



인공지능 입문

▼ ot

- 수업

탐색알고리즘 → 중간시험 → 머신러닝, 딥러닝 → 기말고사
파이썬 사용

- 과제(20)

수업 내용 정리 ex_기술브이로그 - 매주 금요일까지

- 시험 - 객관식 시험, 시험개수가 많음!

중간시험(35)

기말시험(35)

- 교재

교재없어도 상관은 없음

- 실습환경

vscode, jupyter설치, python

- 숙제

jupyter 설치해오기 (vscode extension)

▼ 인공지능 기초 개념과 역사

정의 : 지능을 이해하는 것 뿐만 아니라, 지능적인 실체, 다양한 새로운 환경에서 효과적이고 안전하게 행동하는 방법을 계산해낼 수 있는 기계


	인간적임	합리적임 비인간적?
사고 중심	Thinking Humanly 인간의 사고, 의사결정, 문제해결, 학습 등의 활동에 '연관시킬 수 있는 활동들(의 자동화)' (Bellman, 1978) 컴퓨터가 생각하게 하는 흥미로운 시도 (...) 문자 그대로의 완전한 의미에서 마음을 가진 기계 (Haugeland, 1985)	Thinking Rationally 계산 모형을 이용한 정신 능력 연구 (Charniak & McDermott, 1985) 인지와 추론, 행위를 가능하게 하는 계산의 연구 (Winston, 1992)
행위 중심 (= 에이전트 중심)	Acting Humanly 사람이 지능적으로 수행해야 하는 기능을 수행하는 기계의 제작을 위한 기술 (Kurzweil, 1990) 현재로서는 사람이 더 잘하는 것들을 컴퓨터가 하게 만드는 방법에 대한 연구 (Rich & Knight, 1991)	Acting Rationally 계산적 지능은 지능형 에이전트의 설계에 관한 연구 (Poole et al., 1998)  인공지능은 (...) 인공물의 지능적 행동에 관련된 것 (Nilsson, 1998) 지능을 행위의 것까지 포함시킴

Figure 1.1 Some definitions of artificial intelligence, organized into four categories.

인공지능 분류체계 (feat, 책)

인공지능의 중요성 - 가치정렬문제(value alignment problem) : 정말로 인간이 원하는 것과 컴퓨터에게 제공하는 목적을 잘 조화

인공지능에 영향을 끼친 학문 - 철학, 수학, 경제학, 신경과학, 심리학, 컴공, 제어이론과 인공두뇌학, 언어학

1. 인간적으로 행동하는 인공지능 : 튜링검사 접근방식

2. 인간적으로 사고하는 인공지능 : 인지모형화 접근방식

GPS(general problem solver)

3. 합리적으로 사고하는 인공지능 : 사고의 법칙 접근 논리적 추론

4. 합리적으로 행동하는 인공지능 : 합리적 에이전트 접근방식

(에이전트 : 주체성을 가지고 직접 행동할 수 있는 개체)

▼ 인공지능의 역사

: ai의 탄생 → ai의 융성 → ai의 실패 → 전문가 시스템 → 신경망부활 → 확률적추론과 기계학습 → ai의 실패 → 빅데이터 → 딥러닝

- 1943-1956) ai의 탄생 : 매컬록, 피츠. 인공신경망을 제시 / 헵. 헵 학습, 인공 지능신경망을 학습가능
- 1952-1969) ai의 융성 : 마빈 민스키, 존 매카시. 최초로 인공지능 용어 사용 / 존 매카시, LISP프로그래밍 언어 / 로젠블랫, 퍼셉트론 개발
- 1966-1973) ai의 실패 : 조합폭발이슈(복잡도가 높은 문제는 대처할 수 없음) / 마빈 민스키, 세이무어 페퍼트. 인공신경망 관련 연구 중단
- 1969-1986) 전문가 시스템 : 파이겐바움, denda실험(화합물 분석)
- 1986-) 신경망의 부활 : 존 홉필드, 새로운 방식의 신경망을 제안 / hinton, rumelhart. 역전파라는 학습방법을 대중화함

- 1987-) 확률적추론과 기계학습 : 마코프 모형, 확률론과 결정이론, 베이즈망, 마르코프결정과정
- ai의 실패 : 전문가 시스템이 필요없음
- 2001 -) 빅데이터 : 왓슨, 퀴즈쇼 승리
- 2011 -) 딥러닝 : hinton, 딥러닝시스템(alexnet) / 알파고

▼ 지능형 에이전트

지능적(intelligent) = 합리적(rational)

- **에이전트** : 센서를 통해 환경을 지각(percept)하고 작동기(actuator)을 통해 환경에 어떠한 동작(action)을 수행해 환경과 상호작용(interaction)하는 존재.

자신이 기존에 가진 지식이나 지각열에만 의존해 동작함

환경 : 물리적공간 뿐만 아닌 인터넷 공간이나 소프트웨어 환경도 됨

- **지능형 에이전트(intelligent agent)**

지각한다(percept) : 특정시점에 에이전트가 감지기를 통해 어떠한 대상(내용)을 인식한다

존재의 여부 뿐만아니라 무엇인지 알아야함. 감각한다 - 존재만 알 vs 지각한다 - 내용까지 알

모든 지각열에 에이전트의 동작을 모두 매칭시켜줌

지각열(percept sequence) : 지각한 내용의 집합

에이전트함수(agent function) : 임의의 지각열(x)을 하나의 동작(y)으로 연결 짓는 것. 지각을 동작으로 매칭하는 함수. 테이블 형태로 서술할 수 있음.

- ▼ 합리적 에이전트 : 옳은 일을 하는 에이전트, 성과측정치를 극대화하는 동작을 선택해야함

성과측도(performance measure) : 환경의 주어진 임의의 상태열의 바람직함을 평가

▼ 합리성

성과측도 : 성공의 기준,

사전지식 : 환경에대한 지식,

동작 : 에이전트가 수행할 수 있는 행동세트,

지각 : 지금까지의 지각열

합리적 에이전트는 전지한(omniscient) 완벽한 에이전트가 아님. 기대성과를 극대화 하고자하나 완벽함은 실제성과를 극대화하는 것을 목표로 함

- 기대성과(expected performance) 극대화 : 정보수행(information gathering : 지각들을 수정하기 위한 행위), 학습능력(learning : 지각정보로부터 많은 것을 배움), 자율성(autonomy : 부족한 사전지식학습을 통해 스스로 보완)

▼ 과제환경(task environment) : 풀고자 하는 문제

합리적 에이전트는 해답에 해당됨

PEAS(performance environment actuators sensors)로 과제의 환경을 명확히 표현

▼ ex_PEAS

Agent Type	Performance Measure	Environment	Actuators	Sensors
Medical diagnosis system 의료진단시스템	Healthy patient, reduced costs 환자의 건강	Patient, hospital, staff 병원, 환자, 직원	Display of questions, tests, diagnoses, treatments, referrals 화면을 통해 정보 부여	Keyboard entry of symptoms, findings, patient's answers 증상입력을 위한 키보드
Satellite image analysis system	Correct image categorization	Downlink from orbiting satellite	Display of scene categorization	Color pixel arrays
Part-picking robot	Percentage of parts in correct bins	Conveyer belt with parts; bins	Jointed arm and hand	Camera, joint angle sensors
Refinery controller	Purity, yield, safety	Refinery, operators	Valves, pumps, heaters, displays	Temperature, pressure, chemical sensors
Interactive English tutor 영어선생님	Student's score on test 학생들의 시험성적	Set of students, testing agency 학생들과의 상호작용	Display of exercises, suggestions, corrections 연습문제, 제안	Keyboard input 키보드 입력

▼ 과제환경의 속성(properties) : 아래의 것들로 인해 환경의 속성이 바뀜

- 완전관측가능(fully observable) vs 부분관측가능(partially observable) : 에이전트가 매순간 환경의 완전한 상태에 접근할 수 있는가
- 단일 에이전트(single agent) vs 다중 에이전트(multiagent) : 에이전트가 환경에서 단독으로 동작하는가 아니면 에이전트가 함께 동작하는가
- 경쟁적(competitive) vs 협력적(cooperative) : 에이전트가 서로 경쟁하는 구도인가 협력하는 구도인가
- 결정론적(deterministic) vs 비결정론적(non-deterministic), 확률론적(stochastic) : 환경의 다음 상태가 현재상태와 에이전트가 수행한 행동에의해 완전히 결정되는가

확률론적이라는 것은 비결정론적이지만 비결정론적이라는 것이 곧 확률적인것은 아님

- 일화적(episodic) vs 순차적(sequential) : 경험이 누적되어 순차적으로 영향을 주는가(순차적) 아니면 하나의 경험에 하나의 일이 일어나는가(일화적)
- 정적(static) vs 동적(dynamic) - 환경 : 에이전트가 다음계획을 세우는 동안 환경이 변하는가
- 이산적(discrete) vs 연속적(continuous) : 환경의 상태, 시간의 처리방식, 에이전트의 지각과 동작이 명확하게 정의된 유한한 개수를 가지는가
- 기지(known) vs 미지(unknown) : 에이전트가 환경 내 규칙, 법칙에 대해 알고있는가

▼ ex

Task Environment	Observable	Agents	Deterministic	Episodic	Static	Discrete
체스	o	2	deterministic	sequential	static(체스를 두는 동안 환경이 변하지 않음. 새로운 규칙이 도입되면 바뀔 수 있음).	discrete(정해져있는 칸에 돌아다니는 것이므로)
포커	o	여러명	stochastic(카드가 나올 확률)	sequential	static	discrete

Task Environment	Observable	Agents	Deterministic	Episodic	Static	Discrete
Crossword puzzle	Fully	Single	Deterministic	Sequential	Static	Discrete
Chess with a clock	Fully	Multi	Deterministic	Sequential	Semi	Discrete
Poker	Partially	Multi	Stochastic	Sequential	Static	Discrete
Backgammon	Fully	Multi	Stochastic	Sequential	Static	Discrete
Taxi driving	Partially	Multi	Stochastic	Sequential	Dynamic	Continuous
Medical diagnosis	Partially	Single	Stochastic	Sequential	Dynamic	Continuous
Image analysis	Fully	Single	Deterministic	Episodic	Semi	Continuous
Part-picking robot	Partially	Single	Stochastic	Episodic	Dynamic	Continuous
Refinery controller	Partially	Single	Stochastic	Sequential	Dynamic	Continuous
Interactive English tutor	Partially	Multi	Stochastic	Sequential	Dynamic	Discrete

▼ 에이전트의 구조

에이전트 = 아키텍처 + 프로그램

인공지능에서는 에이전트함수를 구현하는 에이전트 프로그램을 설계하는 것이 중요하다

- 에이전트 아키텍처 : 에이전트 프로그램이 실행되는 감지기와 작동기를 갖춘 계산 일종의 컴퓨팅 장치

▼ 에이전트 프로그램 : 에이전트 함수를 포함한 프로그램

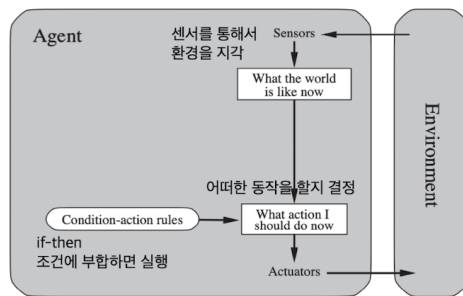
에이전트 프로그램 > 에이전트 함수

아키텍처는 일반 pc일수도 있고, 로봇차량일수도 있음. 프로그램은 이러한 아키텍처에 적합하게 설계되어야함

인공지능의 핵심은 커타란 테이블을 만들어 지각열에 대한 적절한 행동구축이 아닌 조그만한 프로그램으로 가능한 합리적인 행동을 산출할 수 있도록 프로그램을 작성하는 방법을 알아내는 것임

▼ 에이전트의 유형

- 단순 반사 에이전트(simple reflex agents)



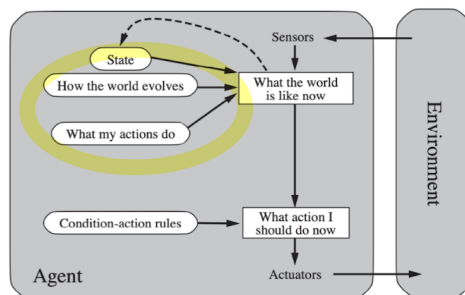
: 에이전트가 과거의 것은 고려치 않고 단순히 **현재** 지각한 상태에 기반해 행동

조건동작규칙(condition-action rules) : 조건에 부합하면 실행

- 단점) 완전히 관측가능한 환경에서만 사용가능, 부분적으로 관측가능한 환경에는 문제에 빠질 수있음.

한정된 지능을 지님. 환경에 변화가 생기면 규칙을 업데이트해야됨. 상태의 비지각적인 부분에 대한 지식이 없음

- 모형기반 반사 에이전트(model-based reflex agents)

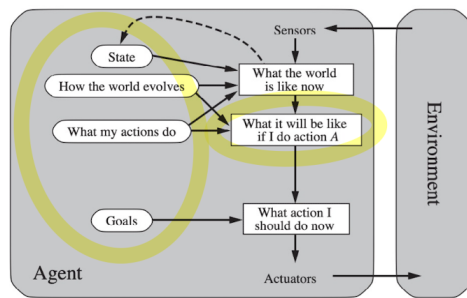


: 모형을 통해 실제 세계에대한 지식을 얻는 방식을 활용. 내부상태에 기반해 더 나은 결정을 내리는 반사적 행동을 수행

- 특징) 부분적으로 관찰가능환경에 적용가능(vs 단순반사에이전트)
- 내부상태(internal state) : 외부 환경에 대한 정보를 저장, 현재 상태와 과거정보, 추론 및 예측 정보를 담아 복잡한 세계를 표상
내부상태 갱신 - 감지기 모형, 전이모형의 조합
- 단점) 아직까지도 수동적임(인간이 관여)

- 감지기모형(sensor model) : 세계의 상태가 지각으로 반영되는 방법
- 전이모형(transition model) : 에이전트가 수행한 동작들의 효과와 에이전트와 독립적으로 세계가 진화하는 방법에 대한

- 목표기반 에이전트(goal-based agents)

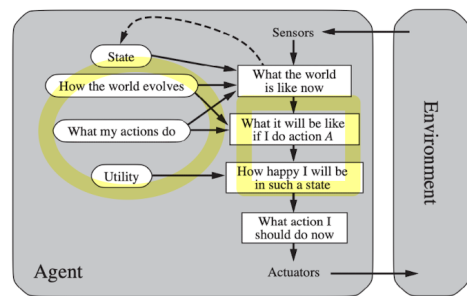


: 정확한 목표정보를 사용하는 에이전트. 목표상태에 도달할 수 있는 다양한 선택지 중 하나를 선택함

탐색, 계획수립을 활용해 동작열 (action sequence)을 찾음

- 특징) 단순 조건-동작 규칙을 기반으로한 앞선 두가지 '반사 에이전트'와 근본적으로 다름

• 효용기반 에이전트(utility-based agents)

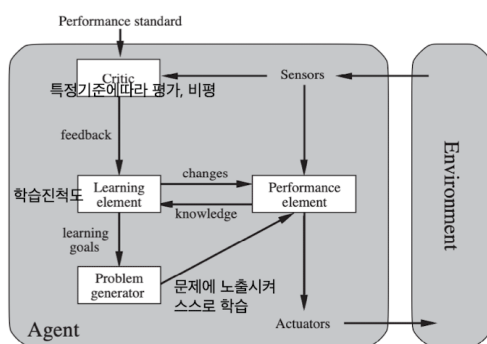


: 수행목표를 선택할때의 문제발생 소지를 해결하기위해 제안된 에이전트

효용성(utility) : 행동의 결과가 얼마나 만족스러운가

- 특징) 행동결과에 대한 기대효용(expected utility)를 극대화하는 방향으로 행동함

▼ 에이전트의 확장 : 학습하는 에이전트



지향해야하는 에이전트의 모습

학습하는 에이전트(learning agent) : 초기지식과 과거경험을 통해 학습을 진행하고 결과에따라 행동하고 적응하는 에이전트

- 배경) 앞선 4가지 에이전트는 스스로 학습하고 개선할 수 없음

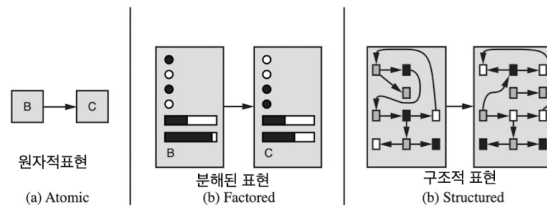
튜링, 학습하는 에이전트 제시

• 개념요소)

- 수행요소 : 현재 지식과 상태를 기반해 어떻게 행동할지 결정
- 비평 : 수행요소가 선택한 행동의 결과를 평가 및 피드백 제공
- 학습요소 : 비평을 통해 행동 개선, 변화를 유도

- 문제 생성기 : 새로운 도전 제공 → 더 많은 학습상황 제공

▼ 에이전트 프로그램 구성요소와 표현방식



- 원자적 표현(atomic representation) : 더이상 분해되지 않는 하나의 상태. 추가적인 내부구조가 없음
- 분해된 표현(factored representation) : 각 상태를 고정된 개수와 종류의 변수, 특성으로 분해. 변수들은 값을 지니고 하나의 상태는 특성값들의 벡터로 구성
- 구조적 표현(structured representation) : 객체와 그것들의 다양하고 가변적인 관계를 서술

▼ 파이썬 복습

▼ 변수와 기본 자료형

기본 자료형은 데이터를 저장하고 처리하는 가장 기본적인 유형을 의미함

1.

Number

-

int : 정수

-

float : 부동소수점

2.

String: 문자열

3. Container

-

List : 제일 많이 쓰이는 행렬 형태의 자료형 ex_ a[1,2,3]

- 위의 a에서 1을 추출하고싶은 경우 a[0], a에서 2를 추출하려면 a[1], 3은

a[2]

-

Tuple : 리스트와 비슷하지만 수정이 어려운 자료형 ex_ a(1,2,3)

-

Dictionary : Key와 Value로 구성. ex_ {'apple':200, 'google': 190, 'lg': 50}

-

Set : 순서가 없고, 중복되지 않는 값의 집합(중복 값 하나로 취급)인 자료형

ex_ {1, 2, 3}, {'apple', 'banana'}

4.

Boolean: True/False 두개의 값만 존재

```
# 변수의 선언
# 변수명 = 값

varInt = 1
varFloat = 3.14
varStr = 'a'
varList = [1,2,3]
varDict = {'name':'sookmyung', 'birth':'0303'}
```

▼ 함수

함수는 재사용의 효율을 도모하기 위해 만들어진 개념

용어) input **x** 는 파라미터(parameter)라고 함, function **f** 는 함수(function)이라 함, output **y**(=f(x))는 반환값(return value)이라 부름

```
# 함수의 선언
# def 함수명(파라미터1, 파라미터2, ...):
#     ...
#     return 반환값

# 함수의 사용
# 함수명(파라미터1, 파라미터2, ...)
def sum_func(num1, num2):
    y = num1 + num2
    return y

sum_func(1, 3)
```

▼ 얕은복사와 깊은 복사

- immutable

int, float, str, 나 bool 등 자료형은
immutable 자료형

- mutable : 변형됨

→ 깊은복사가 필요

list나 dictionary 는 대표적인
mutable 자료형
값 자체가 아닌 주소값을 복사하기 때

```

a = 3
b = a

print(a,b)

a = 2
print(a,b)

```

3 3
2 3

문에, 복사된 값이 변형되면 원래 값도 같이 변형됨

```

list_a = [1,2,3,4]
list_b = list_a
list_a[1] = 5

print(list_a, list_b)

```

[1, 5, 3, 4] [1, 5, 3, 4]

```

import copy
list_a = [1,3,5,7]
list_b = copy.deepcopy(list_a)
list_a[1] = 8

print(list_a, list_b)

```

[1, 8, 5, 7] [1, 3, 5, 7]

mutable한 자료형의 원본은 보존하면서 실제 값만 복사하고 싶은 경우에는, copy 모듈의 **deepcopy** 메소드 사용해 깊은 복사

▼ 조건문

- **if** : 비교 연산자

- $a > b$, $a < b$
- $a \Rightarrow b$, $a \leq b$
- $a == b$, $a != b$
- $a \text{ is } b$
- $a \text{ is not } b$

(is는 a와 b가 서로 같은 주소값을 참조하는 동일한 객체인지 비교. 반면 == 는 단순히 값만 비교함.)

- 논리 연산자

-

and

- **or**

- **not**

⇒ if문을 사용할 경우 논리연산자 (and , or 등)을 사용하여 깊이를 줄이는게 효율적임

- **elif**

```
#조건문 만들기
# if 조건문 :
#     ...
# else :
#     ...
a = 3
if a<5:
    print('a는 5보다 작다')
else:
    print('a는 5보다 작지않거나 같다')
```

a는 5보다 작다

• elif

```
#조건문 만들기
# if 조건문 :
#     ...
# elif 조건문 :
#     ...
# else :
#     ...
a = 3
if a<5:
    print('a는 5보다 작다')
elif a ==3 :
    print('a는 3이다')
else:
    print('a는 5보다 작지않거나 같다')
```

a는 5보다 작다

▼ 반복문

- **for**
- **while** : 조건문이 참(True)이면
안의 명령문을 계속 반복하라는 뜻
을 가진 반복 함수

break: 제어흐름 중단 (for에서도
동일하게 사용 가능)

continue: 제어흐름 유지, 코드
실행만 건너뛴

```
# for 사용해 반복문 만들기
# for 무엇 in 무엇:
#     명령문
list_a = [3,9,7,4]
for i in list_a:
    print(i)
```

3
9
7
4

```
# while 사용해 반복문 만들기
# while(조건문):
#     명령문

a = 7
while(a<10):
    a = a+1
    print(a)
```

8
9
10

- 반복문에서 많이 사용하는 함수

len(): 배열의 길이를 알려주는 함수

range(a,b): a이상 b미만의 값을 포함하는 리스트를 만드는 함수 / (만약에 1,3,5,7 .. 이렇게 2씩 크게 만들고 싶다면, range(a,b,2) 로 선언하면됨)

```
# 응용
a = range(3, 18, 2)
a = list(a)

for i in range(len(a)):
    print(a[i])
```

```
3
5
7
9
11
13
15
17
```

▼ 자료형 조작

- 리스트 조작
 - 리스트.**append(i)**: 리스트에 i 라는 값 추가
 - 리스트.**pop()**: 리스트 마지막 아 이템 지우고 리턴
 - 리스트.**reverse()**: 리스트 아이 템 거꾸로 배치
 - 리스트.**clear()**: 리스트 아이템 전부 지움
 - 리스트.**remove(i)**: 리스트에 들어있는 i라는 값 제거(여러개 있을 시 가장 앞에 하나만 지움)
 - 리스트.**sort()**: 오름차순 정렬 (숫자뿐아니라 문자도 알파벳순으로), 내림차순은 리스트.sort(reverse=True)

- 리스트 응용

```
my_list = []

my_list.append(5)
my_list.append(3)
my_list.append(8)
my_list.append(1)
my_list.append(5)
print(my_list)

last_item = my_list.pop()
print(my_list)
print(last_item)

my_list.reverse()
print(my_list)

my_list.remove(3)
print(my_list)
```

```
[5, 3, 8, 1, 5]
[5, 3, 8, 1]
5
[1, 8, 3, 5]
[1, 8, 5]
```

map(함수, 리스트): 리스트의 아이
템을 해당 함수로 처리해 다시 리스
트 만들어줌

- [식 for 변수 in 리스트]: 리스트 표
현식(for문을 돌려서 나온 값들을 식
에 넣어 리스트 생성), 뒤에 if 문이나
중복 for문도 추가 가능

- for idx, val in enumerate(리스
트): 인덱스와 값을 함께 출력하기 위
한 enumerate

```
numbers = [1,2,3,4,5]

def square(x):
    return x**2

squared_numbers = list(map(square, numbers))
print(squared_numbers)

doubled_even_numbers = [x*2 for x in numbers if x%2==0]
print(doubled_even_numbers)

for idx, val in enumerate(numbers):
    print(f"index:{idx}, Value:{val}")
```

[1, 4, 9, 16, 25]
[4, 8]
index:0, Value:1
index:1, Value:2
index:2, Value:3
index:3, Value:4
index:4, Value:5

▼ 클래스와 객체

- 클래스 : 설계도(속성 및 메소드 들어있음)
- 객체 : 설계도로 이미 만들어진 결과물 통칭, 실제물건
- 인스턴스 : 객체(들) 중 특정 클래스로 만들어진 객체, 특정 클래스로 생성된 객체

특정 객체를 생성하기 위해 '속성'와 '메소드'를 정의하는 기능

- 함수와의 차이점: 독립적인 함수가 늘어날수록 각각의 의미를 파악하기 어려워지기 때문에, 비슷한 역할을 하는 것을 한 곳에 모으는 클래스를 활용)
- 상속(새로운 클래스 만들 시 다른 클래스 기능을 물려받을 수 있음) 가능

```
# class 클래스명:
#     def __init__(self, 파라미터1, 파라미터2): #생성자
#         self.속성1 = 파라미터1
#         self.속성2 = 파라미터2
#     def 메소드(self):
#         ...

class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def greeting(self):
        print("hi, I'm {0}. I'm {1} years old. Nice to meet you.".format(self.name, self.age))

sunghyun = Person('sunghyun', 21)
sunghyun.greeting()
```

hi, I'm sunghyun. I'm 21 years old. Nice to meet you.

▼ 예외처리

- try & except

except 뒤에는 표준 예외 클래스
(파이썬 모듈에 이미 내장된 예외)를
기입
ex_ IndexError, KeyError,
ValueError, TypeError,
NameError, FileNotFoundError
등등

```
# try:
#     실행할 코드
# except 예외이름:
#     예외가 발생했을 때 처리하는 코드

try:
    x = int(0)
    y = 10/x
    print(x)
except ZeroDivisionError:
    print("0으로 나누면 안됩니다.")
```

0으로 나누면 안됩니다.

▼ 람다 표현식

lambda 매개변수: 식

- 함수를 간단히 작성
- 람다 함수 안에는 새로운 변수를
넣을 수 없으니 미리 선언해두고
사용(변수가 필요하면 일반 def
로 작성 권장)

```
# lambda 매개변수: 식

power = lambda x: x**2
power(4)
```

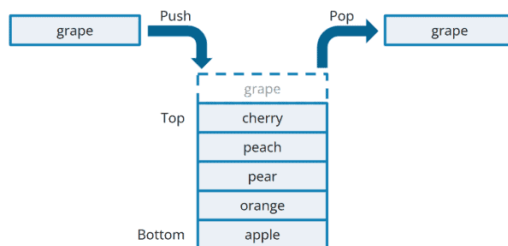
16

```
a = [1,2,3,4,5,6,7,8,9,10]
list(map(lambda x: str(x) if x==1 else float(x) if x==2 else x+10, a))
```

['1', 2.0, 13, 14, 15, 16, 17, 18, 19, 20]

▼ 자료구조 복습

▼ 스택(stack)



- **LIFO**(Last In, First Out): 가장 마지막에 들어간 데이터가 가장 먼저 빠져나가는 구조
- 데이터를 넣는것을 push, 빼내는 것을 pop
- 파이썬의 스택 구조는 기본적으로 단순 리스트만 활용하여 구현이 가능

```
stack = []

stack.append(1)
stack.append(2)
stack.append(3)

print(stack)

stack.pop()
print(stack)
```

[1, 2, 3]
[1, 2]

- **deque**(double-ended queue: 큐 앞뒤 삽입/삭제 가능)

: 양쪽 끝에서 빠르게 요소를 추가하고 제거할 수 있는 자료구조로, 스택과 큐를 구현하는 데 유용

리스트보다 요소의 추가 및 제거가 더 효율적이며, 큐나 스택 작업을 수행할 때 성능 이점이 있음

```
from collections import deque
stack = deque()

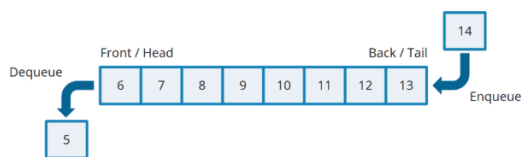
data = [1,2,3,4,5]
for i in data:
    stack.append(i)

print(stack)

stack.pop()
print(stack)
```

deque([1, 2, 3, 4, 5])
deque([1, 2, 3, 4])

▼ 큐(Queue)



```
from collections import deque

list = [1,2,3,4,5]
queue = deque(list)

queue.append(5)
print(queue)

queue.popleft()
print(queue)
```

deque([1, 2, 3, 4, 5, 5])
deque([2, 3, 4, 5, 5])

- **FIFO**(First In, First Out): 가장 먼저 들어간 데이터가 가장 먼저 빠져나가는 구조

- 큐에 자료를 넣는 것을

enqueue, 빼내는 것은 **dequeue**라고 부름

- 파이썬에서는 deque(double-ended queue: 큐 앞뒤 삽입/삭제 가능)를 사용

- **popleft** : 첫번째 데이터를 삭제

- queue.Queue는 큐 내부 요소에 직접 접근/출력 기능을 제공하지 않음.

→ get으로 하나하나 빼내는 불편한 방식을 사용해야함.

- 멀티 스레드 상황에서 안전성이 높다는 장점이 존재

-

put : 데이터를 넣음 / **empty** : 데이터가 비었는가 / **get** : 데이터를 꺼내줌

```
from queue import Queue

que = Queue()
que.put(3)
que.put(6)
que.put(9)

print(que)

while not que.empty():
    print(que.get())
```

<queue.Queue object at 0x104690890>
3
6
9

▼ 우선순위 큐(Priority Queue)

: FIFO 특성을 가진 일반 큐와 달리 추가 순서와 무관하게 **우선순위가 높은** (가장 작은 값)을 제거하는 특이한 자료 구조

- 앞서 활용한 queue 라이브러리 내 PriorityQueue 모듈 활용 가능

```
• 파이썬의 priority queue는 heapq 모듈로도 활용 가능

import heapq

heap = []

heapq.heappush(heap, (4, 'task 4'))
heapq.heappush(heap, (1, 'task 1'))
heapq.heappush(heap, (7, 'task 7'))
heapq.heappush(heap, (9, 'task 9'))

print(heap)

while heap:
    print(heapq.heappop(heap))
```

[(1, 'task 1'), (4, 'task 4'), (7, 'task 7'), (9, 'task 9')]
(1, 'task 1')
(4, 'task 4')
(7, 'task 7')
(9, 'task 9')

```
from queue import PriorityQueue

pqe = PriorityQueue()
pqe.put(4)
pqe.put(1)
pqe.put(2)
pqe.put(7)

while not pqe.empty():
    print(pqe.get())
```

1
2
4
7

- 파이썬의 priority queue는 **heapq** 모듈로도 활용 가능