codingOn x posco

K-Digital Training 신재생에너지 활용 IoT 과정

codingOn

Python 자료형

오늘 수업은?

- 지난시간에는 파이썬의 변수, 연산자, 문자열 등 아주 기초적인 내용에 대해서 공부하였습니다.
- 이번시간에는 변수에 넣을 수 있는 다양한 자료형에 대해서 더 자세히 공부하도록 하겠습니다.

학습목표

- 리스트와 2차원 리스트에 대해서 이해한다.
- 튜플에 대해서 이해한다.
- 셋에 대해서 이해한다.
- 딕셔너리에 대해서 이해한다.

codingOn

1. 리스트(list)

리스트

- 리스트 사용의 필요성
  - 예) 학생의 이름을 저장해야한다고 가정

  ⇒ 각각의 이름을 별도의 변수에 저장해야함

  ⇒ student1 = "홍길동", student2 = "임꺽정", student3 = "성춘향"

  ⇒ 각각의 변수에 담게 되어 관리가 어려움(학생이 100명이라면 100개의 변수생성)

  ⇒ 이때 리스트를 사용하게 되면 하나의 변수에 모든 데이터를 저장하여 사용할 수 있음

리스트

- 리스트란?
  - 지금까지 변수는 1개의 값만을 할당하고 재할당할 수 있었음
  - 여러 개의 연속적인 값을 저장하고자 할 때 사용하는 자료형
  - 리스트는 대괄호([])로 표현, 쉼표(,) 로 구분하여 작성
  - 리스트는 순서가 중요
- 리스트 생성
  - **리스트 식별자명 = [요소1, 요소2, 요소3…]**
  - season = [" 봄", "여름", "가을", "겨울"]
  - number = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
  - students = ["홍길동", "임꺽정", "성춘향", "이몽룡", "전우치"]

특징

codingOn

- 새로운 항목 추가/삭제 가능
- 항목에 대한 순서가 있음 - 모든 요소가 번호(인덱스 값)를 가지고 있음
- 인덱스 값은 0 부터 시작
- 항목 검색 가능 - 인덱스 값으로 특정 가능
- 같은 자료형으로 구성될 필요 없음! - 숫자형, 문자열 혼합하여 사용 가능

```
number = [1, 5, 3, 4, 8, 2, 3]
print(number[3]) # 4
```

- list() 함수로 리스트 생성 가능

슬라이싱

codingOn

- 문자열과 동일하게 인덱스로 슬라이싱 가능

```
shop = ["반팔", "청바지", "이어폰", "키보드"]

# 인덱싱
print(shop[0]) # 반팔

# 슬라이싱
print(shop[0:2]) # ['반팔', '청바지'] 0 <= shop < 2

# 리스트 안에서 리스트 인덱싱
my_shop = ["반팔", "청바지", "이어폰", ["무선 키보드", "유선 키보드", "기계식 키보드"]]
print(my_shop[3]) # ['무선 키보드', '유선 키보드', '기계식 키보드']
print(my_shop[-1]) # ['무선 키보드', '유선 키보드', '기계식 키보드']
print(my_shop[2:4]) # ['이어폰', ['무선 키보드', '유선 키보드', '기계식 키보드']]
```

값 수정

- 리스트에 있는 값을 수정할때는 인덱스로 접근하여 값을 수정
- 주의) 리스트 길이를 넘는 인덱스로 접근하면 IndexError 발생

```
shop = ["반팔", "청바지", "이어폰", "키보드"]

shop[0] = "긴팔"
print(shop) # ['긴팔', '청바지', '이어폰', '키보드']

shop[100] = "신발"
print(shop) # IndexError: list assignment index out of range
```

값 삭제

- del 식별자[인덱스] : 인덱스 위치 값 삭제
- del 식별자[idx:idx] : 슬라이싱으로 여러 요소 한 번에 삭제 가능

```
shop = ["반팔", "청바지", "이어폰", "키보드"]
print(shop) # ['반팔', '청바지', '이어폰', '키보드']

# 리스트 삭제
del shop[1]
print(shop) # ['반팔', '이어폰', '키보드']
del shop[2:]
print(shop) # ['반팔', '이어폰']
```

연산

- 더하기 +
- 반복 *

```
a = [1, 2, 3]
b = [4, 5]

print(a + b) # [1, 2, 3, 4, 5]
print(a * 2) # [1, 2, 3, 1, 2, 3]
```

codingOn

2. 리스트 함수

정렬 함수

codingOn

- sorted(리스트) : 오름차순 정렬 함수
- 내림차순으로 정렬하고 싶을때는 reverse=True를 넣으면 됨
- 원본을 변경하지 않고 새로운 리스트를 반환
  - 숫자정렬

```
num = [3, 1, 5, 2]
num_asc = sorted(num)
print(num_asc) # [1, 2, 3, 5]
print(num) # [3, 1, 5, 2]
```

```
num = [3, 1, 5, 2]
num_desc = sorted(num, reverse=True)
print(num_desc) # [5, 3, 2, 1]
print(num) # [3, 1, 5, 2]
```

  - 문자정렬

```
alphabet = ['b', 'c', 'a', 'd']
print(sorted(alphabet)) # ['a', 'b', 'c', 'd']
print(alphabet) # ['b', 'c', 'a', 'd']

korean = ['강', '이', '박', '최']
print(sorted(korean)) # ['강', '박', '이', '최']
print(korean) # ['강', '이', '박', '최']
```

```
alphabet = ['b', 'c', 'a', 'd']
print(sorted(alphabet, reverse=True)) # ['d', 'c', 'b', 'a']
print(alphabet) # ['b', 'c', 'a', 'd']

korean = ['강', '이', '박', '최']
print(sorted(korean, reverse=True)) # ['최', '이', '박', '강']
print(korean) # ['강', '이', '박', '최']
```

정렬 메서드

codingOn

- sort() : 오름차순 정렬
- 내림차순으로 정렬하고 싶을때는 reverse=True를 넣으면 됨
- 새로운 리스트를 반환하지 않고 원본 리스트가 정렬됨
  - 숫자정렬

```
num = [3, 1, 5, 2]
num.sort()
print(num) # [1, 2, 3, 5]
```

```
num = [3, 1, 5, 2]
num.sort(reverse=True)
print(num) # [5, 3, 2, 1]
```

  - 문자정렬

```
alphabet = ['b', 'c', 'a', 'd']
alphabet.sort()
print(alphabet) # ['a', 'b', 'c', 'd']

korean = ['강', '이', '박', '최']
korean.sort()
print(korean) # ['강', '박', '이', '최']
```

```
alphabet = ['b', 'c', 'a', 'd']
alphabet.sort(reverse=True)
print(alphabet) # ['d', 'c', 'b', 'a']

korean = ['강', '이', '박', '최']
korean.sort(reverse=True)
print(korean) # ['최', '이', '박', '강']
```

정렬 메서드

codingOn

- reverse() : 리스트 요소를 역순으로 정렬
  - 숫자 정렬

```
num = [3, 1, 5, 2]
num.reverse()
print(num) # [2, 5, 1, 3]
```

  - 문자 정렬

```
alphabet = ['b', 'c', 'a', 'd']
alphabet.reverse()
print(alphabet) # ['d', 'a', 'c', 'b']

korean = ['강', '이', '박', '최']
korean.reverse()
print(korean) # ['최', '박', '이', '강']
```

위치 찾기 메서드

codingOn

- 만약에 리스트 개수가 100개가 넘는다면, 중간 정도에 있는 아이템을 찾기 위해 다 셀 수 없음
- 리스트에서 요소가 어디에 위치해 있는지 인덱스값을 찾아 반환
- 중복된 값이 있으면 제일 앞에 위치 반환
  - index(찾을요소)
- 리스트에 찾을 요소가 없으면 오류가 발생

```
a = ['q', 'w', 'e', 'r', 'w']
print(a.index('w')) # 1
print(a.index('a')) # ValueError: 'a' is not in list
```

추가/삭제 메서드

- append() : 맨 마지막에 요소 추가
- pop() : 맨 마지막에 요소 삭제
- pop(idx): idx번째 인덱스 요소 삭제
- remove('a') : 리스트에서 특정 요소 삭제
- insert(idx, 'a') : idx번째 위치에 요소 삽입
- clear() : 모든 요소삭제

```
a = ['a', 'b', 'c', 'd']
a.append('e')
print(a) # ['a', 'b', 'c', 'd', 'e']

a.pop()
print(a) # ['a', 'b', 'c', 'd']

a.pop(1)
print(a) # ['a', 'c', 'd']

a.remove('a')
print(a) # ['c', 'd']

a.insert(1, "a")
print(a) # ['c', 'a', 'd']

a.clear()
print(a) # []
```

개수 세기 메서드

- count(item) : 같은 이름 가진 요소 개수 세기

```
a = ['q', 'w', 'e', 'r', 'w']
print(a.count('w')) # 2
```

실습. 리스트 연습문제

codingOn

다음과 같은 리스트가 있을 때 결과를 출력해보세요.

```
rainbow = ['red', 'orange', 'yellow', 'green', 'blue', 'indigo', 'purple']

# 1. 2번 인덱스 값 출력
# 2. 알파벳 순서로 정렬한 값 출력
# 3. 좋아하는 색 맨 마지막에 추가하기
# 4. 3~6번째 값 삭제하기
```

codingOn

3. 이차원 리스트

이차원 리스트

- 리스트안에 리스트를 넣을 수 있음
- 행(row)과 열(column)로 이루어진 데이터를 표현
- 리스트변수 = [[요소1, 요소2 …], [요소1, 요소2 …]]
  - 행(row): 각 리스트가 한 행
  - 열(column): 각 리스트의 요소가 해당 열
- 2차원 리스트의 사용
  - **행렬 연산**: 수학적 행렬 연산이나 그래프 이론에서 데이터를 저장할 때
  - **표 데이터 처리**: 엑셀 시트와 같은 표 형식 데이터를 다룰 때
  - **이미지 처리**: 이미지 데이터를 픽셀 단위로 저장할 때

추가,수정,삭제

codingOn

```python
# 3 x 3
matrix = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]
value = matrix[1][2]
# 2번째 행의 3번째 열
print(value)  # 6 출력
```

matrix[row][column]

```python
# 요소 추가
matrix[1] = matrix[1] + [100]
print(matrix)  # [[1, 2, 3], [4, 5, 6, 100], [7, 8, 9]]
# 행 추가
matrix = matrix + [[10, 11, 12]]
print(matrix)  # [[1, 2, 3], [4, 5, 6, 100], [7, 8, 9], [10, 11, 12]]
# 요소 수정
matrix[0][0] = 11
matrix[1][1] = 55
matrix[2][2] = 99
print(matrix)  # [[11, 2, 3], [4, 55, 6, 100], [7, 8, 99], [10, 11, 12]]
# 행 삭제
del matrix[1]
print(matrix)  # [[11, 2, 3], [7, 8, 99], [10, 11, 12]]
# 행의 개수
rows = len(matrix)
# 열의 개수 (첫 번째 행의 길이로 판단)
cols = len(matrix[0])
print(f"행: {rows}, 열: {cols}")  # 행: 3, 열: 3
```

메서드

- append() : 요소 전체가 맨마지막에 추가

```
matrix = [[1, 2, 3], [4, 5, 6]]
# 요소 추가
matrix[0].append(10)
print(matrix)  # [[1, 2, 3, 10], [4, 5, 6]]
# 행 추가
matrix.append([7, 8, 9])
print(matrix)  # [[1, 2, 3, 10], [4, 5, 6], [7, 8, 9]]
```

- insert() : 특정위치에 삽입

```
matrix = [[1, 2, 3], [4, 5, 6]]
# 0행 1번에 요소 추가
matrix[0].insert(1, 99)
print(matrix)  # [[1, 99, 2, 3], [4, 5, 6]]
# 인덱스 1번에 행 추가
matrix.insert(1, [10, 11, 12])
print(matrix)  # [[1, 99, 2, 3], [10, 11, 12], [4, 5, 6]]
```

메서드

- extend() : 맨마지막에 여러 개 추가

```
matrix = [[1, 2], [4, 5]]
# 기존 행에 요소 확장
matrix[0].extend([3, 4])
print(matrix)  # [[1, 2, 3, 4], [4, 5]]

my_list = [1, 2, 3]
# 다른 리스트를 확장하여 추가
my_list.extend([4, 5, 6])
print(my_list) # [1, 2, 3, 4, 5, 6]

my_list = ['a', 'b', 'c']
# 문자열을 리스트에 확장하여 추가
my_list.extend('def')
print(my_list) # ['a', 'b', 'c', 'd', 'e', 'f']
```

메서드

- extend() 와 append() 차이

```
# append() 사용
my_list = [1, 2, 3]
my_list.append([4, 5])
print(my_list)  # [1, 2, 3, [4, 5]]  -> 리스트 전체가 하나의 요소로 추가됨

# extend() 사용
my_list = [1, 2, 3]
my_list.extend([4, 5])
print(my_list)  # [1, 2, 3, 4, 5]  -> 리스트의 각 요소가 확장됨
```

4. 튜플(Tuple)

튜플

- 리스트와 비슷하지만 ()를 사용하며 , 로 구분
- 불변: 튜플에 저장된 값은 수정, 추가, 삭제 할 수 없음
- 리스트보다 상대적을 적은 메모리를 사용하며 처리 속도도 빠름
- 변경이 불필요하거나 고정된 데이터 처리에 적합
- 튜플변수명 = (요소1, 요소2, 요소3, …)

튜플

codingOn

- 튜플 생성 예시

```
t1 = (1,)  # 요소 1개만있을시 , 필수
t2 = (1, 2, 3, 4, 3, 3, 5, 6)
t3 = 1, 2, 3    # () 생략 가능
t4 = ('a', 'b', ('ab', 'cd'), 'c')

print(t4[2][1])  # cd
print(t4[0:2])  # ('a', 'b')
print(t3[1])  # 2
print(t2.count(3))  # 3
print(t4.index('c'))  # 3
print(t1 + t3)  # (1, 1, 2, 3)
print(t3 * 3)  # (1, 2, 3, 1, 2, 3, 1, 2, 3)
print(len(t4))  # 4
print(7 in t2) # False
```

codingOn

5. 셋(Set)

셋(Set)

- 셋은 중복을 허용하지 않고 순서가 없는 수학의 집합과 같은 개념의 자료형
- 요소의 순서가 없어 출력을 해보면 매번 요소의 순서가 변경됨
- 또한 리스트나 튜플 처럼 인덱스를 사용하여 접근 할 수 없음
- 동일한 값이 여러 번 추가되면 중복된 값을 제거하고 하나의 값만 저장
- 셋은 요소를 추가, 삭제 가능함
- {} 와 set() 을 이용하여 생성

```
s1 = {1, 2, 3, 3, 4, 4}
print(s1) # {1, 2, 3, 4}
s2 = set((1, 2, 3, 3, 4, 5, 6, 6 ))
print(s2) # {1, 2, 3, 4, 5, 6}
s3 = set(["가", "나", "다", "다", "라", "나"])
print(s3) # {'라', '다', '가', '나'}
```

메서드

- add(), update(), remove(), discard(), clear() 사용가능

```
s1 = {1, 2, 3, 3, 4}

# 추가
s1.add(5)
print(s1)  # {1, 2, 3, 4, 5}
# 업데이트
s1.update([4, 5, 6, 7, 8])
print(s1)  # {1, 2, 3, 4, 5, 6, 7, 8}
# 원하는 요소 삭제
s1.remove(3)
print(s1)  # {1, 2, 4, 5, 6, 7, 8}
# 요소 삭제(요소가 없으시 아무일도 안남)
s1.discard(9)  # 아무일도 안남
# s1.remove(9) # KeyError: 9
s1.discard(1)
print(s1)  # {2, 4, 5, 6, 7, 8}
# 모든 요소 삭제
s1.clear()
print(s1)  # set()
```

연산

- 합집합 : | 또는 union() 메서드
- 교집합 : & 또는 intersection() 메서드
- 차집합 : - 또는 difference()

```
s1 = {1, 2, 3, 4, 5}
s2 = {4, 5, 6, 7, 8}

# 합집합
s3 = s1 | s2
print(s3) # {1, 2, 3, 4, 5, 6, 7, 8}
s3 = s1.union(s2)
print(s3) # {1, 2, 3, 4, 5, 6, 7, 8}

# 교집합
s3 = s1 & s2
print(s3) # {4, 5}
s3 = s1.intersection(s2)
print(s3) # {4, 5}
```

```
s1 = {1, 2, 3, 4, 5}
s2 = {4, 5, 6, 7, 8}

# 차집합
s3 = s1 - s2
print(s3) # {1, 2, 3}
s3 = s1.difference(s2)
print(s3) # {1, 2, 3}
s4 = s2 - s1
print(s4) # {8, 6, 7}
s4 = s2.difference(s1)
print(s4) # {8, 6, 7}
```

6. 딕셔너리(Dictionary)

딕셔너리

codingOn

- 딕셔너리는 키(key)/값(value)의 쌍으로 요소를 갖는 자료형
- 키는 하나의 딕셔너리에 중복되어 사용할 수 없음. 고유해야함
- {}나 dict() 이용해서 만들며 여러 개일 경우 , 로 구분
- 딕셔너리변수 = { 키1: 값1, 키2: 값2 … }

```
# 빈 딕셔너리 생성
dict1 = {}
print(dict1) # {}
dict1 = dict()
print(dict1) # {}

# 키-값 쌍으로 딕셔너리 생성
dict1 = {
    "name": "홍길동",
    "age": 30,
    "city": "서울"
}
print(dict1) # {'name': '홍길동', 'age': 30, 'city': '서울'}

# dict() 함수를 사용해 생성
dict1 = dict(name="홍길동", age=30, city="서울")
print(dict1) # {'name': '홍길동', 'age': 30, 'city': '서울'}
```

접근, 추가, 삭제

codingOn

```
dict1 = { "name" : "홍길동", 1: "a" , "city" : "서울", "hobby": ["캠핑", "등산"] }
# 값 사용
print(dict1) # {'name': '홍길동', 1: 'a', 'city': '서울', 'hobby': ['캠핑', '등산']}
print(dict1[1]) # a
print(dict1["city"]) # 서울
print(dict1["hobby"][1]) # 등산
# 값 변경
dict1["city"] = "인천"
print(dict1) # {'name': '홍길동', 1: 'a', 'city': '인천', 'hobby': ['캠핑', '등산']}
dict1[1] = "A"
print(dict1) # {'name': '홍길동', 1: 'A', 'city': '인천', 'hobby': ['캠핑', '등산']}
# 값 추가
dict1["생년월일"] = 20020625
print(dict1) # {'name': '홍길동', 1: 'A', 'city': '인천', 'hobby': ['캠핑', '등산'], '생년월일': 20020625}
dict1["hobby"] = ["캠핑", "등산", "런닝"]
print(dict1) # {'name': '홍길동', 1: 'A', 'city': '인천', 'hobby': ['캠핑', '등산', '런닝'], '생년월일': 20020625}
# 값 삭제
del dict1["city"]
print(dict1) # {'name': '홍길동', 1: 'A', 'hobby': ['캠핑', '등산', '런닝'], '생년월일': 20020625}
```

메서드

- get(key): 키가 존재하지 않을 때 기본값을 반환하며 오류를 방지
- update(): 새로운 키-값 쌍을 추가하거나 기존 키의 값을 업데이트
- keys(): 딕셔너리의 모든 키를 반환
- values(): 딕셔너리의 모든 값을 반환
- items(): 딕셔너리의 (키, 값) 쌍을 튜플 형태로 반환
- update(): 딕셔너리 모두 지우기

메서드 예문

codingOn

```
# 값 사용
fruits = {'apple': '사과', 'banana': '바나나'}
print(fruits.get('apple', '애플'))  # 사과
print(fruits.get('peach'))  # None
print(fruits.get('peach', '복숭아'))  # 복숭아
# 여러 요소 추가
fruits.update({'grapes': "포도", 'melon': '멜론'})
# {'apple': '사과', 'banana': '바나나', 'grapes': '포도', 'melon': '멜론'}
print(fruits)
# 키 사용
print(fruits.keys())  # dict_keys(['apple', 'banana', 'grapes', 'melon'])
# 값 반환
print(fruits.values())  # dict_values(['사과', '바나나', '포도', '멜론'])
# 튜플형태로 반환
print(fruits.items())
# dict_items([('apple', '사과'), ('banana', '바나나'), ('grapes', '포도'), ('melon', '멜론')])
# 모두 지우기
fruits.clear()
print(fruits)  # {}
```

실습. 성적관리

1. 학생들의 점수를 저장하는 학생 딕셔너리를 생성한다.
2. "Alice", "Bob", "Charlie" 세 명의 학생을 key로 갖고, 각각의 점수 85, 90, 95를 value로 갖는 데이터를 추가한다.
3. 학생 추가: "David" 학생의 점수로 80을 추가한다.
4. 학생 점수 수정: "Alice" 학생의 점수를 88로 수정한다.
5. 학생 삭제: "Bob" 학생을 딕셔너리에서 삭제한다.
6. 1~5번까지 코드를 올려주세요.

7. 내장함수

내장함수

- 내장함수는 지금까지 봐왔던 print() 형태로 파이썬이 미리 지정해둔 함수들을 말함
- print()외 많은 내장함수가 존재하나 지금은 리스트, 튜플, 딕셔너리 등과 함께 사용하기 유용한 함수만 소개하고 넘어감
- 내장함수 내용은 앞으로 계속 추가하면서 학습하도록 하겠음

내장함수

codingOn

- sum(): 숫자 시퀀스의 모든 요소를 더해 합계를 반환
- 시퀀스 : 문자열, 리스트, 튜플, range() 등

```
numbers = [10, 20, 30, 40]
result = sum(numbers)
print(result)  # 100

scores = {"국어": 85, "영어": 90, "수학": 95}
total_score = sum(scores.values())
print(total_score)  # 270
```

내장함수

codingOn

- max() : 최댓값을 반환하는 함수
- min() : 최솟값을 반환하는 함수

```
numbers = [10, 20, 30, 40]
result = max(numbers)
print(result)  # 40
result = min(numbers)
print(result)  # 10

scores = {"국어": 85, "영어": 90, "수학": 95}
highest_score = max(scores.values())
print(highest_score)  # 95
lowest_score = min(scores.values())
print(lowest_score)  # 85
```

내장함수

codingOn

- len() : 길이를 반환하는 함수

```
numbers = [10, 20, 30, 40]
print(len(numbers))  # 4
scores = {"국어": 85, "영어": 90, "수학": 95}
print(len(scores))  # 3
```

- zip() : 여러 시퀀스를 병렬로 묶어주는 함수
- 각 시퀀스의 요소를 튜플로 묶어 반환하며, 길이가 가장 짧은 시퀀스에 맞춰 종료

```
names = ["Alice", "Bob", "Charlie"]
scores = [85, 90, 95]
zipped = list(zip(names, scores))
print(zipped)  # [('Alice', 85), ('Bob', 90), ('Charlie', 95)]
```

학습정리

- 리스트는 하나의 변수의 여러가지 값을 연속적으로 담을 수 있으며 순서가 중요하다.
- 리스트의 값은 추가, 수정, 삭제가 가능한 가변 객체이다.
- 이차원리스트는 리스트 안에 리스트를 넣는 형태로 행과 열로 데이터를 표현한다.
- 튜플은 리스트와 비슷하지만 추가, 수정, 삭제가 불가능한 불변 객체이다.

학습정리

- 셋(Set)은 중복을 허용하지 않는 수학의 집합과 같은 자료형이다.
- 딕셔너리는 키-값 형태로 요소를 갖는다.
- 딕셔너리내에서 키는 중복되어 사용할 수 없다.
- 셋(Set)과 딕셔너리는 가변객체이다.

다음 수업은?

- 이번수업은 앞으로 파이썬에 사용하는 다양한 자료형에 대해서 공부하였습니다.
- 다음시간에는 개발시 꼭 필요한 조건문과 반복문에 대해서 공부하겠습니다.

복.습.철.저

수고하셨습니다