『파이썬 크래시 코스』 CheatSheet

변수와 문자열

변수는 값에 붙이는 이름표입니다. 문자열이란 큰따옴표 또는 작은따옴표로 둘러싸여 있는 일련의 문자입니다. 파이썬은 f-문자열을 통해 문자열 안에서 변수를 사용해 메시지를 동적으로 만들 수 있습니다.

Hello world

```
print("Hello world!")
```

변수와 Hello world

```
msg = "Hello world!"
print(msg)
```

f-문자열 (문자열 안의 변수)

```
first_name = 'albert'
last_name = 'einstein'
full_name = f"{first_name} {last_name}"
print(full_name)
```

리스트

리스트란 일정한 순서에 따라 나열된 일련의 요소입니다. 인덱스를 통해 개별 요소에 접근하거나 리스트 전체를 순회할 수 있습니다.

리스트 생성

```
bikes = ['trek', 'redline', 'giant']
```

리스트의 첫 번째 요소

```
first_bike = bikes[0]
```

리스트의 마지막 요소

```
last_bike = bikes[-1]
```

리스트 순회하기

```
for bike in bikes:
    print(bike)
```

리스트에 요소 추가

```
bikes = []
bikes.append('trek')
bikes.append('redline')
bikes.append('giant')
```

숫자 리스트 생성

```
squares = []
for x in range(1, 11):
    squares.append(x**2)
```

리스트 내포

```
squares = [x**2 for x in range(1, 11)]
```

리스트 슬라이스

```
finishers = ['sam', 'bob', 'ada', 'bea']
first_two = finishers[:2]
```

리스트 복사

```
copy_of_bikes = bikes[:]
```

튜플

튜플은 리스트와 비슷하지만 튜플의 요소는 변경할 수 없습니다.

튜플 생성

```
dimensions = (1920, 1080)
resolutions = ('720p', '1080p', '4K')
```

if 문

if 문은 조건을 테스트하고 그에 맞게 동작합니다.

조건 테스트

```
x == 42     동등
x != 42     비동등
x > 42      초과
x >= 42     이상
x < 42      미만
x <= 42     이하
```

리스트와 조건 테스트(요소 존재 여부)

```
'trek' in bikes
'surly' not in bikes
```

불리언 값 할당

```
game_active = True
can_edit = False
```

단순한 if 테스트

```
if age >= 18:
    print("You can vote!")
```

If-elif-else 문

```
if age < 4:
    ticket_price = 0
elif age < 18:
    ticket_price = 10
elif age < 65:
    ticket_price = 40
else:
    ticket_price = 15
```

딕셔너리

딕셔너리는 정보들 사이의 연결을 저장합니다. 딕셔너리의 요소는 키-값 쌍입니다.

단순한 딕셔너리

```
alien = {'color': 'green', 'points': 5}
```

값에 접근

```
print(f"The alien's color is {alien['color']}.")
```

새로운 키-값 쌍 추가

```
alien['x_position'] = 0
```

키-값 쌍 전체 순회하기

```
fav_numbers = {'eric': 7, 'ever': 4, 'erin': 47}

for name, number in fav_numbers.items():
    print(f"{name} loves {number}.")
```

전체 키 순회하기

```
fav_numbers = {'eric': 7, 'ever': 4, 'erin': 47}

for name in fav_numbers.keys():
    print(f"{name} loves a number.")
```

전체 값 순회하기

```
fav_numbers = {'eric': 7, 'ever': 4, 'erin': 47}

for number in fav_numbers.values():
    print(f"{number} is a favorite.")
```

사용자 입력

프로그램에서 사용자에게 값을 받을 수 있습니다. 사용자 입력은 모두 문자열로 저장됩니다.

값 요청

```
name = input("What's your name? ")
print(f"Hello, {name}!")
```

숫자 요청

```
age = input("How old are you? ")
age = int(age)

pi = input("What's the value of pi? ")
pi = float(pi)
```

while 루프

while 루프는 조건을 만족하는 한 코드 블록을 계속 반복합니다. while 루프는 루프를 몇 번 실행해야 할지 미리 알 수 없을 때 특히 유용합니다.

단순한 while 루프

```
current_value = 1
while current_value <= 5:
    print(current_value)
    current_value += 1
```

사용자가 종료 시점을 선택할 수 있게 만들기

```
msg = ''
while msg != 'quit':
    msg = input("What's your message? ")
    if msg != 'quit':
        print(msg)
```

함수

함수란 이름을 가진 코드 블록이며 특정 작업을 수행하도록 만듭니다. 함수에 전달되는 정보를 인수라 부르고, 함수가 받는 정보를 매개변수라 부릅니다.

단순한 함수

```
def greet_user():
    """단순한 인사말 표시"""
    print("Hello!")

greet_user()
```

인수 전달

```
def greet_user(username):
    """개인화된 인사말 표시"""
    print(f"Hello, {username}!")

greet_user('jesse')
```

매개변수 기본 값

```
def make_pizza(topping='pineapple'):
    """토핑이 하나 있는 피자"""
    print(f"Have a {topping} pizza!")

make_pizza()
make_pizza('mushroom')
```

값 반환

```
def add_numbers(x, y):
    """두 숫자의 합을 반환"""
    return x + y

sum = add_numbers(3, 5)
print(sum)
```

클래스

클래스는 객체가 어떤 정보를 저장하며 어떻게 동작하는지 정의합니다. 클래스에 저장되는 정보를 속성이라 부르고, 클래스에 속하는 함수를 메서드라 부릅니다. 자식 클래스는 부모 클래스에서 속성과 메서드를 상속합니다.

dog 클래스 생성

```
class Dog:
    """개를 나타내는 클래스"""

    def __init__(self, name):
        """dog 객체 초기화"""
        self.name = name

    def sit(self):
        """앉기"""
        print(f"{self.name} is sitting.")

my_dog = Dog('Peso')

print(f"{my_dog.name} is a great dog!")
my_dog.sit()
```

상속

```
class SARDog(Dog):
    """탐지견 클래스"""

    def __init__(self, name):
        """sardog 초기화"""
        super().__init__(name)

    def search(self):
        """탐지"""
        print(f"{self.name} is searching.")

my_dog = SARDog('Willie')

print(f"{my_dog.name} is a search dog.")
my_dog.sit()
my_dog.search()
```

계속 연습

프로그래밍 능력이 뛰어나다면 어떤 걸 만들어보고 싶습니까?

프로그램을 배우며 계속해서 어떤 실제 프로젝트를 만들지 생각해 보는 게 좋습니다. 생각나는 걸 아이디어 노트에 기록해 뒀다가 새로운 프로젝트를 시작할 때 참조하세요.

잠시 시간을 내서 만들어 보고 싶은 프로젝트 세 개를 기록하고 설명도 추가해 보세요. 이런 아이디어와 관련이 있는 작은 프로그램을 만들 수 있게 되면 생각한 프로젝트 완성에도 한 걸음 더 가까워집니다.

파일

프로그램에서 파일을 읽고 쓸 수 있습니다. pathlib 라이브러리를 사용하면 파일과 폴더를 더 쉽게 다룰 수 있습니다. 패스를 정의하면 read_text()와 write_text() 메서드를 쓸 수 있습니다.

파일 콘텐츠 읽기

read_text() 메서드는 파일 콘텐츠 전체를 읽습니다. 그런 다음 이 텍스트를 행으로 분할하고 원하는 행에서 작업하면 됩니다.

```
from pathlib import Path

path = Path('siddhartha.txt')
contents = path.read_text()
lines = contents.splitlines()

for line in lines:
    print(line)
```

파일에 쓰기

```
path = Path('journal.txt')

msg = "I love programming.")
path.write_text(msg)
```

예외

예외는 발생할 거라 예상되는 에러에 적절히 대처하는 방법입니다. 에러가 일어날 것 같은 코드를 try 블록 안에 씁니다. 에러가 일어났을 때 실행할 코드는 except 블록 안에 씁니다. try 블록이 성공적일 때만 실행할 코드는 else 블록 안에 씁니다.

예외 캐치

```
prompt = "How many tickets do you need? "
num_tickets = input(prompt)

try:
    num_tickets = int(num_tickets)
except ValueError:
    print("Please try again.")
else:
    print("Your tickets are printing.")
```

파이썬의 선

단순한 것이 복잡한 것보다 좋습니다.

단순한 해결책과 복잡한 해결책이 있고, 둘 다 정상 동작한다면 단순한 해결책을 선택하세요. 단순한 코드는 유지 관리가 쉽고, 나중에 더 쉽게 확장할 수 있습니다.

리스트 Cheet Sheet

리스트란?

리스트란 일정한 순서에 따라 나열된 일련의 요소입니다. 리스트를 사용하면 필요한 정보를 하나로 묶어 저장할 수 있습니다. 리스트는 파이썬의 가장 강력한 기능이고, 초보자도 쉽게 사용할 수 있습니다. 또한 리스트에는 프로그래밍의 중요 개념들이 많이 포함됩니다.

리스트 정의

리스트를 정의할 때는 대괄호를 사용하며 각 요소는 콤마로 구분합니다. 리스트에는 복수형 이름을 써서 이 변수가 두 개 이상의 요소를 나타낸다는 걸 쉽게 알 수 있게 하세요.

리스트 생성

```
users = ['val', 'bob', 'mia', 'ron', 'ned']
```

요소에 접근

리스트의 각 요소는 위치에 따라 접근할 수 있으며 이 위치를 인덱스라 부릅니다. 첫 번째 요소의 인덱스는 0, 두 번째 요소의 인덱스는 1입니다. 마이너스 인덱스는 마지막 요소를 찾을 때 사용합니다. 요소에 접근할 때는 먼저 리스트 이름을 쓰고, 그 뒤에 요소 인덱스를 대괄호 안에 씁니다.

첫 번째 요소

```
first_user = users[0]
```

두 번째 요소

```
second_user = users[1]
```

마지막 요소

```
newest_user = users[-1]
```

요소 수정

리스트를 정의하고 나면 각 요소를 개별적으로 변경할 수 있습니다. 수정할 요소의 인덱스를 사용하면 됩니다.

요소 수정

```
users[0] = 'valerie'
users[1] = 'robert'
users[-2] = 'ronald'
```

요소 추가

리스트 마지막에 요소를 추가할 수도 있고, 원하는 위치에 삽입할 수도 있습니다. 따라서 기존 리스트를 수정하는 것도 가능하고, 빈 리스트로 시작한 다음 프로그램 안에서 요소를 추가할 수도 있습니다.

리스트 마지막에 요소 추가

```
users.append('amy')
```

빈 리스트로 시작하는 방법

```
users = []
users.append('amy')
users.append('val')
users.append('bob')
users.append('mia')
```

원하는 위치에 요소 삽입

```
users.insert(0, 'joe')
users.insert(3, 'bea')
```

요소 제거

위치 또는 값을 이용해 요소를 리스트에서 제거할 수 있습니다. 값을 이용해 요소를 제거하는 경우 그 값과 일치하는 첫 번째 요소만 제거됩니다.

위치를 이용해 요소 삭제

```
del users[-1]
```

값을 이용해 요소 제거

```
users.remove('mia')
```

요소 꺼내기

리스트에서 제거한 요소를 다른 동작에 사용할 때는 해당 요소를 '꺼내는' 메서드 pop()을 사용합니다.

pop()은 기본적으로 리스트의 마지막 요소를 반환하지만, 꺼낼 위치를 지정할 수도 있습니다.

마지막 요소 꺼내기

```
most_recent_user = users.pop()
print(most_recent_user)
```

첫 번째 요소 꺼내기

```
first_user = users.pop(0)
print(first_user)
```

리스트 길이

len() 함수는 리스트의 요소 개수를 반환합니다.

리스트의 길이 확인

```
num_users = len(users)
print(f"We have {num_users} users.")
```

리스트 정렬

sort() 메서드는 리스트의 순서를 영구히 변경합니다. sorted() 함수는 리스트의 사본을 만들어 정렬하므로 원래 리스트는 바뀌지 않습니다.

리스트 요소는 알파벳 순서로 정렬할 수도 있고 그 반대로 정렬할 수도 있습니다. 원래 리스트의 순서를 뒤집는 것도 가능합니다. 정렬할 때는 대소문자를 구분한다는 점에 주의하세요.

리스트를 영구히 정렬

```
users.sort()
```

리스트를 영구히 알파벳 순서 반대로 정렬

```
users.sort(reverse=True)
```

리스트를 임시로 정렬

```
print(sorted(users))
print(sorted(users, reverse=True))
```

리스트 순서 뒤집기

```
users.reverse()
```

리스트 순회하기

리스트는 수백만 개의 요소를 포함할 수 있으므로 파이썬은 리스트 요소 전체를 효율적으로 순회하는 방법을 제공합니다. 루프를 만들면 파이썬은 리스트에서 한 번에 하나씩 요소를 꺼내 임시 변수에 할당합니다. 이 임시 변수에는 단수형 이름를 써야 합니다.

들여 쓴 코드 블록이 루프 바디이며 여기서 각 요소를 사용합니다. 들여쓰지 않은 행은 루프가 완료된 뒤 실행합니다.

전체 요소 출력

```
for user in users:
    print(user)
```

전체 요소에 각각 메시지를 출력한 후 별도의 메시지 출력

```
for user in users:
    print(f"\nWelcome, {user}!")
    print("We're so glad you joined!")

print("\nWelcome, we're glad to see you all!")
```

range() 함수

range() 함수를 사용해 일정 범위의 숫자를 쉽게 생성할 수 있습니다. range() 함수는 기본적으로 0에서 시작하며 전달된 숫자를 초과하면 멈춥니다. range() 함수와 list() 함수를 사용해 숫자가 많이 들어간 리스트를 쉽게 만들 수 있습니다.

0에서 1000까지 출력

```
for number in range(1001):
    print(number)
```

1에서 1000까지 출력

```
for number in range(1, 1001):
    print(number)
```

1에서 백만까지의 숫자를 리스트로 생성

```
numbers = list(range(1, 1_000_001))
```

단순 통계

숫자 데이터로 이루어진 리스트에서 몇 가지 단순한 통계 연산을 실행할 수 있습니다.

최소 값 찾기

```
ages = [93, 99, 66, 17, 85, 1, 35, 82, 2, 77]
youngest = min(ages)
```

최대 값 찾기

```
ages = [93, 99, 66, 17, 85, 1, 35, 82, 2, 77]
oldest = max(ages)
```

합계 구하기

```
ages = [93, 99, 66, 17, 85, 1, 35, 82, 2, 77]
total_years = sum(ages)
```

리스트 슬라이스

리스트의 부분 집합을 만들 수 있습니다. 이런 부분 집합을 슬라이스라 부릅니다. 리스트에서 슬라이스를 만들 때는 먼저 첫 번째 요소의 인덱스를 쓰고, 그 다음 콜론을 쓴 다음 마지막 요소의 인덱스를 씁니다. 리스트의 처음에서 시작하려면 첫 번째 인덱스를 생략하고, 리스트의 끝까지 진행하려면 두 번째 인덱스를 생략합니다.

처음 세 개의 슬라이스

```
finishers = ['kai', 'abe', 'ada', 'gus', 'zoe']
first_three = finishers[:3]
```

중간 세 개의 슬라이스

```
middle_three = finishers[1:4]
```

마지막 세 개의 슬라이스

```
last_three = finishers[-3:]
```

리스트 복사

리스트를 복사하고 싶을 때는 첫 번째 요소에서 시작해 마지막 요소에서 끝나는 슬라이스를 만들면 됩니다. 이 방법을 사용하지 않고 단순히 변수를 복사하기만 한다면, 사본을 수정한 내용이 원본에도 반영됩니다.

리스트 복사

```
finishers = ['kai', 'abe', 'ada', 'gus', 'zoe']
copy_of_finishers = finishers[:]
```

리스트 내포

일정 범위의 숫자 또는 다른 리스트에 루프를 사용해 리스트를 만들 수 있습니다. 이런 작업은 자주 하는 일이므로 파이썬은 이를 효율적으로 수행하는 방법이 따로 있습니다. 처음에는 리스트 내포가 복잡해 보일 수 있습니다. 잘 이해가 되지 않는다면 for 루프를 사용하세요.

리스트 내포는 리스트에 저장할 값의 표현식으로 시작합니다. 그런 다음 for 루프를 써서 리스트를 생성합니다.

루프를 사용해 제곱수 리스트 생성

```
squares = []
for x in range(1, 11):
    square = x**2
    squares.append(square)
```

리스트 내포를 사용해 제곱수 리스트 생성

```
squares = [x**2 for x in range(1, 11)]
```

루프를 사용해 대문자 리스트 생성

```
names = ['kai', 'abe', 'ada', 'gus', 'zoe']

upper_names = []
for name in names:
    upper_names.append(name.upper())
```

리스트 내포를 사용해 대문자 리스트 생성

```
names = ['kai', 'abe', 'ada', 'gus', 'zoe']

upper_names = [name.upper() for name in names]
```

코드 스타일

가독성이 중요합니다.

파이썬 표기법 가이드를 따르세요.

- 들여쓰기는 공백 네 칸을 씁니다.
- 각 행은 79자 이하로 쓰세요.
- 빈 줄 하나를 써서 프로그램의 각 부분을 그룹처럼 보이게 하세요.

튜플

튜플은 리스트와 비슷하지만 일단 정의하면 그 값을 바꿀 수 없어서 프로그램이 실행되는 동안 바뀌면 안되는 정보를 저장할 때 적합합니다. 튜플은 보통 괄호를 사용해 정의합니다.

튜플은 요소의 값을 바꿀 수 없지만, 튜플 자체를 덮어쓰는 건 가능합니다.

튜플 정의

```
dimensions = (800, 600)
```

튜플 순회하기

```
for dimension in dimensions:
    print(dimension)
```

튜플 덮어쓰기

```
dimensions = (800, 600)
print(dimensions)

dimensions = (1200, 900)
print(dimensions)
```

코드 시각화

리스트 같은 데이터 구조를 처음 배울 때 파이썬이 동작하는 방식을 시각적으로 보면 도움이 됩니다. 파이썬 튜터를 보면 파이썬이 리스트에 포함된 정보를 어떻게 사용하는지 쉽게 알 수 있습니다. pythontutor.com에서 다음 코드를 실행해 보세요.

리스트 생성, 요소 출력

```
dogs = []
dogs.append('willie')
dogs.append('hootz')
dogs.append('peso')
dogs.append('goblin')

for dog in dogs:
    print(f"Hello {dog}!")
print("I love these dogs!")

print("\nThese were my first two dogs:")
old_dogs = dogs[:2]
for old_dog in old_dogs:
    print(old_dog)

del dogs[0]
dogs.remove('peso')
print(dogs)
```

딕셔너리 Cheet Sheet

딕셔너리란?

파이썬 딕셔너리는 연관된 정보를 연결해서 저장합니다. 딕셔너리의 각 정보는 키-값 쌍으로 저장됩니다. 키를 제공하면 파이썬은 그 키에 할당된 값을 반환합니다. 키-값 쌍을 순회할 수도 있고, 키나 값만 따로 순회할 수도 있습니다.

딕셔너리 정의

딕셔너리를 정의할 때는 중괄호를 사용합니다. 콜론을 써서 키와 값을 연결하고, 각 키-값 쌍은 콤마로 구분합니다.

딕셔너리 생성

```
alien_0 = {'color': 'green', 'points': 5}
```

값에 접근

값에 접근할 때는 딕셔너리 이름을 쓰고 연관된 키를 대괄호 안에 씁니다. 제공한 키가 딕셔너리에 존재하지 않으면 에러가 일어납니다.

get() 메서드도 사용할 수 있습니다. 이 메서드는 키가 존재하지 않을 때도 에러를 일으키지 않고 대신 None을 반환합니다. 딕셔너리에 키가 존재하지 않을 때 사용할 기본 값도 지정할 수 있습니다.

키와 연관된 값 가져오기

```
alien_0 = {'color': 'green', 'points': 5}

print(alien_0['color'])
print(alien_0['points'])
```

get()으로 값 가져오기

```
alien_0 = {'color': 'green'}

alien_color = alien_0.get('color')
alien_points = alien_0.get('points', 0)
alien_speed = alien_0.get('speed')

print(alien_color)
print(alien_points)
print(alien_speed)
```

키-값 쌍 추가

컴퓨터의 메모리가 허락하는 한 딕셔너리에 키-값 쌍을 원하는 만큼 저장할 수 있습니다. 기존 딕셔너리에 키-값 쌍을 새로 추가할 때는 딕셔너리 이름을 쓰고 대괄호 안에 키를 쓴 후 새 값을 지정하면 됩니다.

이 방식을 사용하면 빈 딕셔너리로 시작해서 키-값 쌍을 추가하는 방식으로 딕셔너리를 만들 수 있습니다.

키-값 쌍 추가

```
alien_0 = {'color': 'green', 'points': 5}

alien_0['x'] = 0
alien_0['y'] = 25
alien_0['speed'] = 1.5
```

빈 딕셔너리로 시작

```
alien_0 = {}
alien_0['color'] = 'green'
alien_0['points'] = 5
```

값 수정

딕셔너리의 값을 수정할 수도 있습니다. 수정할 때는 딕셔너리 이름 다음에 대괄호 안에 키를 쓰고 새 값을 지정합니다.

딕셔너리의 값 수정

```
alien_0 = {'color': 'green', 'points': 5}
print(alien_0)

# 외계인의 색깔과 점수 변경
alien_0['color'] = 'yellow'
alien_0['points'] = 10
print(alien_0)
```

키-값 쌍 제거

딕셔너리에서 원하는 키-값 쌍을 제거할 수 있습니다. del 키워드 다음에 딕셔너리 이름을 쓰고, 그 다음에 대괄호 안에 키를 쓰면 됩니다. 이렇게 하면 키, 그와 연관된 값이 삭제됩니다.

키 값 쌍 제거

```
alien_0 = {'color': 'green', 'points': 5}
print(alien_0)

del alien_0['points']
print(alien_0)
```

딕셔너리 시각화

딕셔너리 예제 중 일부를 pythontutor.com에서 실행해 보세요.

딕셔너리 순회하기

딕셔너리는 키-값 쌍 전체, 키 전체, 값 전체 세 가지 방식으로 순회할 수 있습니다.

딕셔너리는 키-값 쌍이 추가된 순서를 저장합니다. 추가된 순서와 다른 순서로 순회할 때는 sorted() 함수를 사용하세요.

키-값 쌍 전체 순회하기

```
# 사람들이 좋아하는 언어 저장
fav_languages = {
    'jen': 'python',
    'sarah': 'c',
    'edward': 'ruby',
    'phil': 'python',
    }

# 사람 이름과 좋아하는 언어 출력
for name, language in fav_languages.items():
    print(f"{name}: {language}")
```

키 순회하기

```
# 설문조사에 참가한 사람의 이름 출력
for name in fav_languages.keys():
    print(name)
```

전체 값 순회하기

```
# 선택된 언어 모두 출력
for language in fav_languages.values():
    print(language)
```

모든 키를 역순으로 순회하기

```
# 각 사람과 좋아하는 언어를 출력하되
# 사람 이름의 역순으로 정렬
for name in sorted(fav_languages.keys(),
        reverse=True):
    language = fav_languages[name]
    print(f"{name}: {language}")
```

딕셔너리 길이

len() 함수를 사용해 딕셔너리에 있는 키-값 쌍 개수를 알 수 있습니다.

딕셔너리 길이 확인

```
num_responses = len(fav_languages)
```

중첩: 딕셔너리로 구성된 리스트

딕셔너리들을 리스트에 저장하는 게 적절할 때도 있습니다. 이를 중첩이라 부릅니다.

리스트에 딕셔너리 저장

```
# 빈 리스트로 시작
users = []

# 새로운 사용자를 생성해 리스트에 추가
new_user = {
    'last': 'fermi',
    'first': 'enrico',
```

```
    'username': 'efermi',
    }
users.append(new_user)

# 또 다른 사용자를 생성해 리스트에 추가
new_user = {
    'last': 'curie',
    'first': 'marie',
    'username': 'mcurie',
    }
users.append(new_user)

# 각 사용자의 정보 출력
print("User summary:")
for user_dict in users:
    for k, v in user_dict.items():
        print(f"{k}: {v}")
    print("\n")
```

append()를 쓰지 않고 직접 딕셔너리들을 리스트로 만들 수도 있습니다.

```
# 사용자 리스트 정의
# 각 사용자는 딕셔너리로 표현
users = [
    {
        'last': 'fermi',
        'first': 'enrico',
        'username': 'efermi',
    },
    {
        'last': 'curie',
        'first': 'marie',
        'username': 'mcurie',
    },
]

# 각 사용자의 정보 출력
print("User summary:")
for user_dict in users:
    for k, v in user_dict.items():
        print(f"{k}: {v}")
    print("\n")
```

중첩: 딕셔너리에 포함된 리스트

딕셔너리 내부에 리스트를 사용하면 키 하나에 여러 개의 값을 연결할 수 있습니다.

딕셔너리에 리스트 저장

```
# 각 사용자에 대해 여러 가지 언어를 저장
fav_languages = {
    'jen': ['python', 'ruby'],
    'sarah': ['c'],
    'edward': ['ruby', 'go'],
    'phil': ['python', 'haskell'],
}

# 모든 응답 출력
for name, langs in fav_languages.items():
    print(f"{name}: ")
    for lang in langs:
        print(f"- {lang}")
```

중첩: 딕셔너리로 구성된 딕셔너리

딕셔너리 안에 다른 딕셔너리를 저장할 수도 있습니다. 이런 경우 각 키와 연결되는 '값' 또한 딕셔너리입니다.

딕셔너리에 딕셔너리 저장

```
users = {
    'aeinstein': {
        'first': 'albert',
        'last': 'einstein',
        'location': 'princeton',
        },
    'mcurie': {
        'first': 'marie',
        'last': 'curie',
        'location': 'paris',
        },
    }

for username, user_dict in users.items():
    full_name = f"{user_dict['first']} "
    full_name += user_dict['last']

    location = user_dict['location']

    print(f"\nUsername: {username}")
    print(f"\tFull name: {full_name.title()}")
    print(f"\tLocation: {location.title()}")
```

중첩 레벨

특정 상황에서는 중첩이 대단히 유용합니다. 하지만 코드가 지나치게 복잡해지는 건 피하세요. 세 단계 이상 중첩하는 건 피하고 클래스 등을 사용해 데이터를 더 단순하게 관리할 방법을 찾으세요.

딕셔너리 내포

리스트 내포와 비슷하게 딕셔너리에도 내포 문법이 있습니다. 저장할 키-값 쌍의 표현식을 먼저 정의하고 for 루프를 써서 그 표현식에 값을 공급하세요.

zip() 함수는 두 리스트의 요소를 연결합니다. 이 함수는 딕셔너리에도 쓸 수 있습니다.

루프를 사용해 딕셔너리 생성

```
squares = {}
for x in range(5):
    squares[x] = x**2
```

딕셔너리 내포 사용

```
squares = {x:x**2 for x in range(5)}
```

zip()으로 딕셔너리 생성

```
group_1 = ['kai', 'abe', 'ada', 'gus', 'zoe']
group_2 = ['jen', 'eva', 'dan', 'isa', 'meg']

pairings = {name:name_2
    for name, name_2 in zip(group_1, group_2)}
```

수백만 개의 딕셔너리 생성

비슷한 데이터로 시작하는 딕셔너리를 대량 생성할 때는 루프를 사용하세요.

백만 외계인

```
aliens = []

# 각각 5점인 녹색 외계인 백만개 생성
#
for alien_num in range(1_000_000):
    new_alien = {
        'color': 'green',
        'points': 5,
        'x': 20 * alien_num,
        'y': 0
    }

    aliens.append(new_alien)

# 리스트에 실제로 외계인 백만개가 있는지 확인
num_aliens = len(aliens)

print("Number of aliens created:")
print(num_aliens)
```

if 문과 while 루프 Cheet Sheet

if 문이란? while 루프란?

if 문은 프로그램의 현재 상태를 테스트하고 그에 맞게 반응합니다. 조건 하나만 확인하는 단순한 if 문을 만들 수도 있고, 여러 개의 블록을 연결해 복잡한 조건을 검사할 수도 있습니다.

while 루프는 특정 조건을 만족하는 한 계속 실행됩니다. while 루프를 사용하면 사용자가 원하는 만큼 프로그램을 실행하게 만들 수 있습니다.

조건 테스트

조건 테스트는 True나 False로 평가되는 표현식입니다. 파이썬은 True와 False 값을 사용해 if 문의 코드 실행을 결정합니다.

동일성 체크

등호 하나는 변수에 값을 할당합니다. 두 개의 값이 동일한지 확인할 때는 등호 두 개를 사용합니다.
조건 테스트가 의도한 대로 동작하지 않는다면 실수로 등호를 하나만 쓴 건 아닌지 확인하세요.

```
>>> car = 'bmw'
>>> car == 'bmw'
True

>>> car = 'audi'
>>> car == 'bmw'
False
```

비교할 때 대소문자 무시

```
>>> car = 'Audi'
>>> car.lower() == 'audi'
True
```

비동일성 체크

```
>>> topping = 'mushrooms'
>>> topping != 'anchovies'
True
```

숫자 비교

숫자 값 테스트도 문자열 값 테스트와 비슷합니다.

동일성과 비동일성 체크

```
>>> age = 18
>>> age == 18
True

>>> age != 18
False
```

비교 연산자

```
>>> age = 19
>>> age < 21
True

>>> age <= 21
True

>>> age > 21
False

>>> age >= 21
False
```

여러 가지 조건 체크

동시에 여러 가지 조건을 체크할 수도 있습니다. and 연산자는 모든 조건을 만족할 때 True를 반환합니다. or 연산자는 조건 중 하나만 만족하면 True를 반환합니다.

and로 여러 조건 체크

```
>>> age_0 = 22
>>> age_1 = 18
>>> age_0 >= 21 and age_1 >= 21
False

>>> age_1 = 23
>>> age_0 >= 21 and age_1 >= 21
True
```

or로 여러 조건 체크

```
>>> age_0 = 22
>>> age_1 = 18
>>> age_0 >= 21 or age_1 >= 21
True

>>> age_0 = 18
>>> age_0 >= 21 or age_1 >= 21
False
```

불리언 값

불리언 값은 True 또는 False입니다. 불리언 값은 대개 프로그램 안에서 특정 조건을 저장할 때 사용합니다.

단순한 불리언 값

```
game_active = True
is_valid = True
can_edit = False
```

if 문

if 문에는 여러 가지 종류가 있습니다. 그 중 무엇을 사용할지는 테스트할 조건의 수에 따라 다릅니다. elif 블록은 필요한 만큼 쓸 수 있고 else 블록은 항상 옵션입니다.

단순한 if 문

```
age = 19

if age >= 18:
    print("You're old enough to vote!")
```

if-else 문

```
age = 17

if age >= 18:
    print("You're old enough to vote!")
else:
    print("You can't vote yet.")
```

if-elif-else 문

```
age = 12

if age < 4:
    price = 0
elif age < 18:
    price = 25
else:
    price = 40

print(f"Your cost is ${price}.")
```

리스트와 조건 테스트(요소 존재 여부)

리스트에 특정 값이 존재하는지 쉽게 확인할 수 있습니다. 또한 리스트를 순회하기 전에 비어 있는지도 확인할 수 있습니다.

값이 리스트에 존재하는지 확인

```
>>> players = ['al', 'bea', 'cyn', 'dale']
>>> 'al' in players
True

>>> 'eric' in players
False
```

두 개의 값이 리스트에 존재하는지 확인

```
>>> 'al' in players and 'cyn' in players
```

값이 리스트에 존재하지 않는지 확인

```
banned_users = ['ann', 'chad', 'dee']
user = 'erin'
```

```
if user not in banned_users:
    print("You can play!")
```

리스트가 비어 있는지 확인

빈 리스트는 if 문에서 False로 평가됩니다.

```
players = []

if players:
    for player in players:
        print(f"Player: {player.title()}")
else:
    print("We have no players yet!")
```

입력 받기

input() 함수를 사용해 사용자의 입력을 받을 수 있습니다. 입력은 모두 문자열로 저장됩니다. 숫자 값을 받을 때는 값을 받은 후 숫자 타입으로 변환해야 합니다.

단순한 입력

```
name = input("What's your name? ")
print(f"Hello, {name}.")
```

int()로 문자열을 숫자로 변환

```
age = input("How old are you? ")
age = int(age)

if age >= 18:
    print("\nYou can vote!")
else:
    print("\nSorry, you can't vote yet.")
```

float()를 사용해 숫자 값 받기

```
tip = input("How much do you want to tip? ")
tip = float(tip)
print(f"Tipped ${tip}.")
```

while 루프

while 루프는 조건을 만족하는 한 코드 블록을 계속 반복합니다.

5까지 세기

```
current_number = 1

while current_number <= 5:
    print(current_number)
    current_number += 1
```

사용자가 종료 시점을 선택할 수 있게 만들기

```
prompt = "\nTell me something, and I'll "
prompt += "repeat it back to you."
prompt += "\nEnter 'quit' to end the program. "

message = ""
while message != 'quit':
    message = input(prompt)
    if message != 'quit':
        print(message)
```

플래그

플래그는 오래 실행되는 프로그램에서 다른 일부분의 코드를 사용해 루프 종료를 결정할 때 적합합니다.

```
prompt = "\nTell me something, and I'll "
prompt += "repeat it back to you."
prompt += "\nEnter 'quit' to end the program. "

active = True
while active:
    message = input(prompt)

    if message == 'quit':
        active = False
    else:
        print(message)
```

break를 사용해 루프 종료

```
prompt = "\nWhat cities have you visited?"
prompt += "\nEnter 'quit' when you're done. "

while True:
    city = input(prompt)
    if city == 'quit':
        break
    else:
        print(f"I've been to {city}!")
```

서브라임 텍스트와 사용자 입력

서브라임 텍스트를 포함해 상당수의 텍스트 에디터는 프로그램 안에서 사용자 입력을 받을 수 없습니다. 물론 이런 에디터로 프로그램을 작성할 수는 있지만 실행은 터미널에서 해야 합니다.

루프 종료

파이썬 루프는 모두 break 문과 continue 문을 사용할 수 있습니다. 예를 들어 리스트나 딕셔너리를 순회하는 for 루프를 break로 종료할 수 있습니다. 리스트나 딕셔너리를 순회할 때 continue 문을 사용해 일부 요소를 건너 뛸 수 있습니다.

루프에서 continue 사용

루프에서 continue 사용

```
banned_users = ['eve', 'fred', 'gary', 'helen']

prompt = "\nAdd a player to your team."
prompt += "\nEnter 'quit' when you're done. "

players = []
while True:
    player = input(prompt)
    if player == 'quit':
        break
    elif player in banned_users:
        print(f"{player} is banned!")
        continue
    else:
        players.append(player)

print("\nYour team:")
for player in players:
    print(player)
```

무한 루프 방지

while 루프에는 루프가 영원히 실행되지 않도록 만들 방법이 반드시 필요합니다. 조건을 항상 만족하게 되면 루프는 절대 종료되지 않습니다. 무한 루프가 발생했다면 ctrl + C를 눌러 중지할 수 있습니다.

무한 루프

```
while True:
    name = input("\nWho are you? ")
    print(f"Nice to meet you, {name}!")
```

리스트에서 특정 값 모두 제거

remove() 메서드는 리스트에서 특정 값을 제거하지만, 하나를 제거하면 실행을 멈춥니다. 특정 값을 모두 제거하려면 while 루프와 함께 사용하세요.

반려동물 리스트에서 고양이를 모두 제거

```
pets = ['dog', 'cat', 'dog', 'fish', 'cat',
        'rabbit', 'cat']
print(pets)

while 'cat' in pets:
    pets.remove('cat')

print(pets)
```

함수 Cheet Sheet

함수란?

함수는 이름 붙은 코드 블록이며 한 가지 작업을 하도록 설계합니다. 함수를 만들어 뒀다가 같은 작업을 할 때마다 실행하면 됩니다.

함수는 필요한 정보를 받고, 작업 결과를 반환할 수 있습니다. 함수를 효율적으로 사용하면 프로그램을 더 쉽게 작성하고, 읽고, 테스트하고, 관리할 수 있습니다.

함수 정의

함수의 첫 행에는 def 키워드를 써서 함수를 정의한다고 선언합니다. 키워드 다음에 함수 이름과 괄호를 쓰고 콜론으로 끝냅니다. 다음 행에는 함수가 하는 일을 설명하는 독스트링을 씁니다. 독스트링은 따옴표 세 쌍으로 감쌉니다. 함수 바디는 한 단계 들여 씁니다.

함수를 호출할 때는 함수 이름 다음에 괄호를 붙이면 됩니다.

함수 정의

```
def greet_user():
    """단순한 인사말 표시"""
    print("Hello!")

greet_user()
```

함수에 정보 전달

함수에 전달하는 정보를 인수라 부릅니다. 함수가 받는 정보를 매개변수라 부릅니다. 인수는 함수를 호출하는 괄호 안에 쓰고, 매개변수는 함수 정의의 괄호 안에 씁니다.

인수 한 개 전달

```
def greet_user(username):
    """단순한 인사말 표시"""
    print(f"Hello, {username}!")

greet_user('jesse')
greet_user('diana')
greet_user('brandon')
```

위치 인수와 키워드 인수

인수에는 위치 인수와 키워드 인수 두 종류가 있습니다. 위치 인수를 사용하면 파이썬은 함수 호출의 첫 번째 인수를 첫 번째 매개변수와 연결하는 식으로 동작합니다.

키워드 인수는 함수를 호출할 때 어떤 인수가 어떤 매개변수에 할당되는지 직접 지정합니다. 키워드 인수에서는 인수 순서가 중요하지 않습니다.

위치 인수

```
def describe_pet(animal, name):
    """반려동물 정보 표시"""
    print(f"\nI have a {animal}.")
    print(f"Its name is {name}.")

describe_pet('hamster', 'harry')
describe_pet('dog', 'willie')
```

키워드 인수

```
def describe_pet(animal, name):
    """반려동물 정보 표시"""
    print(f"\nI have a {animal}.")
    print(f"Its name is {name}.")

describe_pet(animal='hamster', name='harry')
describe_pet(name='willie', animal='dog')
```

기본 값

매개변수에 기본 값을 지정할 수 있습니다. 함수를 호출할 때 해당 인수를 생략하면 기본 값이 사용됩니다. 함수 정의에서 기본 값이 있는 매개변수를 사용할 때는 반드시 기본 값이 없는 매개변수들 다음에 써야 정확히 동작합니다.

기본 값

```
def describe_pet(name, animal='dog'):
    """반려동물 정보 표시"""
    print(f"\nI have a {animal}.")
    print(f"Its name is {name}.")

describe_pet('harry', 'hamster')
describe_pet('willie')
```

None을 사용하면 해당 인수는 옵션이 됩니다.

```
def describe_pet(animal, name=None):
    """반려동물 정보 표시"""
    print(f"\nI have a {animal}.")
    if name:
        print(f"Its name is {name}.")

describe_pet('hamster', 'harry')
describe_pet('snake')
```

값 반환

함수는 값을 반환할 수 있습니다. 값을 반환하는 함수를 호출할 때는 해당 행에 그 값이 할당될 변수를 써야 합니다. 함수는 return 문을 만나면 실행을 중지합니다.

값 하나 반환

```
def get_full_name(first, last):
    """보기 좋게 표시한 실명을 반환"""
    full_name = f"{first} {last}"
    return full_name.title()

musician = get_full_name('jimi', 'hendrix')
print(musician)
```

딕셔너리 반환

```
def build_person(first, last):
    """사람의 정보를 담은 딕셔너리 반환"""
    person = {'first': first, 'last': last}
    return person

musician = build_person('jimi', 'hendrix')
print(musician)
```

옵션인 값을 포함하는 딕셔너리 반환

```
def build_person(first, last, age=None):
    """사람의 정보를 담은 딕셔너리 반환"""
    person = {'first': first, 'last': last}
    if age:
        person['age'] = age

    return person

musician = build_person('jimi', 'hendrix', 27)
print(musician)

musician = build_person('janis', 'joplin')
print(musician)
```

함수 시각화

여기서 소개한 예제 몇 가지와 여러분이 만든 함수를 pythontutor.com에서 실행해 보세요.

함수에 리스트 전달

함수 인수로 리스트를 전달할 수 있고, 함수는 받은 리스트의 값을 사용할 수 있습니다. 함수 안에서 리스트를 변경하면 원본 리스트에도 반영됩니다. 리스트의 사본을 만들어 인수로 전달하면 함수가 리스트를 변경하는 일을 막을 수 있습니다.

리스트를 인수로 사용

```
def greet_users(names):
    """모든 이에게 단순한 인사말 출력"""
    for name in names:
        msg = f"Hello, {name}!"
        print(msg)

usernames = ['hannah', 'ty', 'margot']
greet_users(usernames)
```

함수가 리스트를 변경하도록 허용

다음 예제는 모델 리스트를 함수에 전달해 출력합니다. 첫 번째 리스트는 계속 줄어들고, 두 번째 리스트는 그에 따라 채워집니다.

```
def print_models(unprinted, printed):
    """모델을 3D로 출력"""
    while unprinted:
        current_model = unprinted.pop()
        print(f"Printing {current_model}")
        printed.append(current_model)

# 출력하지 않은 설계도를 저장하고 출력

unprinted = ['phone case', 'pendant', 'ring']
printed = []
print_models(unprinted, printed)

print(f"\nUnprinted: {unprinted}")
print(f"Printed: {printed}")
```

함수가 리스트를 수정하지 못하게 방지

다음 예제는 이전 예제와 같지만, print_models()를 호출해도 원본 리스트는 그대로입니다.

```
def print_models(unprinted, printed):
    """모델을 3D로 출력"""
    while unprinted:
        current_model = unprinted.pop()
        print(f"Printing {current_model}")
        printed.append(current_model)

# 출력하지 않은 설계도를 저장하고 출력

original = ['phone case', 'pendant', 'ring']
printed = []

print_models(original[:], printed)
print(f"\nOriginal: {original}")
print(f"Printed: {printed}")
```

임의의 개수인 인수 전달

함수가 인수 몇 개를 받게 될지 모를 때도 있습니다. 이럴 때 * 연산자를 사용해 매개변수를 정의하면 전달받는 인수들을 하나로 모을 수 있습니다. 이렇게 임의의 인수를 받는 매개변수는 반드시 마지막에 정의해야 합니다. 이런 매개변수에는 보통 *args라는 이름을 씁니다.

** 연산자는 매개변수가 키워드 인수를 수집하게 만듭니다. 이렇게 수집된 인수들은 이름을 키, 인수 값을 값으로 딕셔너리에 저장됩니다. 이런 매개변수에는 보통 **kwargs라는 이름을 씁니다.

인수 수집

```
def make_pizza(size, *toppings):
    """피자 조리"""
    print(f"\nMaking a {size} pizza.")

    print("Toppings:")
    for topping in toppings:
        print(f"- {topping}")

# 토핑이 서로 다른 세 개의 피자
make_pizza('small', 'pepperoni')
make_pizza('large', 'bacon bits', 'pineapple')
make_pizza('medium', 'mushrooms', 'peppers',
        'onions', 'extra cheese')
```

키워드 인수 수집

```
def build_profile(first, last, **user_info):
    """사용자 딕셔너리 생성"""
    user_info['first'] = first
    user_info['last'] = last

    return user_info

# 정보가 서로 다른 두 사용자

user_0 = build_profile('albert', 'einstein',
        location='princeton')

user_1 = build_profile('marie', 'curie',
        location='paris', field='chemistry')
print(user_0)
print(user_1)
```

함수를 만드는 가장 좋은 방법

함수를 작성하고 호출하는 방식은 여러 가지입니다. 시작할 때는 우선 동작만 하는 걸 목표로 하세요. 경험이 쌓이면 위치 인수와 키워드 인수의 미묘한 장단점, 함수를 임포트하는 방식의 차이 등을 더 깊이 이해하게 될 겁니다. 일단 지금은 함수가 의도한 대로 실행되기만 한다면, 잘 하고 있는 겁니다.

모듈

함수를 모듈이라 부르는 별도 파일에 저장하고, 그 파일을 메인 프로그램에서 불러올 수 있습니다. 이렇게 하면 프로그램이 더 깔끔해집니다. 모듈은 메인 프로그램 파일과 같은 폴더에 저장하세요.

함수를 모듈에 저장

```
pizza.py
def make_pizza(size, *toppings):
    """피자 조리"""
    print(f"\nMaking a {size} pizza.")
    print("Toppings:")
    for topping in toppings:
        print(f"- {topping}")
```

모듈 전체 임포트

making_pizzas.py 모듈의 모든 함수는 메인 프로그램에서 사용할 수 있습니다.

```
import pizza

pizza.make_pizza('medium', 'pepperoni')
pizza.make_pizza('small', 'bacon', 'pineapple')
```

특정 함수 임포트

프로그램에서 임포트한 함수만 사용할 수 있습니다.

```
from pizza import make_pizza

make_pizza('medium', 'pepperoni')
make_pizza('small', 'bacon', 'pineapple')
```

모듈에 별칭 부여

```
import pizza as p

p.make_pizza('medium', 'pepperoni')
p.make_pizza('small', 'bacon', 'pineapple')
```

함수에 별칭 부여

```
from pizza import make_pizza as mp

mp('medium', 'pepperoni')
mp('small', 'bacon', 'pineapple')
```

모듈의 모든 함수 임포트

되도록 이렇게 하지 마세요. 이름 충돌로 인해 에러가 일어날 수 있습니다. 다른 사람의 모듈을 활용할 수 있다는 것만 기억해 두세요.

```
from pizza import *
make_pizza('medium', 'pepperoni')

make_pizza('small', 'bacon', 'pineapple')
```

클래스 Cheet Sheet

클래스란?

클래스는 객체 지향 프로그래밍의 기반입니다. 클래스는 개, 자동차, 로봇 등 실제 사물을 프로그램에서 표현할 때 사용합니다. 클래스를 사용해 만든 객체는 개별 개, 자동차, 로봇에 해당하는 인스턴스입니다. 클래스는 여러 객체가 공통으로 가지는 동작과 정보를 정의합니다.

클래스는 서로 상속할 수 있습니다. 즉, 기존 클래스의 기능을 확장하는 클래스를 만들 수 있습니다. 이를 잘 이용하면 다양한 상황에서 효율적으로 작업할 수 있습니다. 여러분이 직접 클래스를 만들지 않더라도 다른 사람이 만든 클래스를 사용할 일은 많을 겁니다.

클래스 생성과 사용

자동차를 표현한다고 합시다. 자동차에 어떤 정보와 동작을 연결해야 할까요? 정보는 속성이라 부르는 변수에 할당되고 동작은 함수로 표현합니다. 클래스에 속하는 함수를 메서드라 부릅니다.

Car 클래스

```
class Car:
    """자동차를 표현하는 클래스"""

    def __init__(self, make, model, year):
        """속성 초기화"""
        self.make = make
        self.model = model
        self.year = year

        # 급유량은 갤런 단위
        self.fuel_capacity = 15
        self.fuel_level = 0

    def fill_tank(self):
        """완전 급유"""
        self.fuel_level = self.fuel_capacity
        print("Fuel tank is full.")

    def drive(self):
        """운행 시뮬레이션"""
        print("The car is moving.")
```

클래스에서 인스턴스 생성

```
my_car = Car('audi', 'a4', 2021)
```

속성 값에 접근

```
print(my_car.make)
print(my_car.model)
print(my_car.year)
```

메서드 호출

```
my_car.fill_tank()
my_car.drive()
```

여러 인스턴스 생성

```
my_car = Car('audi', 'a4', 2024)
my_old_car = Car('subaru', 'outback', 2018)
my_truck = Car('toyota', 'tacoma', 2020)
my_old_truck = Car('ford', 'ranger', 1999)
```

속성 수정

속성은 직접 수정할 수도 있고, 메서드를 만들어서 더 신중하게 수정할 수도 있습니다. 메서드를 만들면 속성에 알맞는 값이 할당되는지 테스트할 수 있습니다.

속성 직접 수정

```
my_new_car = Car('audi', 'a4', 2024)
my_new_car.fuel_level = 5
```

속성을 수정하는 메서드

```
def update_fuel_level(self, new_level):
    """급유량 업데이트"""
    if new_level <= self.fuel_capacity:
        self.fuel_level = new_level
    else:
        print("The tank can't hold that much!")
```

속성 값을 증가시키는 메서드

```
def add_fuel(self, amount):
    """연료 급유"""
    if (self.fuel_level + amount
            <= self.fuel_capacity):
        self.fuel_level += amount
        print("Added fuel.")
    else:
        print("The tank won't hold that much.")
```

명명법

파이썬에서는 클래스 이름을 카멜 표기법 형태로 쓰고 객체 이름은 밑줄을 포함해 소문자로 쓰는 게 원칙입니다. 클래스를 포함하는 모듈 역시 밑줄을 포함하는 소문자 이름을 씁니다.

클래스 상속

다른 클래스를 더 세분화하는 클래스를 만들 때는 상속을 이용합니다. 다른 클래스를 상속하는 클래스는 부모 클래스의 속성과 메서드를 자동으로 상속합니다. 자식 클래스에는 자유롭게 새로운 속성이나 메서드를 정의할 수 있고, 부모 클래스의 속성이나 메서드를 덮어쓸 수도 있습니다.

다른 클래스를 상속할 때는 클래스 정의의 괄호 안에 부모 클래스 이름을 씁니다.

자식 클래스의 __init__() 메서드

```
class ElectricCar(Car):
    """전기차 클래스"""

    def __init__(self, make, model, year):
        """전기차 초기화"""
        super().__init__(make, model, year)

        # 전기차에만 해당하는 속성
        # 배터리는 kWh 단위
        self.battery_size = 40

        # 충전량은 % 단위
        self.charge_level = 0
```

자식 클래스에 메서드 추가

```
class ElectricCar(Car):
    --생략--

    def charge(self):
        """완충"""
        self.charge_level = 100
        print("The vehicle is fully charged.")
```

자식 메서드와 부모 메서드 모두 사용

```
my_ecar = ElectricCar('nissan', 'leaf', 2024)

my_ecar.charge()
my_ecar.drive()
```

자신만의 워크플로 찾기

현실의 사물을 코드로 표현하는 방법은 헤아릴 수 없이 많고, 때로는 그 다양성이 벅차게 느껴질 수 있습니다. 방법은 자유롭게 택해도 됩니다. 시도한 방법이 잘 어울리지 않으면 다른 방법을 시도하세요.

부모 메서드 오버라이드

```
class ElectricCar(Car):
    --생략--

    def fill_tank(self):
        """에러 메시지 표시"""
        print("This car has no fuel tank!")
```

인스턴스를 속성으로

클래스 속성에 객체를 쓸 수도 있습니다. 이렇게 하면 클래스를 더 구체적인 사물이나 개념으로 표현할 수 있습니다.

Battery 클래스

```
class Battery:
    """전기차의 배터리"""

    def __init__(self, size=85):
        """배터리 속성 초기화"""
        # 용량은 kWh, 충전량은 % 단위
        self.size = size
        self.charge_level = 0

    def get_range(self):
        """운전 거리 반환"""
        if self.size == 40:
            return 150
        elif self.size == 65:
            return 225
```

인스턴스를 속성으로 사용

```
class ElectricCar(Car):
    --생략--

    def __init__(self, make, model, year):
        """전기차 초기화"""
        super().__init__(make, model, year)
        # 전기차에만 해당하는 속성
        self.battery = Battery()
    def charge(self):
        """완충"""
        self.battery.charge_level = 100
        print("The vehicle is fully charged.")
```

인스턴스 사용

```
my_ecar = ElectricCar('nissan', 'leaf', 2024)

my_ecar.charge()
print(my_ecar.battery.get_range())
my_ecar.drive()
```

클래스 임포트

클래스 파일에 정보와 기능을 상세하게 추가할수록 파일이 커집니다. 프로그램 파일이 지나치게 복잡해지는 걸 막으려면 클래스를 모듈로 저장하고 메인 프로그램에서 임포트하는 게 좋습니다.

클래스를 파일에 저장

car.py

```
"""일반 차와 전기차"""
class Car:
    """자동차를 표현하는 클래스"""
    --snip--

class Battery:
    """전기차의 배터리"""
    --생략--

class ElectricCar(Car):
    """전기차 클래스"""
    --생략--
```

모듈에서 클래스 임포트

my_cars.py

```
from car import Car, ElectricCar

my_beetle = Car('volkswagen', 'beetle', 2021)
my_beetle.fill_tank()
my_beetle.drive()

my_leaf = ElectricCar('nissan', 'leaf', 2024)
my_leaf.charge()
my_leaf.drive()
```

모듈 전체 임포트

```
import car

my_beetle = car.Car(
        'volkswagen', 'beetle', 2021)
my_beetle.fill_tank()
my_beetle.drive()

my_leaf = car.ElectricCar('nissan', 'leaf',
        2024)
my_leaf.charge()
my_leaf.drive()
```

모듈에서 모든 클래스 임포트

(되도록 이렇게 하지 말고, 무슨 뜻인지만 알아 두세요.)

```
from car import *

my_beetle = Car('volkswagen', 'beetle', 2021)
my_leaf = ElectricCar('nissan', 'leaf', 2024)
```

객체를 리스트에 저장

리스트에는 요소를 원하는 만큼 저장할 수 있으므로, 클래스에서 객체를 아주 많이 만들고 이들을 리스트에 저장할 수 있습니다.

다음은 렌터카를 대량으로 생성하고 이들이 모두 운전할 준비가 됐는지 확인하는 예제입니다.

렌터카 군단

```
from car import Car, ElectricCar

# 자동차를 저장할 리스트 생성
gas_fleet = []
electric_fleet = []

# 일반 자동차 250대, 전기차 500대
for _ in range(250):
    car = Car('ford', 'escape', 2024)
    gas_fleet.append(car)
for _ in range(500):
    ecar = ElectricCar('nissan', 'leaf', 2024)
    electric_fleet.append(ecar)

# 급유와 충전
for car in gas_fleet:
    car.fill_tank()
for ecar in electric_fleet:
    ecar.charge()

print(f"Gas cars: {len(gas_fleet)}")
print(f"Electric cars: {len(electric_fleet)}")
```

self 이해

self 변수에 대해 묻는 사람이 많습니다. self 변수는 클래스에서 생성된 객체를 가리키는 참조입니다.

self 변수는 변수와 객체를 클래스 내부 어디에서든 사용할 수 있게 하려고 설계한 변수입니다. self 변수는 객체를 통해 호출된 모든 메서드에 자동으로 전달됩니다. 이 때문에 대부분의 메서드 정의에서 self를 첫 번째 매개변수로 정의합니다. self에 연결된 변수는 클래스 어디에서든 사용할 수 있습니다.

__init__() 이해

__init__() 메서드는 다른 메서드와 마찬가지로 클래스에 속하는 함수입니다. __init__()에서 특별한 점은 클래스에서 인스턴스를 새로 만들 때마다 자동으로 호출된다는 겁니다. 실수로 __init__()이 아닌 다른 이름을 쓰면 그 메서드는 자동으로 호출되지 않고, 객체가 정확히 생성되지 않을 수 있습니다.

파일과 예외 Cheet Sheet

파일을 사용하는 이유, 예외를 사용하는 이유

프로그램은 파일에서 정보를 읽고 파일에 데이터를 기록할 수 있습니다. 파일을 읽으면 다양한 정보에 접근할 수 있고, 파일에 기록하면 사용자가 다음에 프로그램을 실행할 때 처음부터 다시 하지 않아도 됩니다. 파일에는 텍스트는 물론이고 리스트 같은 파이썬 객체도 저장할 수 있습니다.

예외는 프로그램이 에러에 적절히 대응할 수 있게 돕는 특별한 객체입니다. 예를 들어 프로그램에서 존재하지 않는 파일을 열어보려 할 때, 예외를 사용하면 프로그램의 충돌을 피하고 어떤 상황인지 에러 메시지로 알릴 수 있습니다.

파일에서 읽기

파일을 읽으려면 먼저 해당 파일의 경로를 지정하고, 그런 다음 파일 콘텐츠를 읽어야 합니다. `read_text()` 메서드는 파일 콘텐츠 전체를 문자열로 반환합니다.

파일 한번에 읽기

```
from pathlib import Path

path = Path('siddhartha.txt')
contents = path.read_text()

print(contents)
```

행 단위로 작업

행 단위로 작업하는 게 좋을 때도 있습니다. 파일 콘텐츠를 다 읽으면 `split lines()` 메서드를 써서 행으로 분리할 수 있습니다.

```
from pathlib import Path

path = Path('siddhartha.txt')
contents = path.read_text()

lines = contents.splitlines()

for line in lines:
    print(line)
```

파일에 쓰기

파일에 텍스트를 기록할 때는 `write_text()` 메서드를 사용합니다. 이 메서드는 파일을 덮어쓰므로 주의하세요. 파일에 내용을 추가할 때는 먼저 기존 콘텐츠를 붙여넣은 다음 다음 재작성해서 저장하세요.

파일에 기록

```
from pathlib import Path

path = Path("programming.txt")
msg = "I love programming!"

path.write_text(msg)
```

여러 행 기록

```
from pathlib import Path

path = Path("programming.txt")

msg = "I love programming!"
msg += "\nI love making games."
path.write_text(msg)
```

파일에 이어붙이기

```
from pathlib import Path

path = Path("programming.txt")
contents = path.read_text()

contents += "\nI love programming!"
contents += "\nI love making games."
path.write_text(contents)
```

path 객체

`pathlib` 모듈은 파일 작업을 돕습니다. `path` 객체는 파일이나 폴더에 대응하며 자주 사용하는 폴더와 파일 작업을 지원합니다.

상대 경로를 사용하면 파이썬은 현재 실행 중인 `.py` 파일을 기준으로 위치를 찾습니다. 절대 경로는 컴퓨터의 루트 폴더(/)를 기준으로 찾습니다.

윈도우는 파일 경로에 역슬래시를 사용하지만 파이썬 코드에는 슬래시를 써야 합니다.

상대 경로

```
path = Path("text_files/alice.txt")
```

절대 경로

```
path = Path("/Users/eric/text_files/alice.txt")
```

경로에서 파일 이름만 추출

```
>>> path = Path("text_files/alice.txt")
>>> path.name
'alice.txt'
```

경로 생성

```
base_path = Path("/Users/eric/text_files")
file_path = base_path / "alice.txt"
```

파일이 존재하는지 확인

```
>>> path = Path("text_files/alice.txt")
>>> path.exists()
True
```

파일 타입 확인

```
>>> path.suffix
'.txt'
```

try-except 블록

에러가 일어날 수 있다고 판단되면 `try-except` 블록으로 예외를 처리합니다. `try` 블록은 코드 실행을 시도하고, `except` 블록은 그 코드가 특정 에러를 일으켰을 때 할 일을 지정합니다.

ZeroDivisionError 예외 처리

```
try:
    print(5/0)
except ZeroDivisionError:
    print("You can't divide by zero!")
```

FileNotFoundError 예외 처리

```
from pathlib import Path

path = Path("siddhartha.txt")
try:
    contents = path.read_text()
except FileNotFoundError:
    msg = f"Can't find file: {path.name}."
    print(msg)
```

처리할 예외 판단

코드를 작성하는 시점에서 어떤 예외가 일어날지 미리 알기는 어렵습니다. 일단 `try` 블록 없이 코드를 작성하고 에러가 일어나게 두세요. 트레이스백을 보면 어떤 예외를 처리해야 할 지 알 수 있습니다. `docs.python.org/3/library/exceptions.html`에서 예외 리스트를 읽어보는 것도 좋습니다.

else 블록

`try` 블록에는 에러를 일으킬 수 있는 코드만 써야 합니다. `try` 블록이 성공적으로 실패됐을 때 실행할 코드는 `else` 블록에 씁니다.

else 블록 사용

```
print("Enter two numbers. I'll divide them.")

x = input("First number: ")
y = input("Second number: ")
```

```
try:
    result = int(x) / int(y)
except ZeroDivisionError:
    print("You can't divide by zero!")
else:
    print(result)
```

사용자 입력에 의한 충돌 방지

다음 예제에 except 블록이 없다면 프로그램은 사용자가 0으로 나누려 할 때 충돌을 일으킵니다. 하지만 예외를 처리했으므로 말끔하게 실행됩니다.

```
"""나눗셈만 하는 단순한 계산기"""

print("Enter two numbers. I'll divide them.")
print("Enter 'q' to quit.")

while True:
    x = input("\nFirst number: ")
    if x == 'q':
        break

    y = input("Second number: ")
    if y == 'q':
        break

    try:
        result = int(x) / int(y)
    except ZeroDivisionError:
        print("You can't divide by zero!")
    else:
        print(result)
```

보고할 에러 결정

코드를 주의 깊게 작성하고 제대로 테스트한다면 문법이나 논리 에러 같은 내부 에러는 잘 일어나지 않습니다. 하지만 프로그램이 사용자 입력이나 파일의 존재 같은 외부 요인에 따라 동작한다면 예외가 일어날 가능성이 있습니다.

사용자에게 어떤 에러를 전달할지는 여러분이 결정해야 합니다. 파일이 없다는 사실을 사용자가 알아야 하는 경우도 있고, 없으면 그냥 넘어가는 게 더 좋을 때도 있습니다. 경험이 좀 쌓이면 얼마나 알려줘야 할지 판단할 수 있게 될 겁니다.

조용히 실패

에러가 일어날 때도 사용자에게 보고하지 않고 계속 실행하는 게 좋을 때는 except 블록 안에 pass 문을 쓰면 됩니다.

except 블록에서 pass 문 사용

```
from pathlib import Path

f_names = ['alice.txt', 'siddhartha.txt',
        'moby_dick.txt', 'little_women.txt']

for f_name in f_names:
    # 찾아낸 파일의 길이 출력
    path = Path(f_name)
    try:
        contents = path.read_text()
    except FileNotFoundError:
        pass
    else:
        lines = contents.splitlines()
        msg = f"{f_name} has {len(lines)}"
        msg += " lines."
        print(msg)
```

빈 except 블록을 쓰지 마세요.

예외 처리 코드는 프로그램을 실행하는 동안 예상되는 특정 예외만 캐치해야 합니다. 빈 except 블록은 '모든' 예외를 캐치하는데, 이런 블록을 쓰면 시스템 중지나 키보드 인터럽트처럼 여러분이 알아야 하는 예외도 구분 없이 포함됩니다.

try 블록을 써야 하는데 어떤 예외를 캐치해야 할지 정확히 알 수 없다면 Exception을 쓰세요. 이렇게 하면 대부분의 예외를 캐치하면서도 의도적으로 프로그램 실행에 개입할 수 있습니다.

빈 except 블록을 쓰지 마세요.

```
try:
    # 코드
except:
    pass
```

대신 Exception을 쓰세요.

```
try:
    # 코드
except Exception:
    pass
```

예외 출력

```
try:
    # 코드
except Exception as e:
    print(e, type(e))
```

json으로 데이터 저장

json 모듈을 사용하면 파이썬 데이터 구조를 쉽게 파일에 저장하고 불러올 수 있습니다. JSON 데이터 형식은 파이썬 전용이 아니므로 다른 언어를 사용하는 프로그램과 데이터를 공유할 수도 있습니다.

예외 처리는 저장된 데이터를 사용할 때 더욱 중요합니다. 대개는 불러올 데이터가 존재하는지 확인하는 것으로 작업을 시작합니다.

json.dumps()로 데이터 저장

```
from pathlib import Path
import json

numbers = [2, 3, 5, 7, 11, 13]

path = Path("numbers.json")
contents = json.dumps(numbers)
path.write_text(contents)
```

json.loads()로 데이터 불러오기

```
from pathlib import Path
import json

path = Path("numbers.json")
contents = path.read_text()
numbers = json.loads(contents)

print(numbers)
```

데이터가 존재하는지 확인

```
from pathlib import Path
import json

path = Path("numbers.json")

try:
    contents = path.read_text()
except FileNotFoundError:
    msg = f"Can't find file: {path}"
    print(msg)
else:
    numbers = json.loads(contents)
    print(numbers)
```

예외 처리 연습

이미 만든 프로그램 중에서 사용자 입력을 받는 프로그램을 선택하고 에러 처리 코드를 추가해 보세요. 적절한 데이터와 부적절한 데이터를 입력하면서 각 상황을 정확히 처리하는지 확인하세요.

코드 테스트 Cheet Sheet

코드를 테스트하는 이유

함수나 클래스를 작성할 때 해당 코드에 대한 테스트도 작성할 수 있습니다. 테스트는 코드가 의도한 상황을 정확히 처리하는지 확인하고, 동시에 프로그램을 예상하지 못한 방식으로 사용할 때 어떻게 동작하는지도 확인합니다. 테스트를 만들면 많은 사람이 여러분의 프로그램을 사용하기 시작해도 코드가 정확히 동작할 거라고 확신할 수 있습니다. 또한 프로그램에 새로운 기능을 추가할 때도 테스트를 실행해 기존 기능이 제대로 동작하는지 확인할 수 있습니다.

단위 테스트는 코드를 작은 단위로 나누어서 이들이 모두 의도한 대로 동작하는지 확인하는 테스트입니다. 테스트 케이스는 단위 테스트를 모아서 프로그램이 다양한 상황에서 정확히 동작하는지 확인합니다(공간 때문에 출력 일부를 생략했습니다).

성공하는 테스트

pytest 라이브러리는 코드 테스트를 돕는 도구 모음입니다. 실명을 반환하는 함수를 만들어 이 라이브러리를 사용해 봅시다. 먼저 일반적인 프로그램 안에서 함수를 사용하고, 함수에 대한 테스트 케이스를 만듭니다.

테스트할 함수

```
full_names.py
def get_full_name(first, last):
    """실명 반환"""

    full_name = f"{first} {last}"
    return full_name.title()
```

함수 사용

names.py

```
from full_names import get_full_name

janis = get_full_name('janis', 'joplin')
print(janis)

bob = get_full_name('bob', 'dylan')
print(bob)
```

pytest 설치

pip로 pytest 설치

```
$ python -m pip install --user pytest
```

단위 테스트 하나만 있는 테스트 케이스

테스트할 함수를 임포트해서 테스트 케이스를 만듭니다. pytest는 test_로 시작하는 함수를 모두 실행합니다. 다음과 같이 test_full_names.py를 만드세요.

```
from full_names import get_full_name

def test_first_last():
    """Janis Joplin 같은 이름 테스트"""
    full_name = get_full_name('janis',
            'joplin')
    assert full_name == 'Janis Joplin'
```

테스트 실행

pytest 명령을 실행하면 pytest가 test_로 시작하는 파일을 모두 실행합니다. pytest는 테스트 케이스에 포함된 테스트를 모두 보고합니다.
test_full_names.py 다음에 있는 점 하나는 테스트 하나가 통과했다는 뜻입니다. 다음은 0.01초 동안 테스트 하나를 실행했고 성공했다는 뜻입니다.

```
$ pytest
============= test session starts =============
platform darwin -- Python 3.11.0, pytest-7.1.2
rootdir: /.../testing_your_code
collected 1 item
test_full_names.py .                      [100%]
============== 1 passed in 0.01s ==============
```

실패하는 테스트

테스트에 실패하는 것도 중요합니다. 실패하는 테스트는 코드를 변경하면서 기존 동작에 영향이 있었다는 의미입니다. 테스트가 실패하면 코드를 수정해서 기존 동작도 문제가 없게 해야 합니다.

함수 수정

get_full_name()이 중간 이름도 처리하도록 수정합니다. 하지만 여기서는 의도적으로 기존 동작을 망가뜨릴 겁니다.

```
def get_full_name(first, middle, last):
    """실명 반환"""
    full_name = f"{first} {middle} {last}"
    return full_name.title()
```

함수 사용

```
from full_names import get_full_name

john = get_full_name('john', 'lee', 'hooker')
print(john)

david = get_full_name('david', 'lee', 'roth')
print(david)
```

테스트 실행

코드를 변경하면 기존 테스트를 실행해야 합니다. 이렇게 해야만 기존 동작에 영향이 없는지 확인할 수 있습니다.

```
$ pytest
============= test session starts =============
test_full_names_failing.py F              [100%]
================= FAILURES ====================
_____________ test_first_last _____________
>       full_name = get_full_name('janis',
                'joplin')
E       TypeError: get_full_name() missing 1
        required positional argument: 'last'
=========== short test summary info ===========
FAILED test_full_names.py::test_first_last...
============== 1 failed in 0.04s ==============
```

코드 수정

테스트가 실패하면 다시 테스트를 통과하도록 코드를 수정해야 합니다. 새로운 코드가 통과하게끔 테스트를 수정하면 안 됩니다. 실패가 다른 방식으로 바뀔 뿐입니다.
다음 코드는 중간 이름을 옵션으로 처리합니다.

```
def get_full_name(first, last, middle=''):
    """실명 반환"""

    if middle:
        full_name = f"{first} {middle} {last}"
    else:
        full_name = f"{first} {last}"

    return full_name.title()
```

테스트 실행

이제 테스트를 다시 통과합니다. 원래 기능이 잘 동작한다는 뜻입니다.

```
$ pytest
============= test session starts =============
test_full_names.py .                      [100%]
============== 1 passed in 0.01s ==============
```

테스트 추가

테스트 케이스에 단위 테스트를 필요한 만큼 추가할 수 있습니다. 새로운 테스트를 추가하려면 테스트 파일에 함수를 추가하세요. 파일이 너무 길어지면 필요한 만큼 여러 개의 파일로 분리하면 됩니다.

중간 이름 테스트

get_full_name()이 이름과 성을 받았을 때 잘 동작하는 걸 확인했습니다. 이번에는 중간 이름도 잘 처리하는지 확인합시다.

```
from full_names import get_full_name

def test_first_last():
    """Janis Joplin 같은 이름 테스트"""
    full_name = get_full_name('janis',
            'joplin')
    assert full_name == 'Janis Joplin'

def test_middle():
    """David Lee Roth 같은 이름 테스트"""
    full_name = get_full_name('david',
            'roth', 'lee')
    assert full_name == 'David Lee Roth'
```

테스트 실행

test_full_names.py 다음에 있는 점 두 개는 테스트 두 개가 통과했다는 뜻입니다.

```
$ pytest
============= test session starts =============
collected 2 items
test_full_names.py ..                    [100%]
============== 2 passed in 0.01s ==============
```

다양한 assert 문

assert 문은 조건을 정확히 만족하는지 다양한 방법으로 확인합니다.

a == b, a != b 확인

```
assert a == b
assert a != b
```

x가 True인지 False인지 확인

```
assert x
assert not x
```

아이템가 리스트에 존재하는지 확인

```
assert my_item in my_list
assert my_item not in my_list
```

파일 하나에서 테스트 실행

테스트 스위트가 커지면 테스트 파일이 여러 개로 나뉠 수 있습니다. 때때로 파일 하나만 테스트하고 싶을 때도 있습니다. 이럴 때는 테스트할 파일 이름을 전달하면 됩니다.

```
$ pytest test_names_function.py
```

클래스 테스트

클래스 테스트는 주로 메서드를 테스트하므로 함수 테스트와 비슷합니다.

테스트할 클래스

account.py

```
class Account():
    """은행 계좌 관리"""

    def __init__(self, balance=0):
        """초기 잔액 설정"""
        self.balance = balance

    def deposit(self, amount):
        """입금"""
        self.balance += amount

    def withdraw(self, amount):
        """출금"""
        self.balance -= amount
```

테스트 케이스 생성

첫 번째 테스트는 두 가지 초기 잔액으로 시작할 수 있는지부터 확인합니다. 다음과 같이 test_accountant.py를 만드세요.

```
from account import Account

def test_initial_balance():
    """기본 잔액은 0"""
    account = Account()
    assert account.balance == 0

def test_deposit():
    """입금 테스트"""
    account = Account()
    account.deposit(100)
    assert account.balance == 100
```

테스트 실행

```
$ pytest
============= test session starts =============
collected 2 items
test_account.py ..                       [100%]
============== 2 passed in 0.01s ==============
```

테스트 자체를 수정해도 되는 경우

일반적으로 일단 테스트를 작성하면 수정하지 않는 게 좋습니다. 테스트가 실패한다면 새로운 코드가 기존 기능을 망쳤다는 뜻이며, 기존 테스트를 통과할 때까지 새 코드를 수정해야 합니다.

하지만 요건 자체가 바뀌었다면 테스트를 수정해야 할 수도 있습니다. 이런 경우는 대개 프로젝트 초반에 기능의 방향을 확정하지 못했고, 아직 사용자가 없는 경우에 한합니다.

픽스처

픽스처는 여러 테스트에서 공유하는 기능입니다. 테스트 함수에서 픽스처 함수의 이름을 인수로 정의하면 픽스처의 반환 값이 테스트 함수로 전달됩니다.

클래스를 테스트하다 보면 인스턴스를 만들어야 할 때가 많습니다. 픽스처를 사용하면 인스턴스를 하나만 만들어도 됩니다.

여러 테스트에 픽스처 사용

acc 인스턴스는 여러 가지 테스트에서 사용할 수 있습니다.

```
import pytest
from account import Account

@pytest.fixture
def account():
    account = Account()
    return account

def test_initial_balance(account):
    """기본 잔액은 0"""
    assert account.balance == 0

def test_deposit(account):
    """입금 테스트"""
    account.deposit(100)
    assert account.balance == 100

def test_withdrawal(account):
    """입금 후 출금 확인"""
    account.deposit(1_000)
    account.withdraw(100)
    assert account.balance == 900
```

테스트 실행

```
$ pytest
============= test session starts =============
collected 3 items
test_account.py ...                      [100%]
============== 3 passed in 0.01s ==============
```

pytest 플래그

pytest에는 테스트를 효율적으로 실행할 수 있도록 돕는 몇 가지 플래그가 있습니다. 프로젝트에 포함되는 테스트가 많아져도 플래그는 여전히 유용합니다.

테스트가 하나라도 실패하면 중지

```
$ pytest -x
```

이전에 실패했던 테스트만 다시 실행

```
$ pytest --last-failed
```

파이게임 Cheet Sheet

파이게임이란?

파이게임은 파이썬으로 게임을 만들 때 사용하는 프레임워크입니다. 게임을 만드는 건 재미있는 일이며 프로그래밍 기술과 지식을 늘리는 훌륭한 방법입니다. 파이게임은 게임을 만들 때 필요한 저수준 작업을 대부분 처리하므로 프로그래머는 게임을 흥미롭게 만드는 데만 집중하면 됩니다.

파이게임 설치

pip를 이용해서 파이게임 설치

```
$ python -m pip install --user pygame
```

게임 시작

다음 코드는 비어 있는 게임 윈도우를 생성하고, 이벤트 루프를 시작하고, 계속해서 화면을 새로고침하는 루프를 만듭니다.

빈 게임 화면

```
import sys
import pygame

class AlienInvasion:
    """게임을 전체적으로 관리하는 클래스"""

    def __init__(self):
        pygame.init()
        self.clock = pygame.time.Clock()
        self.screen = pygame.display.set_mode(
                (1200, 800))
        pygame.display.set_caption(
                "Alien Invasion")

    def run_game(self):
        while True:
            for event in pygame.event.get():
                if event.type == pygame.QUIT:
                    sys.exit()

            pygame.display.flip()
            self.clock.tick(60)

if __name__ == '__main__':
    # 게임 인스턴스를 생성하고 게임을 실행
    ai = AlienInvasion()
    ai.run_game()
```

윈도우 크기 설정

display.set_mode() 함수는 화면 크기를 정의하는 튜플을 받습니다

```
screen_dim = (1500, 1000)
self.screen = pygame.display.set_mode(
        screen_dim)
```

배경색 설정

색깔을 정의할 때는 빨간색, 녹색, 파란색을 값으로 가진 튜플을 사용합니다. 각 값은 0에서 255 범위입니다. fill() 메서드는 지정한 색깔로 화면을 채웁니다. 다른 요소를 그리기 전에 가장 먼저 이 메서드를 호출해야 합니다.

```
def __init__(self):
    --생략--
    self.bg_color = (225, 225, 225)

def run_game(self):
    while True:
        for event in pygame.event.get():
            --생략--

        self.screen.fill(self.bg_color)
        pygame.display.flip()
```

rect 객체

게임에 존재하는 대부분의 객체를 실제 모양이 아닌 단순한 사각형으로 취급해도 무방합니다. 이렇게 하면 실제 게임 플레이에 거의 영향 없이 코드를 단순화할 수 있습니다. 파이게임에는 게임 객체를 다루기 쉽게 하는 rect 객체가 있습니다.

화면의 rect 객체

화면 객체는 이미 존재하며 연관된 rect 객체에 쉽게 접근할 수 있습니다.

```
self.screen_rect = self.screen.get_rect()
```

화면 중앙 찾기

rect 객체에는 중앙을 가리키는 center 속성이 있습니다.

```
screen_center = self.screen_rect.center
```

rect 객체의 유용한 속성

rect 객체에는 객체의 위치를 알아내고 상대적인 위치 관계를 파악할 때 유용한 속성이 많이 있습니다(파이게임 문서에 더 많은 속성이 있습니다. 명쾌함을 위해 self 변수는 생략했습니다).

```
# 개별적으로 존재하는 x와 y 값
screen_rect.left,    screen_rect.right
screen_rect.top,     screen_rect.bottom
screen_rect.centerx, screen_rect.centery
screen_rect.width,   screen_rect.height

# 튜플
screen_rect.center
screen_rect.size
```

rect 객체 생성

rect 객체를 처음부터 만들 수도 있습니다. 예를 들어 작은 rect 객체를 만들고 색깔을 입혀 게임 안에서 탄환으로 쓸 수 있습니다. Rect() 클래스는 rect의 왼쪽 상단 꼭짓점 좌표, 너비, 높이를 받습니다. draw.rect() 함수는 화면 객체, 색깔, rect를 받습니다. 이 함수는 주어진 rect를 주어진 색깔로 칠합니다.

```
bullet_rect = pygame.Rect(100, 100, 3, 15)
color = (100, 100, 100)

pygame.draw.rect(screen, color, bullet_rect)
```

이미지 사용

게임의 객체 중 상당수는 이리저리 움직이는 이미지입니다. 비트맵(.bmp) 이미지 파일을 사용하는 게 가장 쉽지만 jpg, png, gif 파일도 사용할 수 있습니다.

이미지 로딩

```
ship = pygame.image.load('images/ship.bmp')
```

이미지에서 rect 객체 생성

```
ship_rect = ship.get_rect()
```

이미지 위치 지정

rect를 사용하면 화면 위 어디든 이미지를 배치할 수 있습니다. 다음 코드는 우주선의 midbottom과 화면의 midbottom을 일치시키는 방법으로 우주선을 화면의 하단 중앙에 배치합니다.

```
ship_rect.midbottom = screen_rect.midbottom
```

화면에 이미지 그리기

이미지를 불러와서 배치한 다음에는 blit() 메서드로 화면에 그립니다. blit() 메서드는 화면 객체의 메서드이며 인수로 이미지 객체와 rect를 받습니다.

```
# 화면에 우주선 표시
screen.blit(ship, ship_rect)
```

이미지 변환

transform 모듈은 이미지를 회전하거나 크기를 바꾸는 등의 작업을 할 수 있습니다.

```
rotated_image = pygame.transform.rotate(
        ship.image, 45)
```

blitme() 메서드

우주선 같은 게임 객체는 대개 클래스로 작성합니다. 그리고 보통 해당 객체를 화면에 그리는 blitme() 메서드도 정의합니다.

```
def blitme(self):
    """현재 위치에 우주선 표시"""
    self.screen.blit(self.image, self.rect)
```

키보드 입력에 반응

파이게임은 키 입력이나 마우스 동작 같은 이벤트를 주시합니다. 이벤트 루프에서 원하는 이벤트를 감지하고 게임에 알맞게 대처할 수 있습니다.

키 입력에 반응

파이게임의 메인 이벤트 루프는 사용자가 키를 누를 때마다 KEYDOWN 이벤트를 등록합니다. 이 이벤트에서 원하는 키인지 확인할 수 있습니다.

```
for event in pygame.event.get():
    if event.type == pygame.KEYDOWN:
        if event.key == pygame.K_RIGHT:
            ship_rect.x += 1
        elif event.key == pygame.K_LEFT:
            ship_rect.x -= 1
        elif event.key == pygame.K_SPACE:
            ship.fire_bullet()
        elif event.key == pygame.K_q:
            sys.exit()
```

키에서 손을 뗐을 때 반응

사용자가 키에서 손을 떼면 KEYUP 이벤트가 일어납니다.

```
for event in pygame.event.get():
    if event.type == pygame.KEYUP:
        if event.key == pygame.K_RIGHT:
            ship.moving_right = False
```

게임 자체도 객체입니다.

현재 전체적인 구조에서는 게임 자체가 클래스입니다. 그러므로 게임을 자동으로 실행할 수 있고, 여러 가지 게임을 모은 컬렉션 형태로도 만들 수 있습니다.

파이게임 문서

여러분이 직접 게임을 만들 때는 파이게임 문서가 정말 유용할 겁니다. 파이게임 프로젝트의 홈페이지는 pygame.org/, 문서 홈페이지는 pygame.org/docs/입니다.

문서에서 가장 유용한 부분은 Rect() 클래스나 sprite 모듈 같은 부분입니다. 도움말 페이지 상단에서 관심이 가는 부분을 찾아 보세요.

마우스 이벤트에 반응

파이게임의 이벤트 루프는 마우스가 움직이거나 버튼을 누르고 뗄 때도 이벤트를 등록합니다.

마우스 버튼에 반응

```
for event in pygame.event.get():
    if event.type == pygame.MOUSEBUTTONDOWN:
        ship.fire_bullet()
```

마우스 위치 찾기

마우스 위치는 튜플로 반환됩니다.

```
mouse_pos = pygame.mouse.get_pos()
```

버튼 클릭

버튼 같은 객체를 클릭하면 그에 반응해야 합니다. rect.collidepoint() 메서드는 마우스 포인트가 rect 객체와 겹칠 때 True를 반환합니다.

```
if button_rect.collidepoint(mouse_pos):
    start_game()
```

마우스 숨기기

```
pygame.mouse.set_visible(False)
```

파이게임 그룹

파이게임에는 비슷한 객체들을 그룹으로 묶어 더 쉽게 다룰 수 있는 Group 클래스가 있습니다. 그룹은 리스트와 비슷하지만 게임을 만들 때 유용한 몇 가지 기능이 추가되어 있습니다.

그룹 생성

그룹에 저장될 객체는 반드시 Sprite를 상속해야 합니다.

```
from pygame.sprite import Sprite, Group
class Bullet(Sprite):
    ...
    def draw_bullet(self):
        ...
    def update(self):
        ...

bullets = Group()

new_bullet = Bullet()
bullets.add(new_bullet)
```

그룹 순회하기

sprites() 메서드는 그룹 요소를 모두 반환합니다.

```
for bullet in bullets.sprites():
    bullet.draw_bullet()
```

그룹에서 update() 호출

그룹에서 update()를 호출하면 자동으로 그룹의 모든 요소에서 update()를 호출합니다.

```
bullets.update()
```

그룹에서 요소 제거

다시 등장하지 않을 요소가 있다면 삭제해서 메모리와 자원 낭비를 막아야 합니다.

```
bullets.remove(bullet)
```

충돌 감지

객체 하나가 그룹 요소와 충돌하는 걸 감지할 수 있습니다. 또한 그룹의 요소가 다른 그룹의 요소와 충돌하는 것도 감지할 수 있습니다.

객체와 그룹의 충돌

spritecollideany() 함수는 객체와 그룹을 인수로 받고, 객체가 그룹 요소 중 하나와 겹치면 True를 반환합니다.

```
if pygame.sprite.spritecollideany(ship, aliens):
    ships_left -= 1
```

그룹 사이의 충돌

sprite.groupcollide() 함수는 인수로 그룹 두 개와 불리언 두 개를 받습니다. 이 함수는 충돌한 요소들의 정보를 포함하는 딕셔너리를 반환합니다. 불리언은 두 그룹의 충돌한 요소들을 삭제할지 결정합니다.

```
collisions = pygame.sprite.groupcollide(
        bullets, aliens, True, True)

score += len(collisions) * alien_point_value
```

텍스트 렌더링

게임에서 다양한 목적으로 텍스트를 사용할 수 있습니다. 예를 들어 플레이어에게 정보나 점수 등을 표시할 수 있습니다.

메시지 표시

다음 코드는 메시지, 텍스트 색깔, 배경색을 정의합니다. 폰트는 컴퓨터의 기본 폰트를 사용하며 크기는 48로 정합니다. font.render() 함수는 메시지를 이미지로 변환하고, 이와 연관된 rect 객체를 가져옵니다. 그런 다음 이미지를 화면 중앙에 표시합니다.

```
msg = "Play again?"
msg_color = (100, 100, 100)
bg_color = (230, 230, 230)

f = pygame.font.SysFont(None, 48)
msg_image = f.render(msg, True, msg_color,
        bg_color)
msg_image_rect = msg_image.get_rect()
msg_image_rect.center = screen_rect.center
screen.blit(msg_image, msg_image_rect)
```

Matplotlib Cheet Sheet

Matplotlib이란?

데이터 시각화는 데이터를 전체적으로 파악하기 쉽게 만드는 작업입니다. Matplotlib 라이브러리는 작업할 데이터를 매력적으로 표현할 수 있게 합니다. Matplotlib은 아주 유연합니다. 다음 예제들은 여러 작업 중 극히 일부에 불과합니다.

Matplotlib 이후에도 여러 가지 라이브러리가 발표됐지만 상당수는 Matplotlib을 보조하는 형태이므로, Matplotlib을 잘 이해하면 여러 라이브러리를 효율적으로 사용할 수 있습니다.

Matplotlib 설치

pip를 이용해서 Matplotlib 설치

```
$ python -m pip install --user matplotlib
```

직선 그래프와 산포도

직선 그래프

fig 객체는 전체 그림, 달리 말해 그래프 컬렉션이며 ax는 그림에 표시된 그래프 하나입니다. 그림에 그래프가 단 하나만 존재하더라도 이렇게 표기합니다.

```
import matplotlib.pyplot as plt

x_values = [0, 1, 2, 3, 4, 5]
squares = [0, 1, 4, 9, 16, 25]

fig, ax = plt.subplots()
ax.plot(x_values, squares)

plt.show()
```

산포도

scatter()는 x와 y 리스트를 받습니다. s=10 인수는 각 점의 크기를 지정합니다.

```
import matplotlib.pyplot as plt

x_values = list(range(1000))
squares = [x**2 for x in x_values]

fig, ax = plt.subplots()
ax.scatter(x_values, squares, s=10)

plt.show()
```

그래프 사용자 정의

그래프는 다양한 방식으로 설정할 수 있습니다. 거의 모든 요소를 세부 설정할 수 있습니다.

내장된 스타일 사용

Matplotlib을 설치하면 다양한 스타일이 함께 설치됩니다. 단 한 줄의 코드로 이들을 이용할 수 있습니다. 그림을 만들기 전에 반드시 스타일부터 지정해야 합니다.

```
import matplotlib.pyplot as plt

x_values = list(range(1000))
squares = [x**2 for x in x_values]

plt.style.use('seaborn-v0_8')
fig, ax = plt.subplots()
ax.scatter(x_values, squares, s=10)

plt.show()
```

사용할 수 있는 스타일 확인

사용할 수 있는 모든 스타일은 터미널 세션에서 확인 가능합니다.

```
>>> import matplotlib.pyplot as plt
>>> plt.style.available
['Solarize_Light2', '_classic_test_patch', ...
```

제목 지정, 이름표 추가, 축 크기 지정

```
import matplotlib.pyplot as plt

x_values = list(range(1000))
squares = [x**2 for x in x_values]

# 전체적인 스타일과 그래프 데이터
plt.style.use('seaborn-v0_8')
fig, ax = plt.subplots()
ax.scatter(x_values, squares, s=10)

# 제목과 축 이름표 지정
ax.set_title('Square Numbers', fontsize=24)
ax.set_xlabel('Value', fontsize=14)
ax.set_ylabel('Square of Value', fontsize=14)

# 축 배율, 눈금 이름표 크기 지정
ax.axis([0, 1100, 0, 1_100_000])
ax.tick_params(axis='both', labelsize=14)

plt.show()
```

컬러맵

컬러맵은 데이터 포인트의 특정 값을 바탕으로 색깔을 점진적으로 바꾸는 그레이디언트입니다. 데이터 포인트의 색깔은 c 인수로 지정하고, 사용할 컬러맵은 cmap 인수로 지정합니다.

```
ax.scatter(x_values, squares, c=squares,
        cmap=plt.cm.Blues, s=10)
```

포인트 강조

그래프에는 수많은 데이터가 포함됩니다. 다음 코드는 첫 번째와 마지막 포인트를 더 크게 만들어 강조합니다.

```
import matplotlib.pyplot as plt

x_values = list(range(1000))
squares = [x**2 for x in x_values]

fig, ax = plt.subplots()
ax.scatter(x_values, squares, c=squares,
        cmap=plt.cm.Blues, s=10)

ax.scatter(x_values[0], squares[0], c='green',
        s=100)
ax.scatter(x_values[-1], squares[-1], c='red',
        s=100)

ax.set_title('Square Numbers', fontsize=24)
--생략--
```

축 제거

그래프의 축은 자유롭게 설정할 수 있으며 완전히 제거하는 것도 가능합니다. 다음은 각 축에 접근해 숨기는 방법입니다.

```
ax.get_xaxis().set_visible(False)
ax.get_yaxis().set_visible(False)
```

그림 크기 설정

figsize 인수를 통해 크기를 지정할 수 있습니다. dpi 인수는 옵션입니다. 컴퓨터의 해상도를 모른다면 이 인수는 생략하고 figsize 인수를 조절해도 됩니다.

```
fig, ax = plt.subplots(figsize=(10, 6),
        dpi=128)
```

그래프 저장

Matplotlib 뷰어에도 저장 버튼이 있지만, plt.show() 대신 plt.savefig()를 사용하면 시각화 결과를 저장할 수 있습니다. bbox_inches 인수는 그림 주위의 공백을 줄일 때 사용합니다.

```
plt.savefig('squares.png', bbox_inches='tight')
```

여러 개의 그래프

그림 하나에 그래프를 원하는 만큼 넣을 수 있습니다. 그래프 여러 개를 사용하면 데이터 사이의 관계가 드러납니다. 예를 들어 두 데이터 집합 사이의 공간에 색을 칠할 수 있습니다.

두 개의 데이터 그래프

다음 예제는 ax.scatter()를 두 번 호출해서 같은 그림에 제곱수와 세제곱 그래프를 그립니다.

```
import matplotlib.pyplot as plt

x_values = list(range(11))
squares = [x**2 for x in x_values]
cubes = [x**3 for x in x_values]

plt.style.use('seaborn-v0_8')
fig, ax = plt.subplots()

ax.scatter(x_values, squares, c='blue', s=10)
ax.scatter(x_values, cubes, c='red', s=10)

plt.show()
```

데이터 집합 사이의 공백 칠하기

fill_between() 메서드는 두 데이터 집합 사이의 공간에 색을 칠합니다. 이 함수는 x 값 리스트 하나, y 값 리스트 두 개를 받습니다. 또한 facecolor를 통해 색깔을 지정하고, 옵션인 alpha 인수는 색깔의 투명도를 지정합니다.

```
ax.fill_between(x_values, cubes, squares,
        facecolor='blue', alpha=0.25)
```

날짜와 시간

날짜와 시간이 관련된 데이터 집합은 대부분 날짜와 시간을 x 값으로 사용합니다. 파이썬의 datetime 모듈을 사용해 이런 데이터를 쉽게 다룰 수 있습니다.

현재 날짜 확인

datetime.now() 함수는 현재 날짜와 시간을 나타내는 datetime 객체를 반환합니다.

```
from datetime import datetime as dt

today = dt.now()
date_string = today.strftime('%m/%d/%Y')
print(date_string)
```

특정 날짜

원하는 날짜와 시간을 나타내는 datetime 객체를 만들 수도 있습니다. 인수는 년, 월, 일 순서로 씁니다. 시, 분, 초, 밀리초 인수는 옵션입니다.

```
from datetime import datetime as dt

new_years = dt(2023, 1, 1)
fall_equinox = dt(year=2023, month=9, day=22)
```

날짜와 시간 형식

strptime() 함수는 문자열에서 datetime 객체를 생성하고, strftime() 메서드는 반대로 datetime 객체에서 형식화된 문자열을 만듭니다. 다음 코드를 숙지하면 날짜를 원하는 형식으로 표시할 수 있습니다.

```
%A    요일 이름 (Monday 등)
%B    월 이름 (January 등)
%m    숫자로 나타낸 월 (01 ~ 12)
%d    숫자로 나타낸 일 (01 ~ 31)
%Y    네 자리 숫자인 년 (2021)
%y    두 자리 숫자인 년 (21)
%H    24시간 형식의 시간 (00 ~ 23)
%I    12시간 형식의 시간 (01 ~ 12)
%p    AM, PM
%M    분 (00 ~ 59)
%S    초 (00 ~ 61)
```

문자열을 datetime 객체로 변환

```
new_years = dt.strptime('1/1/2023', '%m/%d/%Y')
```

datetime 객체를 문자열로 변환

```
ny_string = new_years.strftime('%B %d, %Y')
print(ny_string)
```

최고 기온 그래프

다음 코드는 날짜 리스트를 만들고 그에 대응하는 최고 기온 리스트를 만듭니다. 그런 다음 최고 기온을 그래프로 그리고, 날짜 이름표는 지정된 형식으로 표시합니다.

```
from datetime import datetime as dt

import matplotlib.pyplot as plt
from matplotlib import dates as mdates

dates = [
    dt(2023, 6, 21), dt(2023, 6, 22),
    dt(2023, 6, 23), dt(2023, 6, 24),
    ]

highs = [56, 57, 57, 64]

plt.style.use('seaborn-v0_8')
fig, ax = plt.subplots()
ax.plot(dates, highs, c='red')

ax.set_title("Daily High Temps", fontsize=24)
ax.set_ylabel("Temp (F)", fontsize=16)
x_axis = ax.get_xaxis()
x_axis.set_major_formatter(
        mdates.DateFormatter('%B %d %Y')
        )
fig.autofmt_xdate()

plt.show()
```

그림 하나에 여러 그래프

그림 하나에 그래프를 원하는 만큼 넣을 수도 있습니다.

x 축 공유

다음 코드는 제곱수와 세제곱수를 그래프로 그립니다. 두 그래프는 x 축을 공유합니다. plt.subplots() 함수는 figure 객체와 축 튜플을 반환합니다. 각 축 집합은 서로 다른 그래프와 연결됩니다. 첫 번째와 두 번째 인수는 그림에 생성될 행과 열 개수를 지정합니다.

```
import matplotlib.pyplot as plt

x_values = list(range(11))
squares = [x**2 for x in x_values]
cubes = [x**3 for x in x_values]

fig, axs = plt.subplots(2, 1, sharex=True)

axs[0].scatter(x_values, squares)
axs[0].set_title('Squares')

axs[1].scatter(x_values, cubes, c='red')
axs[1].set_title('Cubes')

plt.show()
```

y 축 공유

y 축을 공유하려면 sharey=True 인수를 사용합니다.

```
import matplotlib.pyplot as plt

x_values = list(range(11))
squares = [x**2 for x in x_values]
cubes = [x**3 for x in x_values]

plt.style.use('seaborn-v0_8')
fig, axs = plt.subplots(1, 2, sharey=True)

axs[0].scatter(x_values, squares)
axs[0].set_title('Squares')

axs[1].scatter(x_values, cubes, c='red')
axs[1].set_title('Cubes')

plt.show()
```

온라인 자료

matplotlib.org/에 Matplotlib 갤러리와 문서가 있습니다. 예제, 교재[tutorial], 사용자 가이드는 꼭 방문해 보세요.

Plotly Cheet Sheet

Plotly란?

데이터 시각화는 데이터를 전체적으로 파악하기 쉽게 만드는 작업입니다. Plotly는 데이터를 더 시각적으로 매력 있게 표현합니다. Plotly는 요소의 상호작용을 지원하므로 시각화을 온라인으로 사용할 때도 알맞습니다.

Plotly Express는 단 몇 줄의 코드로 기본적인 그래프를 그릴 수 있습니다. 그래프가 데이터와 잘 어울린다고 판단하면 스타일을 조정하면 됩니다.

Plotly 설치

Plotly Express를 사용하려면 판다스 라이브러리가 필요합니다.

pip를 이용해서 Plotly 설치

```
$ python -m pip install --user plotly
$ python -m pip install --user pandas
```

직선 그래프, 산포도, 막대 그래프

탭이 새로 열리고 이 탭에 그래프가 그려집니다. 기본적인 그래프는 단 두 행의 코드로도 만들 수 있습니다.

직선 그래프

Plotly는 그래프 파일을 실제로 그릴 자바스크립트 코드를 생성하는 방식으로 동작합니다. 자바스크립트 코드가 궁금하면 브라우저의 개발자 도구를 보세요.

```
import plotly.express as px

# 데이터 정의
x_values = list(range(11))
squares = [x**2 for x in x_values]

# 데이터 시각화
fig = px.line(x=x_values, y=squares)
fig.show()
```

산포도

산포도를 그릴 때는 `line()`을 `scatter()`로 바꾸면 됩니다. 이런 방식이 Plotly Express의 핵심적인 장점입니다. 스타일을 세부적으로 지정하기 전에, 어떤 그래프가 데이터와 가장 잘 어울리는지 빠르게 훑어볼 수 있습니다.

```
fig = px.scatter(x=x_values, y=squares)
```

막대 그래프

```
fig = px.bar(x=x_values, y=squares)
```

기본적인 사용자 정의

그래프를 그리는 함수는 제목, 이름표, 기타 형식 지시자도 매개변수로 받습니다.

제목과 축 이름표 추가

제목은 문자열입니다. `labels` 딕셔너리는 그래프에 사용할 이름표 형식을 지정합니다.

```
import plotly.express as px

# 데이터 정의
x_values = list(range(11))
squares = [x**2 for x in x_values]

# 데이터 시각화
title = "Square Numbers"
labels = {'x': 'Value', 'y': 'Square of Value'}

fig = px.scatter(x=x_values, y=squares,
        title=title, labels=labels)
fig.show()
```

더 다양한 사용자 정의

Plotly Express는 가능한 적은 코드로 상세하게 설정할 수 있도록 설계됐습니다. 다음 코드는 한 번의 호출로 얼마나 상세히 설정할 수 있는지 묘사하는 예시입니다. 대부분의 인수는 값 하나, 또는 전체 데이터 집합에 대응하는 리스트입니다.

```
import plotly.express as px

x_values = list(range(11))
squares = [x**2 for x in x_values]

title = "Square Numbers"
labels = {'x': 'Value', 'y': 'Square of Value'}

fig = px.scatter(
    x=x_values,
    y=squares,
    title=title,
    labels=labels,
    size=squares,
    color=squares,
    opacity=0.5,
    width=1200,
    height=800,
)

fig.show()
```

더 많은 사용자 정의

`update` 메서드를 써서 그래프를 더 상세히 설정할 수 있습니다. 예를 들어 `update_layout()`에는 여러 가지 형식 옵션이 있습니다.

`update_layout()` 메서드 사용

다음 코드는 `update_layout()` 메서드를 써서 폰트 크기와 x 축의 눈금 이름표 사이의 간격을 지정했습니다.

```
import plotly.express as px

x_values = list(range(11))
squares = [x**2 for x in x_values]

title = "Square Numbers"
labels = {'x': 'Value', 'y': 'Square of Value'}

fig = px.scatter(
    x=x_values,
    y=squares,
    ...
)

fig.update_layout(
    title_font_size=30,
    xaxis_title_font_size=24,
    xaxis_dtick=1,
    xaxis_tickfont_size=16,
    yaxis_title_font_size=24,
    yaxis_tickfont_size=16,
)

fig.show()
```

Plotly Express 문서

Plotly 문서는 상세하고 잘 정리되어 있습니다. 하지만 너무 방대하다 보니 어디에 뭐가 있는지 찾기 어려울 수도 있습니다. `plotly.com/python/plotly-express`에서 먼저 개요를 보세요. 이 페이지 자체도 꽤 유용합니다. 그리고 가장 자주 사용하는 그래프 문서를 읽어보세요.

또한 `plotly.com/python-api-reference`에서 파이썬 API 참조 문서도 읽어보세요. 이 페이지에서 Plotly로 만들 수 있는 다양한 그래프 종류를 확인할 수 있습니다. 링크를 클릭하면 함수에 쓸 수 있는 인수가 모두 나열됩니다.

미리 정의된 테마 사용

테마란 Plotly에서 시각화에 적용할 수 있는 스타일 모음입니다. 테마는 템플릿을 통해 만들어집니다.

테마 사용

```
import plotly.express as px

# 데이터 정의
x_values = list(range(11))
```

```
squares = [x**2 for x in x_values]

# 데이터 시각화
title = "Square Numbers"
labels = {'x': 'Value', 'y': 'Square of Value'}

fig = px.scatter(x=x_values, y=squares,
        title=title, labels=labels,
        template='plotly_dark')
fig.show()
```

사용할 수 있는 테마 모두 확인

```
>>> import plotly.io as pio
>>> pio.templates
Templates configuration
-----------------------
    Default template: 'plotly'
    Available templates:
        ['ggplot2', 'seaborn',...,
         'ygridoff', 'gridon', 'none']
```

Plotly Express 그래프에 트레이스 추가

Plotly에서 트레이스란 그래프로 만들 수 있는 데이터 집합입니다. 기존의 Plotly Express 그래프에 트레이스를 추가할 수 있습니다. 그래프를 추가하려면 graph_objects 모듈을 사용해야 합니다.

fig.add_trace() 사용

```
import plotly.express as px
import plotly.graph_objects as go

days = list(range(1, 10))
highs = [60, 63, 68, 70, 68, 70, 66, 62, 64]
lows = [51, 54, 53, 57, 54, 56, 52, 53, 49]

# 먼저 최저 기온 그래프로 시작
fig = px.line(x=days, y=lows)

# 최고 기온 트레이스 추가
new_trace = go.Scatter(x=days, y=highs,
        mode='lines')
fig.add_trace(new_trace)

fig.show()
```

서브플롯

여러 개의 그래프가 축 하나를 공유하게 만들어야 할 때가 종종 있습니다. 이럴 때 subplots 모듈을 사용합니다.

그림에 서브플롯 추가

subplots 모듈을 사용하려면 우선 모든 그래프를 포함할 그림을 만듭니다. 그런 다음 add_trace() 메서드를 써서 전체 그림에 데이터를 추가합니다.
그래프는 모두 graph_objects 모듈로 만들어야 합니다.

```
from plotly.subplots import make_subplots
import plotly.graph_objects as go

x_values = list(range(11))
squares = [x**2 for x in x_values]
cubes = [x**3 for x in x_values]

# y 축을 공유하는 서브플롯 두 개
fig = make_subplots(rows=1, cols=2,
        shared_yaxes=True)

# 제곱수 그래프로 시작
squares_trace = go.Scatter(x=x_values,
        y=squares)
fig.add_trace(squares_trace, row=1, col=1)

# 세제곱 트레이스 추가
cubes_trace = go.Scatter(x=x_values, y=cubes)
fig.add_trace(cubes_trace, row=1, col=2)

title = "Squares and Cubes"
fig.update_layout(title_text=title)

fig.show()
```

그 외의 문서

Plotly Express 문서를 읽은 후에는 plotly.com/python/styling-plotly-express에서 Plotly Express 파이썬 스타일 가이드를 읽으세요. 이 문서에는 그래프에 스타일과 형식을 지정하는 방법이 모두 있습니다. 그 다음에는 파이썬 그림 참조(plotly.com/python/reference/index)를 읽어보세요. 이 문서에는 변경할 수 있는 설정 전체와 예제가 들어 있습니다.

plotly.com/python/creating-and-updating-figures의 '매직 언더스코어'도 놓치지 마세요. 이 문서에서 소개하는 문법에 익숙해지려면 시간이 조금 걸릴 수 있지만 일단 익숙해지면 원하는 설정을 훨씬 쉽게 만들 수 있습니다.

서브플롯을 사용한다면 plotly.com/python/subplots를 읽으세요. 또한 plotly.com/python/graph-objects에서 그래프 객체에 관한 설명도 읽어 보세요.

Plotly와 지도

Plotly에는 지도 관련 도구도 많이 있습니다. 예를 들어 위도와 경도를 나타내는 포인트 집합을 가지고 지도 위에 산포도를 오버레이로 그릴 수 있습니다.

scattergeo 타입

다음 코드는 북미에서 가장 고도가 높은 지역 세 개를 나타내는 지도입니다. 포인트 위에 마우스를 올려 보면 위치와 함께 산 이름이 나타납니다.

```
import plotly.express as px

# (위도, 경도) 형식
peak_coords = [
    (63.069, -151.0063),
    (60.5671, -140.4055),
    (46.8529, -121.7604),
]

# 일치하는 위도, 경도, 이름표 리스트 생성

lats = [pc[0] for pc in peak_coords]
lons = [pc[1] for pc in peak_coords]

peak_names = [
    "Denali",
    "Mt Logan",
    "Mt Rainier"
]
elevations = [20_000, 18_000, 14_000]

# 지도 초기화
title = "Selected High Peaks"
fig = px.scatter_geo(
    lat=lats,
    lon=lons,
    title=title,
    projection="natural earth",
    text=peak_names,
    size=elevations,
    scope="north america",
)

# 옵션 설정
fig.update_layout(titlefont_size=24)
fig.update_traces(
    textposition="middle right",
    textfont_size=18,
)

fig.show()
```

Django Cheet Sheet

Django란?

Django는 파이썬으로 대화형 홈페이지를 만들 때 사용하는 웹 프레임워크입니다. Django를 사용해 홈페이지에서 제공하는 데이터 종류를 정의하고 사용자가 그 데이터를 어떻게 사용할지 결정합니다.

Django는 작은 프로젝트뿐만 아니라 수백만 명의 사용자가 사용하는 대형 사이트도 문제 없이 만들 수 있습니다.

Django 설치

Django는 가상 환경에 설치해서 진행하는 프로젝트를 다른 파이썬 프로젝트에서 분리시키는 게 좋습니다. 대부분의 명령어는 여러분이 활성 가상 환경에 있다고 가정합니다.

가상 환경 생성

```
$ python –m venv ll_env
```

환경 활성화 (macOS와 리눅스)

```
$ source ll_env/bin/activate
```

환경 활성화 (윈도우)

```
> ll_env\Scripts\activate
```

활성 환경에 Django 설치

```
(ll_env)$ pip install Django
```

프로젝트 생성

새 프로젝트를 시작하고 데이터베이스를 생성한 다음 개발 서버를 시작합니다.

새 프로젝트 생성

다음 명령어 마지막의 점을 잊지 마세요.

```
$ django-admin startproject ll_project .
```

데이터베이스 생성

```
$ python manage.py migrate
```

프로젝트 보기

이 명령어를 실행한 다음부터는 http://localhost:8000/에서 프로젝트를 볼 수 있습니다.

```
$ python manage.py runserver
```

새 애플리케이션 생성

Django 프로젝트는 하나 이상의 애플리케이션으로 구성됩니다.

```
$ python manage.py startapp learning_logs
```

모델

Django 프로젝트의 데이터는 모델 집합으로 구성됩니다. 각 모델은 클래스로 표현합니다.

모델 정의

애플리케이션 폴더에 models.py 파일을 만들어 애플리케이션에서 사용할 모델을 정의합니다. __str__() 메서드는 Django가 이 모델의 데이터 객체를 어떻게 표현할지 지정합니다.

```
from django.db import models

class Topic(models.Model):
    """사용자가 배우고 있는 주제"""

    text = models.CharField(max_length=200)
    date_added = models.DateTimeField(
            auto_now_add=True)

    def __str__(self):
        return self.text
```

모델 활성화

모델을 사용하기 위해서는 프로젝트의 settings.py 파일 INSTALLED_APPS 리스트에 애플리케이션을 추가해야 합니다.

```
INSTALLED_APPS = [
    # 내 애플리케이션
    'learning_logs',

    # 기본 Django 애플리케이션
    'django.contrib.admin',
]
```

데이터베이스 마이그레이션

모델이 나타내는 데이터를 저장하기 위해서는 데이터베이스를 수정해야 합니다. 모델을 새로 만들거나 기존 모델을 수정한 다음에는 항상 이 명령어를 실행해야 합니다.

```
$ python manage.py makemigrations learning_logs
$ python manage.py migrate
```

슈퍼유저 생성

슈퍼유저는 프로젝트 전체를 관리할 수 있는 사용자 계정입니다.

```
$ python manage.py createsuperuser
```

모델 등록

Django의 관리자 페이지에서 모델을 등록할 수 있고, 이런 방식은 프로젝트의 데이터를 더 쉽게 관리할 수 있게 합니다. 이를 위해서는 애플리케이션의 admin.py 파일을 수정해야 합니다. http://localhost:8000/admin/에서 관리자 페이지를 확인하세요. 이를 위해서는 슈퍼유저 사용자 계정으로 로그인해야 합니다.

```
from django.contrib import admin

from .models import Topic

admin.site.register(Topic)
```

단순한 홈페이지

사용자는 웹 페이지를 통해 프로젝트를 사용합니다. 프로젝트 홈페이지는 데이터가 없이 단순하게 만들어도 됩니다. 페이지에는 URL, 뷰, 템플릿이 필요합니다.

프로젝트 URL 연결

프로젝트의 메인인 urls.py 파일은 각 애플리케이션에서 사용하는 urls.py 파일의 위치를 지정합니다.

```
from django.contrib import admin
from django.urls import path, include

urlpatterns = [
    path('admin/', admin.site.urls),
    path('', include('learning_logs.urls')),
]
```

애플리케이션 URL

애플리케이션의 urls.py 파일은 Django가 애플리케이션의 URL에 어떤 뷰를 사용할지 지정합니다. 이 파일은 애플리케이션 폴더에 직접 만들어야 합니다.

```
from django.urls import path

from . import views

app_name = 'learning_logs'
urlpatterns = [
    # 홈페이지
    path('', views.index, name='index'),
]
```

간단한 뷰 만들기

뷰는 요청에서 정보를 추출해 브라우저에 데이터를 보내며, 보통 템플릿을 사용합니다. 뷰 함수는 views.py에 저장합니다. 다음의 뷰 함수는 데이터를 사용하지 않으며 템플릿인 index.html을 사용해 홈페이지를 렌더링합니다.

```
from django.shortcuts import render

def index(request):
    """학습 로그 홈페이지"""
    return render(request,
            'learning_logs/index.html')
```

단순한 템플릿

템플릿은 페이지 구조입니다. 템플릿은 html과 템플릿 코드의 혼합입니다. 템플릿 코드는 파이썬과 비슷하지만 파이썬만큼 강력하지는 않습니다. 프로젝트 로직은 .py 파일에 작성해야 하지만, 일부 로직은 템플릿 코드에 쓰는 게 더 적합합니다. 프로젝트 폴더에 templates/ 폴더를 만드세요. templates/ 폴더 안에 애플리케이션과 같은 이름의 폴더를 만들고, 이 폴더에 템플릿 파일을 저장합니다. 학습 로그 홈페이지 템플릿 경로는 learning_logs/templates/learning_logs/index.html입니다.

```
<p>Learning Log</p>
<p>Learning Log helps you keep track of your
learning, for any topic you're learning
about.</p>
```

템플릿 상속

홈페이지의 요소 중 상당수는 모든 섹션에 걸쳐 동일한 모양의 페이지가 반복됩니다. 홈페이지의 부모 템플릿을 만들고 상속을 통해 모든 페이지를 만들면 홈페이지 전체의 형태와 분위기를 쉽게 수정할 수 있습니다.

부모 템플릿

부모 템플릿은 여러 페이지에 공통으로 사용하는 요소를 정의합니다. 이 템플릿에서 정의한 블록을 개별 페이지에서 데이터로 채웁니다.

```
<p>
  <a href="{% url 'learning_logs:index' %}">
    Learning Log
  </a>
</p>

{% block content %}{% endblock content %}
```

자식 템플릿

자식 템플릿은 템플릿 태그 {% extends %}를 사용해서 부모 템플릿의 구조를 상속합니다. 그리고 부모 템플릿에서 정의한 블록에 들어갈 콘텐츠를 지정합니다.

```
{% extends 'learning_logs/base.html' %}

{% block content %}

  <p>
    Learning Log helps you keep track
    of your learning, for any topic you're
    learning about.
  </p>

{% endblock content %}
```

템플릿 들여쓰기

파이썬 코드는 보통 공백 네 칸으로 들여씁니다. 템플릿은 일반적으로 파이썬보다 더 깊이 중첩하므로 공백 두 칸으로 들여쓰기할 때가 많습니다.

다른 모델

기존 모델을 바탕으로 새로운 모델을 만들수 있습니다. ForeignKey 속성은 관련된 두 모델 사이의 연결을 나타냅니다. 애플리케이션에 모델을 새로 추가한 뒤에는 반드시 데이터베이스 마이그레이션이 필요합니다.

외래 키 정의

```
class Entry(models.Model):
    """주제의 학습 로그 항목"""
    topic = models.ForeignKey(Topic,
            on_delete=models.CASCADE)
    text = models.TextField()
    date_added = models.DateTimeField(
            auto_now_add=True)

    def __str__(self):
        return f"{self.text[:50]}..."
```

데이터로 페이지 생성

프로젝트 페이지 대부분은 현재 사용자에 속하는 데이터를 표시해야 합니다.

URL 매개변수

URL에는 보통 데이터베이스의 어떤 데이터에 접근해야 하는지 나타내는 매개변수가 들어갑니다. 다음 URL 패턴은 특정 주제의 ID를 topic_id 매개변수에 할당합니다.

```
urlpatterns = [
    --생략--
    # 단일 주제의 상세 페이지
    path('topics/<int:topic_id>/', views.topic,
            name='topic'),
]
```

뷰에서 데이터 사용

뷰는 URL 매개변수를 사용해 데이터베이스에서 해당 데이터를 가져옵니다. 이 예제의 뷰는 페이지에 표시할 데이터가 포함된 context 딕셔너리를 템플릿에 보냅니다. 사용할 모델을 임포트해야 합니다.

```
def topic(request, topic_id):
    """주제 하나와 그에 연관된 항목을 모두 표시"""
    topic = Topic.objects.get(id=topic_id)
    entries = topic.entry_set.order_by(
            '-date_added')
    context = {
        'topic': topic,
        'entries': entries,
    }
    return render(request,
            'learning_logs/topic.html', context)
```

템플릿에서 데이터 사용

뷰 함수가 context 딕셔너리를 통해 보낸 데이터를 템플릿에서 사용할 수 있습니다. 이 데이터는 템플릿 변수를 통해 접근합니다. 템플릿 변수는 이중 중괄호로 표시합니다.
템플릿 변수 다음의 세로선은 필터를 뜻합니다. 다음 예제에서는 date 필터가 date 객체의 형식을 지정하고, linebreaks 필터는 문단 렌더링에 사용됩니다.

```
{% extends 'learning_logs/base.html' %}

{% block content %}

  <p>Topic: {{ topic }}</p>

  <p>Entries:</p>
  <ul>
  {% for entry in entries %}
    <li>
    <p>
      {{ entry.date_added|date:'M d, Y H:i' }}
    </p>

    <p>
      {{ entry.text|linebreaks }}
    </p>
    </li>
  {% empty %}
    <li>There are no entries yet.</li>
  {% endfor %}
  </ul>

{% endblock content %}
```

개발 서버 재시작

프로젝트를 변경했는데 효과가 없는 것처럼 보인다면 $ python manage.py runserver 명령으로 서버를 재시작해 보세요.

Django 셸

명령행에서 프로젝트 데이터를 열람할 수 있습니다. 이런 기능은 쿼리를 만들거나 코드 예제를 테스트할 때 유용합니다.

셸 세션 시작

```
$ python manage.py shell
```

프로젝트 데이터에 접근

```
>>> from learning_logs.models import Topic
>>> Topic.objects.all()
[<Topic: Chess>, <Topic: Rock Climbing>]
```

```
>>> topic = Topic.objects.get(id=1)
>>> topic.text
'Chess'

>>> topic.entry_set.all()
<QuerySet [<Entry: In the opening phase...>]>
```

사용자와 폼

웹 애플리케이션은 사용자 계정을 통해 사용자가 데이터를 생성하고 사용할 수 있게 합니다. 이런 데이터 중에는 개인적인 데이터도 있고 공개적인 데이터도 있습니다. Django는 폼을 사용해 사용자의 데이터 입력 및 수정을 처리합니다.

사용자 계정

사용자 계정은 전용 애플리케이션을 통해 처리합니다. 사용자 계정 관련 기능은 등록, 로그인, 로그아웃입니다. Django는 이와 관련된 작업 상당 부분을 자동화합니다.

account 애플리케이션

애플리케이션을 만들면 프로젝트의 settings.py 파일의 INSTALLED_APPS 섹션에 'accounts'를 반드시 추가하세요.

```
$ python manage.py startapp accounts
```

account URL

다음과 같이 프로젝트의 urls.py 파일을 수정해야 프로젝트에서 account 애플리케이션을 사용할 수 있습니다.

```
from django.contrib import admin
from django.urls import path, include

urlpatterns = [
    path('admin/', admin.site.urls),
    path('accounts/', include('accounts.urls')),
    path('', include('learning_logs.urls')),
]
```

URL 정의

사용자에게 로그인, 로그아웃, 등록 기능을 제공해야 합니다. users 애플리케이션 폴더에 urls.py 파일을 새로 만드세요.

```
from django.urls import path, include

from . import views

app_name = 'accounts'
urlpatterns = [
    # 기본 인증 url 포함
    path('', include(
            'django.contrib.auth.urls')),

    # 등록 페이지
    path('register/', views.register,
            name='register'),
]
```

로그인 템플릿

로그인 뷰가 기본적으로 제공되긴 하지만 로그인 템플릿은 직접 만들어야 합니다. 다음 템플릿은 단순한 로그인 폼이며 기본적인 에러 메시지를 제공합니다. accounts/ 폴더 안에 templates/ 폴더를 만들고, 그 안에 registration/ 폴더를 만드세요. 다음 코드를 login.html 파일로 저장하세요. 경로는 accounts/templates/registration/login.html이어야 합니다.
{% csrf_token %} 태그는 폼에 자주 이루어지는 공격을 예방합니다. {{ form.as_div }} 요소는 기본 로그인 폼을 적절한 형식으로 표시합니다.

```
{% extends "learning_logs/base.html" %}

{% block content %}

  {% if form.errors %}
  <p>
    Your username and password didn't match.
    Please try again.
  </p>
  {% endif %}

  <form action="{% url 'users:login'
      method="post" %}">

    {% csrf token %}
    {{ form.as_div }}
    <button name="submit">Log in</button>

  </form>

{% endblock content %}
```

settings.py의 로그아웃 리디렉트 설정

이 설정은 사용자가 로그아웃할 때 Django가 사용자를 어디로 보낼지 지정합니다.

```
LOGOUT_REDIRECT_URL = 'learning_logs:index'
```

현재 로그인 상태 표시

base.html 템플릿을 수정해서 사용자가 현재 로그인 상태인지 나타내고 로그인과 로그아웃 페이지 링크를 표시할 수 있습니다. Django는 모든 템플릿에서 사용자 객체를 사용할 수 있습니다. 이 템플릿도 이 객체를 사용합니다.
템플릿에서 user.is_authenticated를 확인해 사용자의 로그인 상태에 맞는 콘텐츠를 표시할 수 있습니다. {{ user.username }} 프로퍼티를 사용해 로그인한 사용자를 환영합니다. 로그인하지 않은 사용자에게는 등록 페이지나 로그인 페이지 링크가 보입니다.

```
<p>
  <a href="{% url 'learning_logs:index' %}">
    Learning Log
  </a>

  {% if user.is_authenticated %}
    Hello, {{ user.username }}.
    <a href="{% url 'accounts:logout' %}">
      Log out
    </a>
  {% else %}
    <a href="{% url 'accounts:register' %}">
      Register
    </a> -
    <a href="{% url 'accounts:login' %}">
      Log in
    </a>
  {% endif %}

</p>

{% block content %}{% endblock content %}
```

로그아웃 폼

Django는 로그아웃 기능도 제공하지만 사용자가 로그아웃하는 폼은 직접 만드는 게 좋습니다. settings.py에 LOGOUT_REDIRECT_URL 설정을 추가하는 걸 잊지 마세요.

```
{% if user.is_authenticated %}
  <form action="{% url 'accounts:logout' %}"
    method='post'>

    {% csrf_token %}
    <button name='submit'>Log out</button>

  </form>
{% endif %}
```

register 뷰

register 뷰는 페이지를 처음 요청할 때 빈 등록 폼을 표시하고, 완료된 등록 폼을 처리해야 합니다.
등록에 성공하면 사용자는 로그인하고 홈페이지로 리디렉트됩니다. 폼이 유효하지 않으면 적절한 에러 메시지와 함께 등록 페이지를 다시 표시합니다.

```
from django.shortcuts import render, redirect
from django.contrib.auth import login
from django.contrib.auth.forms import \
        UserCreationForm

def register(request):
    """사용자 등록"""

    if request.method != 'POST':
        # 빈 등록 폼 표시
        form = UserCreationForm()
```

```
    else:
        # 완료된 폼 처리
        form = UserCreationForm(
                data=request.POST)

        if form.is_valid():
            new_user = form.save()

            # 로그인, 홈페이지로 리디렉트
            login(request, new_user)
            return redirect(
                    'learning_logs:index')

    # 빈 폼, 또는 유효하지 않은 폼 표시
    context = {'form': form}

    return render(request,
        'registration/register.html', context)
```

Django 폼 사용

폼을 만들고 사용하는 방식은 아주 다양합니다. Django에서 기본으로 제공하는 폼을 쓸 수도 있고 여러분이 직접 만들 수도 있습니다. 모델폼(ModelForm)을 사용하면 사용자가 여러분의 모델에 맞는 데이터를 입력할 수 있습니다. 모델에 맞는 데이터를 입력할 필드는 자동으로 생성됩니다.

이 치트 시트 후반에 있는 register 뷰는 폼 처리를 단순화한 뷰입니다. 이 뷰는 폼에서 데이터를 받지 않으면 빈 폼을 렌더링하고, 폼에서 POST 데이터를 받으면 유효성 검사를 하고 데이터베이스에 저장합니다.

프로젝트 스타일

django-bootstrap5는 부트스트랩 라이브러리를 사용해 프로젝트를 멋지게 꾸밉니다. 이 애플리케이션은 페이지의 각 요소에 스타일을 적용할 수 있는 태그를 제공합니다. 더 자세한 내용은 django-bootstrap5.readthedocs.io/를 읽어보세요.

프로젝트 배포

plaform.sh에서는 프로젝트를 실제 서버에 푸시해 인터넷에 연결된 모든 사람이 사용할 수 있게 만듭니다. plaform.sh는 무료 평가판 기간도 제공합니다.

계속 진행하려면 plaform.sh 명령행 도구를 설치하고, 깃을 설치해 프로젝트 이력을 추적할 수 있어야 합니다. 더 자세한 내용은 https://platform.sh/marketplace/django를 보세요.

register 템플릿

다음 register.html 템플릿은 단순한 등록 폼을 표시합니다.

```
{% extends 'learning_logs/base.html' %}

{% block content %}

  <form action="{% url 'accounts:register'
      method='post' %}">

    {% csrf_token %}
    {{ form.as_div }}

    <button name='submit'>Register</button>

  </form>

{% endblock content %}
```

사용자와 데이터 연결

사용자가 만드는 데이터 일부는 그들에게 속합니다. 사용자와 직접 연결되어야 하는 모델에는 그 인스턴스를 특정 사용자와 연결할 수 있는 필드가 있어야 합니다.

주제와 사용자 연결

계층 구조에서 최상위에 있는 데이터만 사용자와 직접 연결하면 됩니다. 이를 위해 user 모델을 임포트하고 데이터 모델에 외래 키로 추가합니다.
모델을 수정하면 데이터베이스를 마이그레이션해야 합니다. 기존의 인스턴스와 연결할 사용자 ID를 선택해야 합니다.

```
from django.db import models
from django.contrib.auth.models import User

class Topic(models.Model):
    """사용자가 배우고 있는 주제"""

    text = models.CharField(max_length=200)
    date_added = models.DateTimeField(
            auto_now_add=True)

    owner = models.ForeignKey(User,
            on_delete=models.CASCADE)

    def __str__(self):
        return self.text
```

현재 사용자의 데이터 쿼리

뷰의 요청 객체에는 user 속성이 있습니다. 이 속성을 통해 사용자 데이터를 가져올 수 있습니다. 다음 filter() 메서드는 현재 사용자에게 속하는 데이터만 가져옵니다.

```
topics = Topic.objects.filter(
        owner=request.user)
```

로그인하지 않은 사용자의 접근 제한

일부 페이지는 등록된 사용자만 사용할 수 있습니다. 이런 페이지는 @login_required 데코레이터로 보호됩니다. 이 데코레이터를 사용한 뷰는 로그인하지 않은 사용자를 자동으로 적절한 페이지로 리디렉트합니다. 다음 코드는 views.py 파일 예제입니다.

```
from django.contrib.auth.decorators import \
        login_required
--생략--

@login_required
def topic(request, topic_id):
    """주제 하나와 그에 연관된 항목을 모두 표시"""
```

리디렉트 URL 설정

@login_required 데코레이터는 로그인하지 않은 사용자를 로그인 페이지로 리디렉트합니다. 다음 행을 프로젝트의 settings.py 파일에 추가해 로그인 페이지를 지정합니다.

```
LOGIN_URL = 'accounts:login'
```

강제 접근 방지

일부 페이지는 URL 매개변수를 바탕으로 데이터를 가져옵니다. 현재 사용자가 요청된 데이터의 소유자인지 확인하고, 그렇지 않다면 404 에러를 일으킬 수 있습니다. 다음 예제를 보세요.

```
from django.http import Http404
--생략--

@login_required
def topic(request, topic_id):
    """주제 하나와 그에 연관된 항목을 모두 표시"""
    topic = Topics.objects.get(id=topic_id)
    if topic.owner != request.user:
        raise Http404
    --생략--
```

폼에서 데이터 수정

초기 데이터를 지정하면 Django는 사용자의 데이터를 사용해 폼을 생성합니다. 사용자는 이를 보고 데이터를 수정할 수 있습니다.

초기 데이터가 있는 폼

instance 매개변수를 통해 폼의 초기 데이터를 지정합니다.

```
form = EntryForm(instance=entry)
```

저장하기 전에 데이터 수정

commit=False 인수를 사용해 데이터를 데이터베이스에 기록하기 전에 변경할 수 있습니다.

```
new_topic = form.save(commit=False)
new_topic.owner = request.user
new_topic.save()
```

온라인 자료

docs.djangoproject.com/에서 Django 문서를 볼 수 있습니다. Django 문서는 아주 자세하면서도 사용자 친화적이니 꼭 읽어보세요.

깃 Cheet Sheet

버전 관리

버전 관리 소프트웨어를 사용하면 프로젝트에 문제가 없는 상태의 스냅샷을 찍을 수 있습니다. 프로젝트가 망가지면 잘 동작하는 최근 상태로 돌아갈 수 있습니다.

즉 버전 관리가 중요한 이유는 전체 프로젝트를 망칠 걱정 없이 자유롭게 새로운 아이디어를 시도해 볼 수 있다는 겁니다. 깃허브 같은 분산형 버전 관리 시스템을 사용하면 다른 개발자와의 협업도 쉬워집니다.

깃 설치

git-scm.com/에서 여러분의 컴퓨터에 맞는 설치 파일을 찾을 수 있습니다. 하지만 그 전에 깃이 이미 설치되어 있는지 확인하세요.

```
$ git --version
git version 2.30.1 (Apple Git-130)
```

깃 설정

일부 설정을 마치면 깃을 더 쉽게 사용할 수 있습니다. 예를 들어 editor 설정을 하면 깃은 여러분이 텍스트를 입력해야 할 때 해당 에디터를 불러옵니다.

모든 설정 확인

```
$ git config --list
```

사용자 이름 설정

```
$ git config --global user.name "eric"
```

이메일 설정

```
$ git config --global user.email
    "eric@example.com"
```

에디터 설정

```
$ git config --global core.editor "nano"
```

폴더 무시

.gitignore 파일을 만들어 무시할 파일을 지정합니다. 이 파일은 점으로 시작하고, 확장자는 없습니다. 그런 다음 이 파일에 무시할 파일과 폴더를 기록하면 됩니다.

폴더 무시

```
__pycache__/
my_venv/
```

특정 파일 무시

```
.DS_Store
secret_key.txt
```

특정 확장자의 파일 무시

```
*.pyc
```

저장소 초기화

깃은 저장소 관리에 필요한 파일을 모두 숨긴 폴더 .git에 저장합니다. 이 폴더를 삭제하면 프로젝트 이력도 모두 삭제됩니다.

저장소 초기화

```
$ git init
Initialized empty Git repository in
    my_project/.git/
```

상태 확인

프로젝트 상태는 자주 확인해야 합니다. 커밋한 내용이 전혀 없더라도 주기적으로 상태를 확인하는 게 좋습니다. 상태 확인을 통해 깃이 어떤 파일을 모니터링하는지 볼 수 있습니다.

상태 확인

```
$ git status
On branch main
No commits yet
Untracked files:
    .gitignore
    hello.py
    ...
```

파일 추가

깃이 파일을 추적하게 하려면 이들을 추가해야 합니다. .gitignore에 기록한 파일을 제외하고 모든 파일을 추가하려면 다음과 같이 명령합니다.

.gitignore에 없는 파일 추가

```
$ git add .
```

파일 하나만 추가

```
$ git add hello.py
```

커밋

커밋 명령의 -am 플래그는 추가된 파일을 모두 커밋하면서 커밋 메시지를 기록합니다(커밋하기 전에 항상 상태를 확인하세요).

메시지와 함께 커밋

```
$ git commit -am "Started project, everything
    works."
2 files changed, 7 insertions(+)
create mode 100644 .gitignore
create mode 100644 hello.py
```

로그 확인

깃은 여러분의 커밋을 모두 기록합니다. 로그를 확인하면 프로젝트 이력을 쉽게 이해할 수 있습니다.

기본 형식으로 로그 확인

```
$ git log
commit dc2ebd6... (HEAD -> main)
Author: Eric Matthes <eric@example.com>
Date:   Feb 27 11:27:07 2023 -0900
    Greets user.
commit bf55851...
...
```

단순한 형식으로 로그 확인

```
$ git log --oneline
dc2ebd6 (HEAD -> main) Greets uer.
bf55851 Started project, everything works.
```

이력 탐색

특정 커밋 해시에 방문하거나 프로젝트 헤드를 참조해 프로젝트 이력을 탐색할 수 있습니다. 여기서 헤드란 현재 브랜치의 최근 커밋을 말합니다.

특정 커밋 방문

```
$ git checkout b9aedbb
```

메인 브랜치의 최근 커밋으로 복귀

```
$ git checkout main
```

이전 커밋 방문

헤드 체크아웃은 설명하지 않았습니다. https://mytory.net/archives/10078, https://velog.io/@jhyeom1545/git-9-HEAD-checkout-fetch-pull의 글을 참고하세요.

```
$ git checkout HEAD^
```

이전 커밋 방문 또 다른 방법

```
$ git checkout HEAD~1
```

더 일찍 커밋 방문

```
$ git checkout HEAD^^^
```

더 일찍 커밋 방문 또 다른 방법

```
$ git checkout HEAD~3
```

추가 학습

git help 명령으로 깃에 대해 더 알아볼 수 있습니다. 또한 스택 오버플로우에 방문해 git을 검색하고 좋아요 순으로 정렬해서 보는 것도 좋은 방법입니다.

브랜치

프로젝트를 변경하려 하는데 이 과정에서 커밋을 여러 번 할 것 같다면 브랜치를 따로 만들어 실험할 수 있습니다. 브랜치를 병합merge하기 전에는 해당 브랜치의 변경 내용은 메인 브랜치와 섞이지 않습니다. 브랜치를 메인 브랜치에 병합한 뒤에는 일반적으로 삭제합니다.

브랜치는 프로젝트를 독립적인 여러 버전으로 유지하려 할 때도 사용합니다.

브랜치를 새로 만들고 그 브랜치로 이동

```
$ git checkout -b new_branch_name
Switched to a new branch 'new_branch_name'
```

모든 브랜치 확인

```
$ git branch
main
* new_branch_name
```

다른 브랜치로 이동

```
$ git checkout main
Switched to branch 'main'
```

변경 내용 병합

```
$ git merge new_branch_name
Updating b9aedbb..5e5130a
Fast-forward
 hello.py | 5 +++++
 1 file changed, 5 insertions(+)
```

브랜치 삭제

```
$ git branch -D new_branch_name
Deleted branch new_branch_name
    (was 5e5130a).
```

마지막 커밋을 새로운 브랜치로 이동

```
$ git branch new_branch_name
$ git reset --hard HEAD~1
$ git checkout new_branch_name
```

최근 변경 취소

언제든 프로젝트가 동작하던 최근 상태로 돌아가 거기에서 다시 시작할 수 있다는 게 버전 관리의 핵심 중 하나입니다.

커밋되지 않는 변경 내용 모두 제거

```
$ git checkout .
```

특정 커밋 이후의 변경 내용 모두 제거

```
$ git reset --hard b9aedbb
```

이전 커밋에서 시작하는 새로운 브랜치 생성

```
$ git checkout -b branch_name b9aedbb
```

변경 내용 임시 보관

커밋은 하지 않고 변경 내용을 임시 저장stash할 수 있습니다. 이 기능은 새로 커밋하지 않고 최근 커밋을 다시 방문하려 할 때 유용합니다. 임시 저장에는 제한이 없습니다.

마지막 커밋 이후의 변경 내용 임시 저장

```
$ git stash
Saved working directory and index state
    WIP on main: f6f39a6...
```

임시 저장된 내용 확인

```
$ git stash list
stash@{0}: WIP on main: f6f39a6...
stash@{1}: WIP on main: f6f39a6...
...
```

최근 임시 저장의 변경 내용 적용

```
$ git stash pop
```

특정 임시 저장의 변경 내용 적용

```
$ git stash pop --index 1
```

임시 저장 이력 삭제

```
$ git stash clear
```

커밋 비교

프로젝트의 상태들을 비교해 봐야 할 때도 있습니다.

마지막 커밋 이후의 변경 내용 모두 표시(diff는 difference의 약자입니다)

```
$ git diff
```

파일 하나에서 마지막 커밋 이후의 변경 내용 표시

```
$ git diff hello.py
```

특정 커밋 이후의 변경 내용 표시

```
$ git diff HEAD~2
$ git diff HEAD^^
$ git diff fab2cdd
```

두 커밋 사이의 변경 내용 확인

```
$ git diff fab2cdd 7c0a5d8
```

파일 하나에서 두 가지 커밋 비교

```
$ git diff fab2cdd 7c0a5d8 hello.py
```

좋은 커밋 습관

프로젝트를 변경하고 의도한 대로 동작할 때마다 커밋하세요. 어떤 부분을 변경했는지 간결하게 설명하는 커밋 메시지를 남기세요. 새로운 기능을 구현하거나 버그 수정을 시작했다면 새로운 브랜치에서 시도하는 것도 좋은 방법입니다.

깃과 깃허브

깃허브는 코드를 공유하고 협업하는 플랫폼입니다. 깃허브에 공개된 프로젝트는 모두 클론할 수 있습니다. 깃허브에 계정이 있으면 여러분의 프로젝트를 업로드할 수 있고 공개 또는 비공개로 설정할 수 있습니다.

저장소를 로컬 컴퓨터로 클론

```
$ git clone
    https://github.com/ehmatthes/pcc_3e.git/
Cloning into 'pcc_3e'...
...
Resolving deltas: 100% (1503/1503), done.
```

로컬 프로젝트를 깃허브 저장소에 푸시

먼저 깃허브에 빈 저장소를 만들어야 합니다.

```
$ git remote add origin
  https://github.com/username/hello_repo.git
$ git push -u origin main
Enumerating objects: 10, done.
...
To https://github.com/username/hello_repo.git
 * [new branch]      main -> main
Branch 'main' set up to track remote branch
    'main' from 'origin'.
```

최근 변경 내용을 깃허브 저장소에 푸시

```
$ git push origin branch_name
```

풀 리퀘스트

깃허브 프로젝트의 메인 브랜치에 다른 브랜치의 변경 내용을 적용하고 싶다면 풀 리퀘스트를 보낼 수 있습니다. 여러분의 저장소에서 풀 리퀘스트를 연습하고 싶다면 새로운 브랜치를 만드세요. 작업이 끝나면 그 브랜치를 여러분의 저장소로 푸시하세요. 그런 다음 깃허브의 '풀 리퀘스트' 탭으로 이동하고, 병합할 브랜치에서 'Compare & pull request'를 클릭하세요. 준비가 되면 'Merge pull request'을 클릭하세요.

`git pull origin main` 명령으로 이 변경 내용을 다시 여러분의 로컬 메인 브랜치에 가져올 수 있습니다. 변경 내용을 로컬에서 메인 브랜치에 병합한 다음 이를 깃허브 메인 브랜치에 푸시하는 것과 결과적으로는 같습니다.

깃 연습

혼자 개발할 때는 깃의 단순한 기능만 사용해도 됩니다. 규모가 큰 프로젝트에서 협업할 때는 고급 기능을 사용하는 것이 좋습니다. 언제든 폐기할 수 있는 연습용 프로젝트를 만들어 깃을 충분히 연습하세요. 가치 있는 경험이 될 겁니다. 깃이 어떻게 동작하는지 명확히 파악하려면 프로젝트가 여러 파일로 구성되어야 하고, 중첩된 폴더도 있어야 할 겁니다.