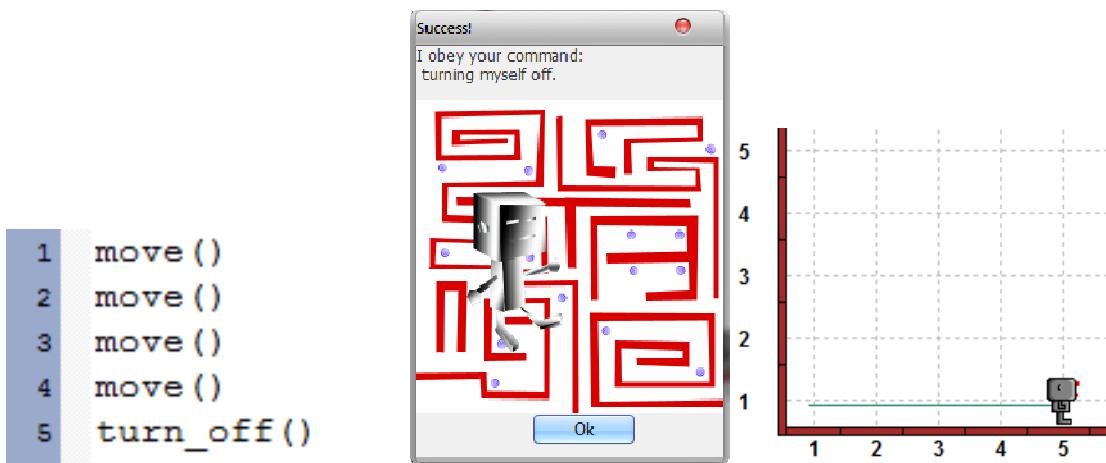


3. `turn_off()`

`turn_off()`는 리보그의 전원을 끄는 명령입니다. 따라서 `turn_off()`가 실행되면 리보그는 더 이상 움직이지 않습니다.

[따라 하기]

아래의 코드를 입력하고 실행하여 결과를 확인합니다.



[스스로 하기 4]

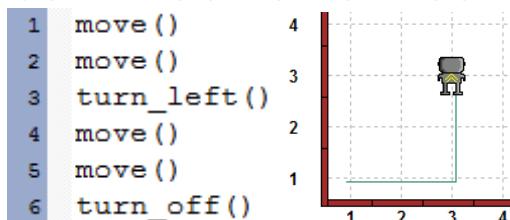
아래 코드의 결과를 예상해 본 후, 입력하여 확인하고 예상한 결과와 비교하시오.

4. turn_left()

turn_left()는 키보드의 방향키(와 같은 역할을 하는 명령으로 리보그를 현재 위치에서 왼쪽으로 90 도 회전 시킵니다.

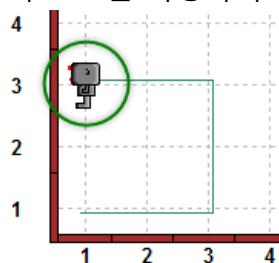
[따라 하기]

1) 아래 코드를 입력하고 실행하여 결과를 확인합니다.



[스스로 하기 5]

아래와 같은 결과가 나오도록 코드를 작성하시오.



※ 리보그의 방향에 주의하십시오.

[스스로 하기 6]

아래 코드의 결과를 예상해 본 후, 입력하여 확인하고 예상한 결과와 비교하시오.

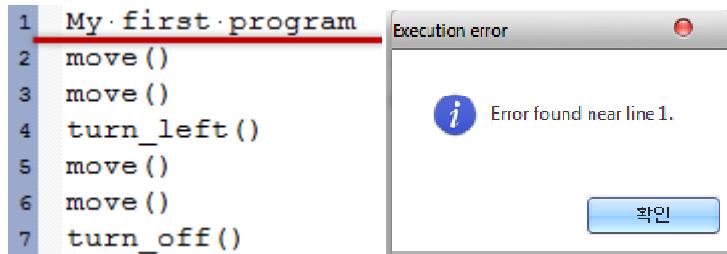
```
1 move()
2 move()
3 turn_left()
4 turn_left()
5 move()
6 move()
7 turn_off()
```

5. 주석문

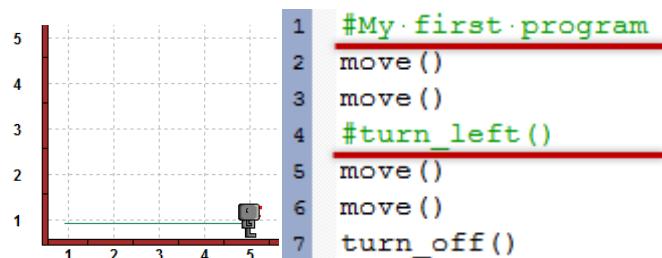
주석문은 주로 코드에 대한 설명이나 일시적으로 해당 코드를 실행하고 싶지 않을 때 사용합니다. # 기호 이후에 나오는 문자는 실행하지 않습니다.

[따라 하기]

1) 아래 코드를 입력하고 실행하여 결과를 확인합니다.



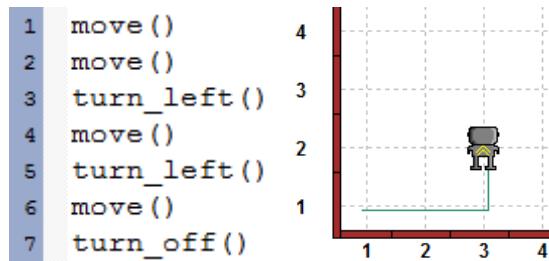
첫 번째 줄에서 에러가 발생합니다. My first program 이라는 명령을 리보그는 이해하지 못합니다. My first program 은 리보그가 실행할 명령이 아니라 이 프로그램에 대한 설명이기 때문에 주석처리 해야 합니다.



4 번째 줄의 turn_left() 명령이 주석처리 되었기 때문에 실행되지 않았다.

[스스로 하기 7]

아래와 같은 결과가 나오기 위해 코드를 주석처리 하시오.

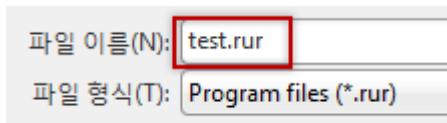


6. 코드 저장하기

코드를 재사용하기 위해 저장합니다.

[따라 하기]

- 1) 상단 메뉴의 프로젝트 저장버튼()을 클릭합니다.
- 2) 저장할 폴더를 선택하고 원하는 파일명을 입력한 후 확장자를 “rur”로 지정하여 저장합니다.



- 3) 러플(RUR-PLE) 프로그램을 재실행한 후, 상단 메뉴의 프로젝트 저장버튼() 버튼을 클릭합니다.
- 4) 파일을 저장한 폴더로 이동하여 코드파일을 선택하고 열기버튼을 클릭합니다.

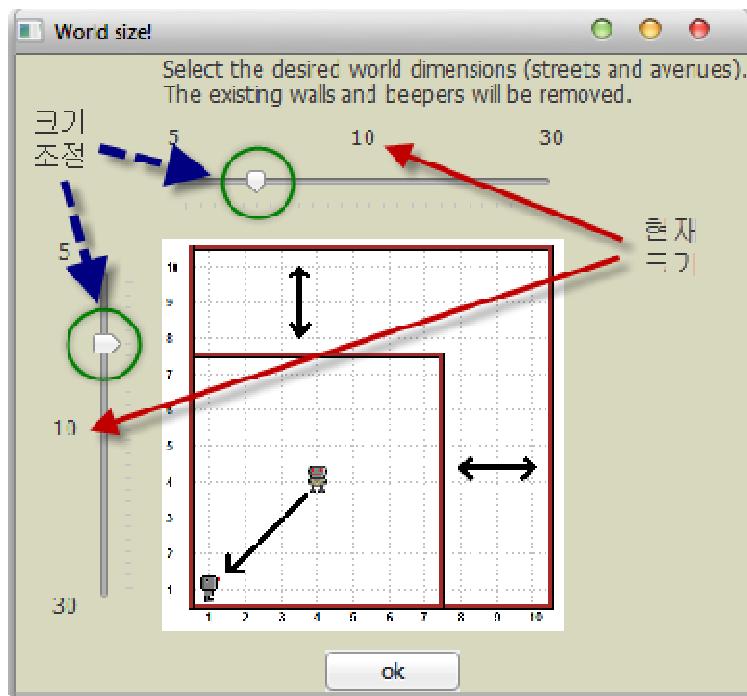
월드 꾸미기

1. 월드 크기 바꾸기

러플(RUR-PLE)의 기본 월드 크기는 10×10 이지만, 월드의 크기를 최소 5×5 에서부터 최대 30×30 까지 변경할 수 있습니다.

[따라 하기]

- 1) 상단 메뉴의  버튼을 클릭합니다.
- 2) World Size 창이 뜨면 게이지를 조절하여 월드 크기를 변경합니다.
- 3) 상단 메뉴의 월드초기화버튼()을 클릭하면 월드의 초기 상태로 돌아갑니다.



[스스로 하기 8]

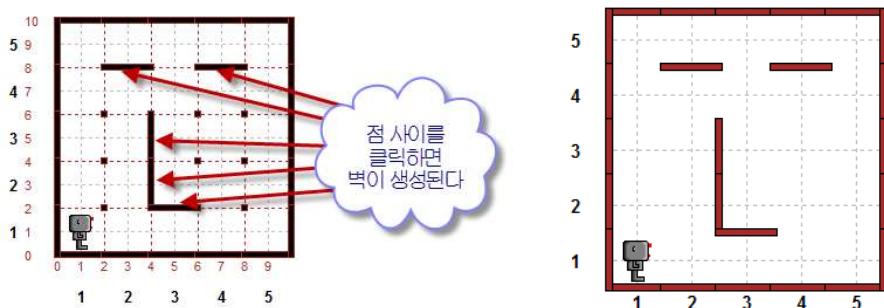
월드의 크기를 8×7 로 변경하시오.

2. 벽 만들기

러플(RUR-PLE) 월드 내에 벽을 만들어 리보그의 이동에 제약을 줄 수 있습니다.

[따라 하기]

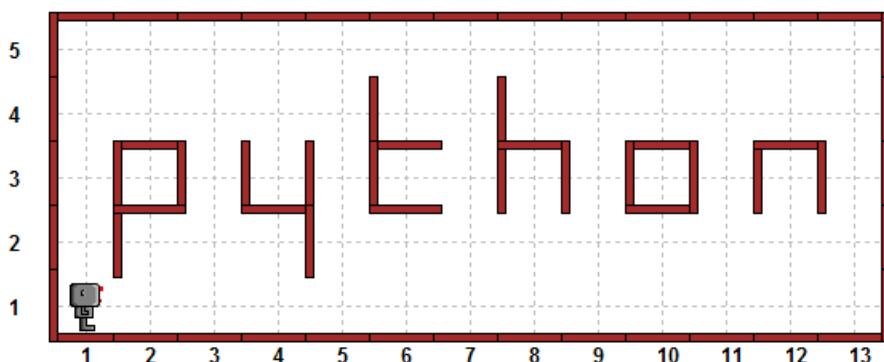
- 1) 상단 메뉴의 벽버튼()을 클릭하면 화면에 점이 생기고, 그 점 사이를 클릭하면 벽이 생성됩니다. 생성된 벽을 클릭하면 사라집니다.



- 2) 벽버튼()을 다시 클릭하면 화면의 점이 사라지고 생성된 벽만 남습니다.
- 3) 월드를 수정하고 싶을 때 벽버튼()을 다시 클릭합니다.

[스스로 하기 9]

아래와 같은 월드를 만드시오.

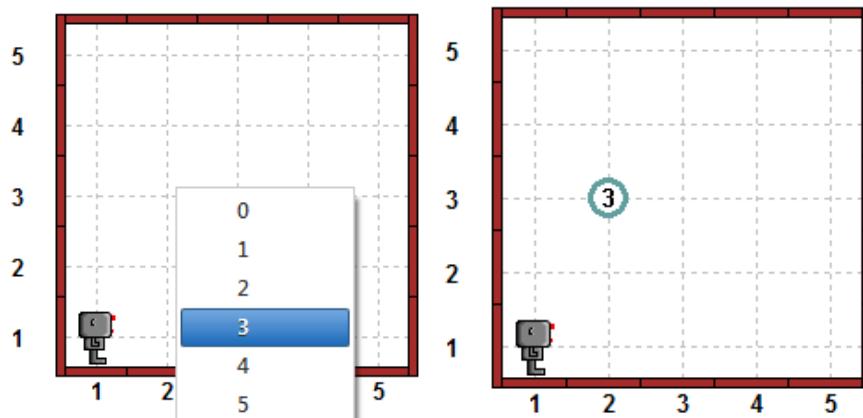


3. 비퍼 놓기

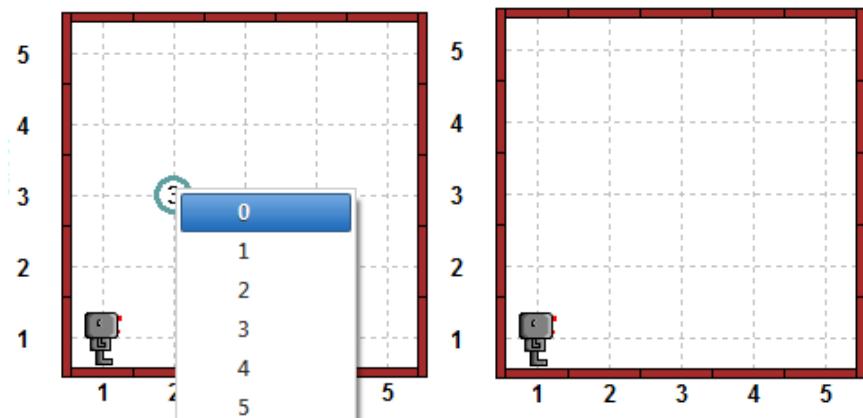
비퍼(beeper)는 러플(RUR-PLE) 월드에 배치할 수 있는 물건이며 다양한 용도로 쓰입니다. 비퍼는 리보그가 주울 수 있습니다. (물론 다시 버릴 수 있습니다.)

[따라 하기]

- 1) 월드에서 마우스 오른쪽 버튼을 클릭하면 아래(좌)와 같이 숫자가 적힌 메뉴가 나옵니다. 아래(우)와 같이 선택한 숫자가 적힌 비퍼가 생성됩니다.

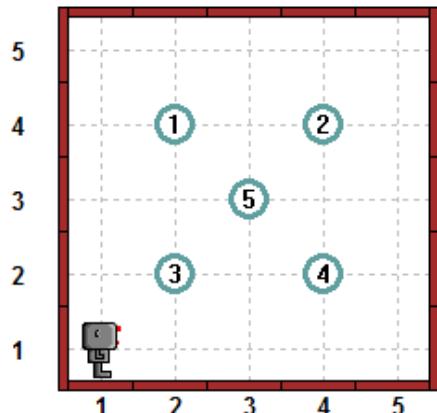


- 2) 다시 비퍼 위에서 마우스 오른쪽 버튼을 클릭하여 왼쪽아래그림과 같이 '0'을 클릭하면 비퍼가 사라집니다.



[스스로 하기 10]

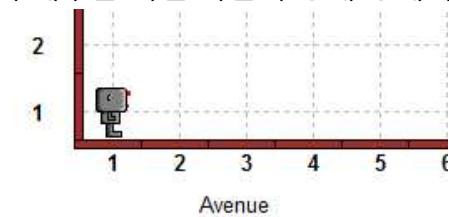
아래와 같이 월드에 비퍼를 생성하시오.



비퍼가 나타내는 숫자는
비퍼의 개수를 의미한다.

4. 리보그에게 비퍼 주기

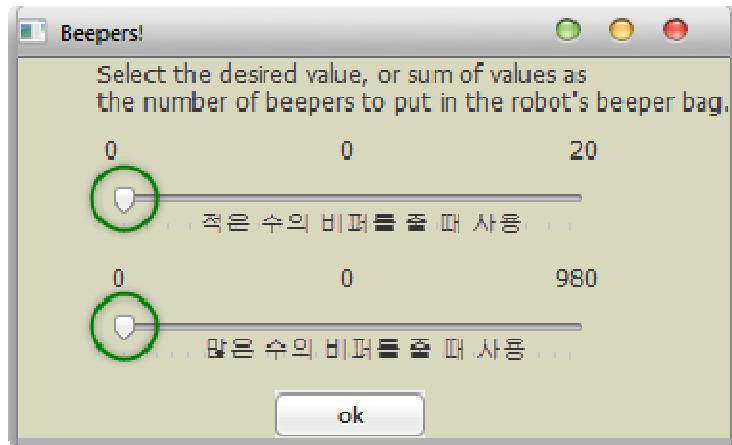
리보그는 비퍼를 보관할 수 있는 가방이 있습니다. (보이진 않습니다.) 가방에 보관하고 있는 비퍼의 개수를 화면 하단의 상태 창에서 확인할 수 있습니다.



비퍼주기버튼()을 클릭하여 리보그에게 비퍼를 줄 수 있습니다.

[따라 하기]

- 1) 상단 메뉴의 비퍼주기버튼()을 클릭하여 리보그에게 비퍼를 줍니다.



- 2) 상태 창에서 리보그가 보관하고 있는 비퍼의 수를 확인합니다.

5. 리보그 삭제/추가

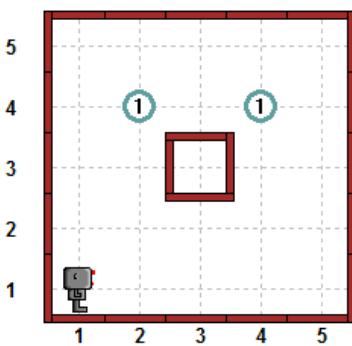
리보그삭제/추가버튼() 버튼을 클릭하면 현재 월드에 리보그가 삭제/추가 됩니다.

6. 월드(월드) 저장

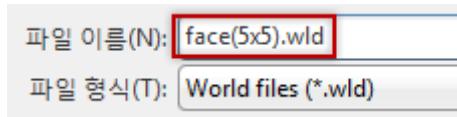
월드를 재사용하기 위해 저장합니다.

[따라 하기]

- 1) 아래와 같은 월드를 만들고, 상단 메뉴의 프로젝트 저장버튼()을 클릭합니다.

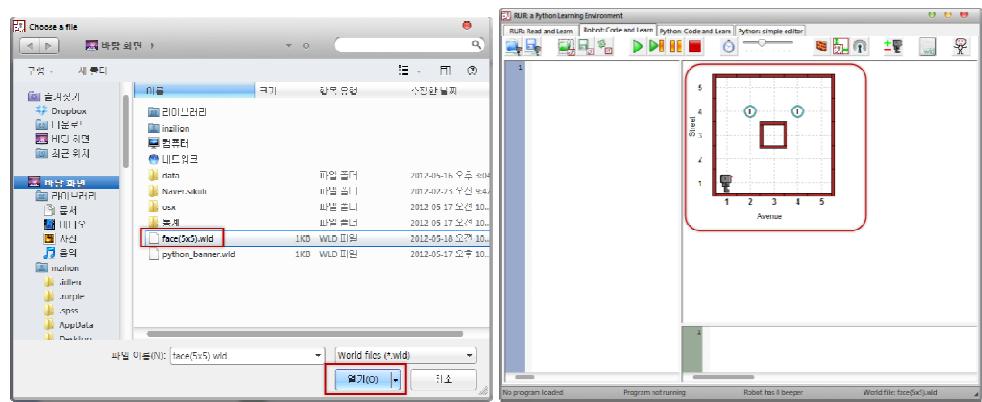


- 2) 저장할 폴더를 선택하고 원하는 파일명을 입력한 후 확장자를 'wld'로 지정하여 저장합니다.



- 3) 러플(RUR-PLE) 프로그램을 재실행한 후, 상단 메뉴의 프로젝트 열기버튼()을 클릭합니다.

- 4) 해당 폴더로 이동하여 월드파일을 선택하고 열기버튼을 클릭합니다.

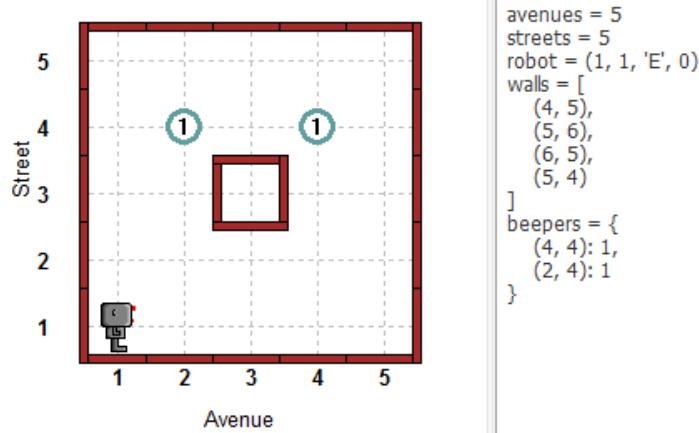


7. 월드 파일 보기

월드파일 보기버튼()을 클릭하여 월드 파일이 저장된 형태를 확인할 수 있습니다.

[따라 하기]

- 1) 상단 메뉴의 월드파일 보기버튼()을 클릭합니다.
- 2) 아래와 같은 월드 파일의 내용을 확인할 수 있습니다.



- 3) 다시 월드파일 보기버튼()을 클릭하면 월드 파일 정보가 사라집니다.

※ 월드파일 내용 설명

