04. 파이썬 제어문

저자는 매일 아침 회사에 출근합니다. 이때 스마트폰을 통해 날씨를 확인하고 비가 올 것 같으면 우산을 챙기고 그렇지 않으면 그냥 출근합니다. 여기서 저자가 매일 아침 회사로 출근하는 것은 반복적인 일에 해당하고, 날씨에 따라 우산을 챙기는 일은 조건을 보고 어떤 것을 선택하는 일에 해당합니다.

프로그램에도 이와 비슷한 것들이 존재하는데 어떤 조건을 보고 코드를 수행하는 것을 분기문이라고 하고, 어떤 조건 하에서 반복적으로 코드를 수행하는 것을 반복문이라고 부릅니다. 일반적으로 프로그램은 명령어가 기술된 순서대로 순차 수행되는데 분기문과 반복문은 순차 수행의 흐름을 변화시키기 때문에 제어문이라고도 부릅니다. 파이썬에는 if, for, while 이라는 제어문이 있으며, 이번 장에서는 이러한 제어문을 배워보겠습니다.

1) Boolean

2 장에서는 파이썬의 기본 자료형인 정수형, 실수형, 문자열을 배웠습니다. 이 세 가지 자료형이 파이썬에서 주로 사용되긴 하지만 Boolean(불리언)도 알게 모르게 자주 사용되는 기본 자료형입니다. 다만 Boolean 은 다른 자료형과 달리 True 또는 False 라는 값만 바인딩할 수 있습니다.

다음 코드에서 a 와 b 라는 변수는 각각 True 와 False 라는 값을 바인딩하고 있습니다. type() 내장 함수를 이용해 타입을 확인해보면 `<class 'bool'>`이 반환되는 것을 볼 수 있습니다. 짐작하시겠지만 여기서 bool 은 Boolean 의 줄임말입니다. 참고로 True 나 False 는 파이썬의 예약어로 true 나 false 가 아니라 첫 글자로 대문자를 사용해야 합니다.

```
>>> a = True
>>> type(a)
<class 'bool'>
>>> b = False
>>> type(b)
<class 'bool'>
>>>
```

그렇다면 Boolean 이라는 자료형은 언제 사용될까요? Boolean 이라는 자료형은 파이썬의 비교 연산자에서 많이 사용됩니다. 여기서 비교 연산자라는 것은 어떤 두 값을 비교하는 데 사용되는 연산자를 의미합니다. 예를 들어, 표 4.1 의 연산자가 바로 파이썬에서 사용되는 비교 연산자입니다.

표 4.1 파이썬 비교 연산자

연산자	연산자 의미
==	같다.
!=	다르다.
>	크다.
<	작다.
>=	크거나 같다.

연산자	연산자 의미
<=	작거나 같다.

파이썬의 비교 연산자는 비교 연산자를 기준으로 왼쪽과 오른쪽에 비교할 값이 존재합니다. 다음코드에서 첫 번째 문장은 정수형 값인 3 을 서로 비교하는 예입니다. 첫 번째 문장의 결과를 보면 True 가 반환된 것을 확인할 수 있는데, 비교 연산자를 기준으로 왼쪽에 있는 3 과 오른쪽에 있는 3 이 같은 값이기 때문입니다. 두 번째 문장을 보면 반환값이 False 인 것을 확인할 수 있는데, 이는 두 값이 같지 않다(!=)라고 했는데, 실제로는 같은 값이기 때문입니다. 세 번째는 왼쪽의 3 이 오른쪽의 3 보다 작다(<)라고 했는데, 실제로는 작지 않기 때문에 이번에도 반환값이 False 가 됩니다.

```
>>> 3 == 3
True
>>> 3 != 3
False
>>> 3 < 3
False
>>> 3 > 3
False
>>> 3 <= 3
True
>>> 3 >= 3
True
>>>
```

위 코드처럼 Boolean 자료형은 비교 연산자의 반환값으로 자주 사용됩니다. 표 4.1 의 비교 연산자는 정수형뿐만 아니라 실수형과 문자열 자료형에도 그대로 적용할 수 있습니다. 다음 코드를 보면 mystock 이라는 변수는 "Naver"라는 문자열을 바인딩하고 있습니다. 이 상태에서 mystock 이 바인딩하는 문자열이 "Naver"와 같은지(==) 확인해 보니 반환값으로 True 가 반환된 것을 확인할 수 있습니다.

```
>>> mystock = "Naver"
>>> mystock == "Naver"
True
>>>
```

사실 파이썬의 비교 연산자를 보면 너무나 당연해서 비교 연산자를 처음 배울 때는 큰 감흥이 없습니다. 그러나 비교 연산자는 앞으로 주식 프로그래밍을 할 때 자주 사용하게 될 연산자이니 꼭 기억하기 바랍니다. 예를 들어, 상한가였던 종목을 자동으로 매수하는 프로그램을 작성하는 경우를 생각해봅시다. 코스피와 코스닥에 있는 수많은 종목 중에서 어제 상한가였던 종목을 알아내려면 어떻게 해야 할까요? 일단 수많은 종목 중에서 찾는 것은 이번 장에서 배우게 될 반복문과 관련이 있으므로 그때 다시 다루기로 하고, 먼저 한 종목에 대해 두 개의 종가가 있을 때 이 종목이 상한가였는지 확인해 봅시다.

다음 코드에서 day1 은 어떤 주식 종목의 전일 종가이고 day2 는 당일 종가를 의미합니다. day2 의 값인 13,000 원은 day1 의 값인 10,000 원에서 30% 오른 가격이므로 상한가가 맞습니다(참고로 2015 년 6 월 15 일부터는 가격제한폭이 30%로 증가했습니다). 이러한 상한가를 확인하는 알고리즘을 파이썬 코드로 작성해보면 day2 에서 day1 을 뺀 금액이 day1 의 0.3(30%)에 해당하는 금액과 같은지(==) 확인하면 됩니다.

```
>>> day1 = 10000
>>> day2 = 13000
>>> (day2- day1) == (day1 * 0.3)
True
>>>
```

2) 논리 연산자

앞서 비교 연산자를 이용해 상한가를 확인하는 코드를 작성해봤습니다. 그러나 사실 한국 거래소(www.krx.co.kr)의 가격제한폭 제도를 살펴보면 정확한 상한가를 산출하는 방식은 기준 가격(전일 종가)에 단순히 0.3 을 곱하는 것이 아니며 다음과 같이 조금 복잡하게 계산됩니다.

1 차 계산: 기준 가격에 0.3 을 곱한다.

2 차 계산: 기준 가격의 호가 가격단위에 미만을 절사한다.

3 차 계산: 기준 가격에 2 차 계산에 의한 수치를 가감하되, 해당 가격의 호가 가격단위 미만을 절사한다.

예를 들어, 기준 가격이 9,980 원인 경우 다음과 같이 계산됩니다.

1 차 계산: 9,980 원 x 0.3 = 2,994 원

2 차 계산: 2,990 원(기준 가격(9,980 원)의 호가 가격단위인 10 원 미만 절사)

3 차 계산:

합산가격: 9,980 원 + 2,990 원 = 12,970 원

호가 가격단위 적용: 12,970 원의 호가 가격단위인 50 원 미만 절사(2 차 절사)

상한가: 12,950 원

위의 계산 과정을 살펴보면 호가 가격에 따라 두 번의 절사 과정을 거칩니다. KRX 한국 거래소에서 제공하는 호가 가격단위를 살펴보면 표 4.2 와 같습니다. 여기서 '호가 가격단위'란 가격대별로 호가할 수 있는 최소 단위를 말합니다. 참고로 '호가'는 거래소의 회원인 증권회사가 자기 명의로 시장에 매도 또는 매수 의사를 표시하는 것을 의미합니다. 잘 알고 계시다시피 증권사는 증권사의 고객으로부터 주문을 위탁받은 후 동 주문을 거래소에 호가하게 되는 것입니다.

표 4.2 호가 가격 단위

구분	단위
1,000원 미만	1원
1,000원 이상 5,000원 미만	5원

구분	단위
5,000원 이상 10,000원 미만	10원
10,000원 이상 50,000원 미만	50원
50,000원 이상 100,000원 미만	100원
100,000원 이상 500,000원 미만	500원
500,000원 이상	1,000원

문제는 "표 4.2 로부터 위에서 예시로 든 기준 가격(9,980 원)에 대한 호가를 파이썬을 통해 어떻게 구할 것인가?"입니다. 이럴 때 사용할 수 있는 연산자가 바로 논리 연산자인데, 파이썬에는 and, or, not 이라는 세 가지 종류의 논리 연산자가 있습니다. 논리 연산자 중 and 에는 '그리고'라는 뜻이 있고, 'or'에는 '또는'이라는 뜻이 있습니다. 마지막으로 'not'은 '~아닌'이라는 뜻이 있습니다.

간단한 코드를 통해 논리 연산자에 대해 좀 더 살펴보겠습니다. 다음 코드는 cur_price 가 바인딩하고 있는 기준 가격이 '5,000 원 이상 10,000 원 미만'의 호가 구간에 해당하는지 나타냅니다. cur_price 라는 변수가 현재 바인딩하고 있는 값인 9,980 원은 5,000 원보다는 큰 가격이고 10,000 원보다 낮은 가격이기 때문에 True 라는 불리언 값이 반환됩니다. and 연산자는 나열된 여러 조건이 모두 True 일 때만 연산의 결과로 True 를 반환합니다.

```
>>> cur_price = 9980
>>> cur_price >= 5000 and cur_price < 10000
True
>>>
```

앞서 당일 종가가 전일 종가보다 정확히 30% 상승했을 때 상한가로 판단했습니다. 그러나 앞에서 설명한 것처럼 실제로는 호가의 범위에 따라 두 번의 절사 과정을 거치기 때문에 실제로는 30%보다 낮은 가격일 수도 있습니다. 물론 호가를 정확히 계산하고 절사도 정확히 계산해서 실제 상한가를 계산할 수도 있지만 다음 코드와 같이 or 라는 논리 연산자를 사용해 대략 상한가와 비슷하게 오른 종목을 찾을 수도 있습니다. 다음 코드는 당일 종가(day2)에서 전일 종가(day1)를 뺀 금액이 전일 종가의 30%에 해당하는 금액이거나 29.2%에 해당하는 금액보다 큰 금액이면 상한가로 판단합니다. or 연산자는 나열된 여러 가지 조건 중 하나라도 True 이면 연산의 결과로 True 를 반환합니다.

```
>>> day1 = 10000
```

```
>>> day2 = 13000
>>> ((day2- day1) == (day1 * 0.3)) or ((day2-day1) > (day1 * 0.292))
True
>>>
```

3) 파이썬 if 문

이번 절에서는 어떤 조건에 따라 코드를 다르게 수행하는 조건문에 대해 배우겠습니다. 조건문이라고 하니 막상 뭔가 어려워 보이지만 아래와 같은 문장이 조건문의 예입니다. 주식 투자를 하다 보면 실제로도 아래와 같은 행동을 많이 합니다.

```
"위키북스의 현재가가 10,000 원 이상이면 10 주 매수"
```

```
"위키북스의 현재가가 9,000 원 이하이면 10 주 매도"
```

그렇다면 파이썬에서는 위와 같은 것들을 어떻게 표현할 수 있을까요? 먼저 위의 두 문장을 잘 살펴보면 공통으로 '~이면'이라는 단어가 있는 것을 볼 수 있습니다. 참고로 영어 단어 중 if 라는 단어의 뜻이 '만약 ~라면'입니다. 파이썬에서도 '만약 ~라면'이라는 뜻을 가진 if 라는 키워드로 조건문을 표현합니다. if 문을 이용해 앞에서 예로 든 매수 상황을 파이썬으로 표현하면 다음과 같습니다.

```
>>> wikibooks_cur_price = 11000
```

```
>>> if wikibooks_cur_price >= 10000:
        print("Buy 10")
```

```
Buy 10
```

```
>>>
```

위 코드를 살펴보면 몇 가지 새로운 파이썬 문법을 볼 수 있습니다.

```
1) 조건문에는 if 라는 키워드를 사용한다.
```

```
2) if 다음에는 '조건'이 존재하는데 이 조건이 참(True)이면 들여쓰기한 문장이 실행된다.
```

```
3) if 문의 끝에는 콜론(:)을 입력한다.
```

```
4) if 문의 조건이 참(True)일 때 실행되는 문장은 들여쓰기해야 한다.
```

위 코드에서 첫 번째 문장이 실행되면 wikibooks_cur_price 라는 변수는 11000 이라는 정숫값을 바인딩하게 됩니다. 두 번째 문장이 실행될 때 먼저 if 문의 조건에 해당하는 'wikibooks_cur_price >= 10000'에 대해 참인지 거짓인지가 판단됩니다. wikibooks_cur_price 변수가 현재 바인딩하고 있는 값은 11000 이고, 이 값은

10000 보다 크거나 같으므로 조건식은 참이 됩니다. if 문의 조건식이 참이므로 4 칸의 공백으로 들여쓰기된 'print("Buy 10")'이라는 문장이 실행됩니다. 그래서 화면에 'Buy 10'이 출력된 것입니다.

if 문을 이용해 위의 매도 상황을 파이썬 코드로 표현하면 다음과 같습니다. 조건문을 표현하기 위해 if 라는 키워드를 사용했으며, if 뒤에 조건식을 적었습니다. wikibooks_cur_price 변수는 현재 8,000 원이라는 값을 바인딩하고 있으므로 if 문의 조건식인 'wikibooks_cur_price < 9000'은 True 가 됩니다. 따라서 들여쓰기된 코드가 실행되고 화면에는 "Sell 10"이 출력됩니다.

```
>>> wikibooks_cur_price = 8000
>>> if wikibooks_cur_price < 9000:
        print("Sell 10")
```

```
Sell 10
>>>
```

앞의 코드에서는 if 문의 조건식이 참일 때 들여쓰기된 한 줄의 코드만 실행됐습니다. 그런데 "위키북스의 현재가가 10,000 원 이상이면 5 주씩 3 번을 연속해서 매수"하는 경우는 어떻게 프로그래밍할까요? 이럴 때는 다음 코드와 같이 조건이 참일 때 수행될 문장을 동일한 들여쓰기로 표현하면 됩니다. 참고로 C/C++ 또는 자바 같은 프로그래밍 언어에서는 동시에 수행될 코드 블록을 { } 기호로 감싸서 표현하는 반면 파이썬에서는 들여쓰기를 통해 코드 블록을 표현합니다.

```
>>> wikibooks_cur_price = 11000
>>> if wikibooks_cur_price >= 10000:
        print("Buy 5")
        print("Buy 5")
        print("Buy 5")
```

```
Buy 5
Buy 5
Buy 5
```

```
>>>
```

파이썬이 아닌 다른 프로그래밍 언어를 배우신 분이라면 이 부분이 처음에는 조금 어색할 수 있습니다. 그러나 시간이 지남에 따라 파이썬과 같이 들여쓰기를 통해 코드 블록을 표현하는 방식이 더 가독성 있는 코드를 작성하는 데 도움된다는 사실을 알 수 있을 것입니다. 참고로 파이썬에서 들여쓰기는 키보드의 스페이스 바를 눌러서 같은 개수의 공백을 넣거나 탭을 사용해 들여쓰면 되는데, 보통 네 칸의 공백을 더 많이 사용합니다.

3-1) if ~ else 문

이번에는 "위키북스의 현재가가 10,000 원 이상이면 10 주 매수, 그렇지 않으면 보유"와 같은 문장을 파이썬 코드로 작성해 봅시다. 우선 앞에서 배운 조건문과 다른 부분을 살펴보면 '그렇지 않으면'이라는 새로운 조건이 추가된 것을 확인할 수 있습니다. 파이썬에서는 '그렇지 않으면'이라는 조건을 else 라는 키워드를 사용해서 표현합니다.

앞의 문장을 if 와 else 라는 키워드를 사용해서 다음 코드와 같이 나타낼 수 있습니다. 이때 else 문은 if 문과 달리 뒤에 조건식이 위치하지 않습니다. 왜냐하면 else 문은 if 문의 조건식이 참이 아닐 때 수행되기 때문입니다. 다음 코드에서 wikibooks_cur_price 라는 변수는 11000 을 바인딩하고 있으므로 if 문의 조건식은 참이 됩니다. 따라서 if 문에 들여쓰기된 'print("Buy 10")'이 수행되고 화면에는 'Buy 10'이 출력됩니다.

```
>>> wikibooks_cur_price = 11000
>>> if wikibooks_cur_price >= 10000:
        print("Buy 10")
else:
        print("Holding")
```

```
Buy 10
>>>
```

이번에는 코드를 조금 수정해서 wikibooks_cur_price 가 바인딩하는 값을 9000 으로 변경한 후 if ~ else 문을 실행해 보겠습니다. 다음 코드에서는 if 문 이후의 조건식이 False 가 되므로 else 문에 들여쓰기된 코드인 'print("Holding")'이 실행되어 화면에 'Holding'이 출력됩니다.

```
>>> wikibooks_cur_price = 9000
>>> if wikibooks_cur_price >= 10000:
        print("Buy 10")
else:
        print("Holding")
```

```
Holding
>>>
```

if ~ else 문에서 한 가지 주의해야 할 점은 바로 들여쓰기입니다. 그림 4.1 의 코드를 보면 앞서 작성한 코드와 거의 비슷하지만 에러가 발생했습니다. 이는 else 문의 위치가 잘못됐기 때문입니다. 그림 4.1 에서 if 키워드 앞에는 공백이 없지만 else 키워드 앞에는 4 칸의 공백이 존재하는데, 이처럼 if 와 else 의 들여쓰기가 서로 다르면 파이썬에서는 문법 에러가 발생합니다.

```
>>> wikibooks_cur_price = 9000
>>> if wikibooks_cur_price >= 10000:
        print("Buy 10")
    else:
        print("Holding")

SyntaxError: unindent does not match any outer indentation level
>>>
```

그림 4.1 파이썬 들여쓰기 오류

파이썬 IDLE 를 처음 사용할 때는 오히려 다음과 같은 코드가 if 와 else 의 들여쓰기가 잘못된 것처럼 보입니다.

```
>>> if 3 > 1:
        print("3 이 크다")
else:
        print("3 이 작다")
```

```
3 이 크다
>>>
```

그러나 파이썬 IDLE 의 프롬프트(>>>)를 지워보면 실제로는 if 와 else 가 같은 위치에서 시작됨을 확인할 수 있습니다.

```
if 3 > 1:
        print("3 이 크다")
```

```
else:
        print("3 이  작다")
```

3-2) if ~ elif ~ else 문

if ~ else 문을 이용하면 if 문 뒤에 존재하는 조건식이 참이냐 거짓이냐에 따라 그림 4.2 와 같이 두 개의 코드 블록 중 하나를 실행할 수 있었습니다.

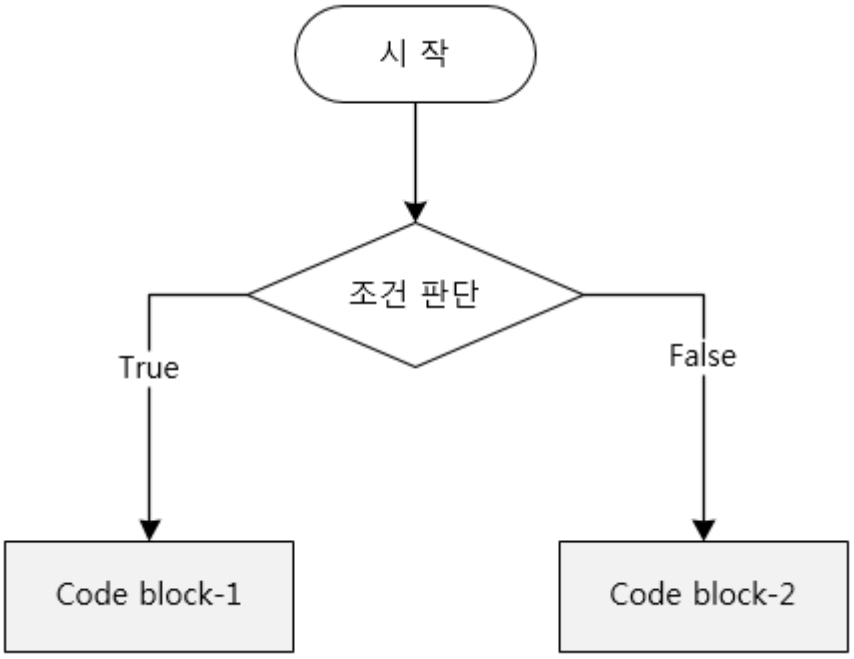

그림 4.2 if ~ else 문의 실행 구조

그런데 실행해야 할 경우의 수가 두 가지가 아니라 그보다 많은 경우에는 어떻게 표현해야 할까요? 예를 들어, 표 4.2 의 호가 가격단위를 보면 호가가 결정되는 경우의 수가 두 가지만 있는 것이 아니라 가격 범위가 7 개라는 것을 확인할 수 있습니다. 이럴 때 사용할 수 있는 추가 키워드가 바로 elif 입니다. elif 키워드는 '그렇지 않고 ~이면' 정도의 뜻을 가집니다. 표 4.2 를 한국어로 표현해보면 다음과 같이 표현할 수 있을 것입니다.

현재가가 1,000 원 미만이면 호가 가격은 1 원,

그렇지 않고 현재가가 1,000 원 이상이고 5,000 원 미만이면 호가 가격단위는 5 원,

그렇지 않고 현재가가 5,000 원 이상이고 10,000 원 미만이면 호가 가격단위는 10 원,

그렇지 않고 현재가가 10,000 원 이상이고 50,000 원 미만이면 호가 가격단위는 50 원,

그렇지 않고 현재가가 50,000 원 이상이고 100,000 원 미만이면 호가 가격단위는 100 원,

그렇지 않고 현재가가 100,000 원 이상이고 500,000 원 미만이면 호가 가격단위는 500 원,

그렇지 않고 현재가가 500,000 원 이상이면 호가 가격단위는 1,000 원

위의 문장에서 '~이면'을 if 로 '그렇지 않고'에 elif 라는 키워드를 사용하면 다음과 같이 파이썬 코드를 나타낼 수 있습니다. 물론 조건식이 있는 부분에는 파이썬의 비교 연산자와 논리 연산자를 사용했습니다.

```
>>> price = 7000
>>> if price < 1000:
        bid = 1
elif price >= 1000 and price < 5000:
        bid = 5
elif price >= 5000 and price < 10000:
        bid = 10
elif price >= 10000 and price < 50000:
        bid = 50
elif price >= 50000 and price < 100000:
        bid = 100
elif price >= 100000 and price < 500000:
        bid = 500
elif price >= 500000:
        bid = 1000

>>> bid
10
>>>
```

위 코드에서 price 라는 변수는 7000 을 바인딩하고 있으므로 호가 가격단위(bid)는 10 원이 됩니다. 파이썬 IDLE 에서도 bid 변수가 바인딩하고 있는 값을 확인해보면 10 이 반환되는 것을 확인할 수 있습니다.

4) for 문

이번 절에서는 파이썬의 제어문 중 하나인 for 반복문을 배우겠습니다. '반복'이라는 단어의 의미는 '같은 일을 되풀이함'입니다. 그렇다면 프로그래밍에서는 언제 반복문이 필요할까요?

화면에 1 부터 10 까지 숫자를 출력한다고 가정해 봅시다. 파이썬에 반복문이 없다면 다음 코드와같이 한번에 하나씩 출력해야 할 것입니다. 그러나 10 까지가 아니라 100 까지 출력해야 할 경우 이를 코드를 작성하려면 시간이 오래 걸리고 지루할 것입니다. 이럴 때 사용하는 것이 바로 반복문입니다.

```
>>> print(1)
1
>>> print(2)
2
>>> print(3)
3
>>> print(4)
4
>>> print(5)
5
>>> print(6)
6
>>> print(7)
7
>>> print(8)
8
>>> print(9)
```

```
9
>>> print(10)
10
>>>
```

for 문을 사용하면 1~10 까지를 출력하는 기능을 다음과 같이 구현할 수 있습니다.

```
>>> for i in [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]:
        print(i)
```

```
0
1
2
3
4
5
6
7
8
9
10
>>>
```

위 코드에서는 화면에 문자열을 출력하는 함수인 print 를 단 한 번만 사용했지만 화면에는 0 부터 10 까지의 값이 출력된 것을 확인할 수 있습니다. 이처럼 for 문을 사용하면 중복적인 부분의 코드를 줄일 수 있어 프로그램을 더욱 빨리 개발할 수 있고 코드의 가독성도 향상됩니다.

위 코드를 살펴보면 for 라는 새로운 파이썬 키워드를 볼 수 있습니다. 영어 사전에서 for 라는 단어의 의미를 찾아보면 정말로 많은 뜻이 나옵니다. 대표적으로 '왜냐하면', '동안', '위한', '대한', '로써는', '치고는', '에게', '찬성하는', '에도 불구하고'와 같은 뜻이 있습니다. 물론 이러한 뜻을 다 왜 울 필요는 없고 파이썬에서 for 는 '~대한' 정도의 의미를 가진 것으로 생각하면 됩니다.

in 이라는 영어 단어는 '~안에'라는 뜻이 있으므로 위 파이썬 코드를 영어 문장을 해석하듯이 해석해보면 "파이썬 리스트 안에 있는 각 i 에 대해 print(i)하시오." 정도가 됩니다. for 문을 작성할 때 주의할 점은 앞에서 배운 if 문과 마찬가지로 for 문의 끝에 콜론(:)이 있어야 하며, for 문에서 수행되는 문장은 들여쓰기 돼 있어야 한다는 점입니다.

이번에는 for 문을 사용하여 0 부터 4 까지의 숫자가 두 번씩 출력되는 프로그램을 작성해 봅시다. 힌트를 조금 드리자면 if 문에서도 조건이 참일 때 수행될 문장은 if 문의 시작 위치로부터 동일하게 들여쓰기했습니다.

다음 코드를 보면 for 문에서 들여쓰기된 두 개의 print 문을 볼 수 있습니다. 코드가 실행되면 먼저 리스트 내의 첫 번째 요소인 0 에 대해 print(0), print(0)이 수행됩니다. 그런 다음 리스트의 두 번째 요소인 1 에 대해 print(1), print(1)이 수행됩니다. 이렇게 해서 리스트의 마지막 요소인 4 에 대해서까지 print 문이 수행됩니다.

```
>>> for i in [0, 1, 2, 3, 4]:
        print(i)
        print(i)
```

```
0
0
1
1
2
2
3
3
4
```

```
4
```

```
>>>
```

위 코드에서 for 라는 키워드 뒤에 위치하는 i 는 변수로서 in 다음에 있는 리스트의 요소를 하나씩 바인딩합니다. 처음에는 0 을 바인딩하고 그 다음에는 1, 2, 3, 4 순으로 하나씩 바인딩합니다. i 는 for 나 in 과 같은 키워드가 아니라 변수이기 때문에 변수 이름 규칙만 만족한다면 마음대로 지정할 수 있습니다.

4-1) for 와 range

앞서 for 라는 파이썬 키워드를 사용해 단 두 줄의 코드로 화면에 0 부터 10 까지를 출력해 봤습니다. 그러나 한 가지 불편한 점은 출력하고자 하는 숫자를 리스트를 통해 명시적으로 저장하고 있어야 한다는 점입니다. 물론 앞의 예시처럼 반복하는 데이터가 많지 않은 경우에는 리스트를 사용해 명시적으로 데이터를 나열할 수 있습니다. 그런데 0 부터 10 까지가 아니라 0 부터 100 까지 출력해야 할 때는 어떻게 해야 할까요?

이럴 때 사용하는 것이 바로 range 입니다. range 는 '범위'라는 뜻을 가진 영어 단어인데, 파이썬에서 range 를 이용하면 간단히 정수 범위를 표현할 수 있습니다. 예를 들어, range(1, 10)은 1 부터 9 까지의 숫자 범위를 나타냅니다.

```
>>> range(1,10)
range(1, 10)
>>>
```

파이썬 range 에서 중요한 점은 시작 숫자와 끝 숫자를 지정했을 때 끝 숫자는 범위에 포함되지 않는다는 점입니다. 이를 확인하기 위해 range(1, 10)을 list()를 통해 리스트 객체로 변환해보면 리스트에 정말로 1 부터 9 까지만 저장돼 있음을 확인할 수 있습니다.

```
>>> list(range(1,10))
[1, 2, 3, 4, 5, 6, 7, 8, 9]
>>>
```

range 를 이용하면 큰 수의 범위에 대해서도 for 문을 쉽게 작성할 수 있습니다. 먼저 0 부터 10 까지의 숫자를 출력하는 코드를 for 와 range 를 사용해 다시 작성해 보겠습니다. 다음 코드를 보면 in 다음에 리스트를 사용하는 경우보다 코드가 매우 간단하다는 것을 확인할 수 있습니다. 0 부터 10 까지 출력하기 위해 range(0, 10)이 아니라 range(0, 11)이라는 표현이 사용된 것도 눈여겨보기 바랍니다.

```
>>> for i in range(0, 11):
        print(i)

0
1
```

```
2
3
4
5
6
7
8
9
10
>>>
```

독자 중에서는 다음 코드와 같이 작성한 분도 있을 것입니다. range(0, 11)을 list 라는 키워드를 이용해 리스트 객체로 변환한 후 해당 리스트 객체에 대해 for 문을 수행했습니다. range(0, 11)을 사용한 것과 list(range(0, 11))을 사용한 것의 실행 결과는 동일합니다. 다만 list(range(0, 11))과 같이 명시적으로 리스트 객체로 변환하기보다는 range(0, 11)을 그대로 사용하는 편이 메모리 사용 관점에서 더 효율적입니다.

```
>>> for i in list(range(0, 11)):
        print(i)
```

```
0
1
2
3
4
5
6
```

```
7
8
9
10
>>>
```

range 객체를 그대로 사용하면 for 문에서 필요할 때마다 값을 i 라는 변수가 바인딩해서 사용하게 되지만 리스트 객체로 변환하면 모든 범위의 값을 한 번에 리스트로 만들어서 저장하기 때문에 더 큰 메모리가 필요해집니다. 메모리 절약뿐만 아니라 코드 가독성 측면에서도 range 객체를 그대로 사용하는 편이 더 간결해 보입니다.

4-2) for 와 리스트

앞 절에서는 파이썬 리스트에 숫자를 저장한 후 파이썬 for 문에서 해당 리스트를 이용해 반복적으로 어떤 작업을 수행하는 코드를 작성해 봤습니다. 앞에서 배운 것처럼 파이썬 리스트에는 정수형만 입력할 수 있는 것이 아니라 문자열도 입력할 수 있습니다. 따라서 파이썬의 for 문과 리스트 자료구조를 적절히 사용하면 리스트 내에 있는 원소에 대해 반복적인 작업을 쉽게 수행할 수 있습니다.

예를 들어, 관심 있게 보고 있는 종목들의 종목명을 리스트로 저장하고 있다가 해당 종목에 대해 각각 10 주씩 매수하는 프로그램을 작성한다고 가정해 봅시다. 다음 코드를 보면 interest_stocks 리스트에는 'Naver', 'Samsung', 'SK Hynix'와 같은 종목명이 있습니다.

```
>>> interest_stocks = ["Naver", "Samsung", "SK Hynix"]
```

관심 종목 리스트에 있는 각 원소에 대해 10 주씩 매수하기 위해 for 문을 사용했습니다. for 키워드 다음에 나오는 company 는 변수 이름으로서 interest_stocks 리스트에 있는 원소를 하나씩 바인딩하는 역할을 합니다. for 문에서 리스트의 각 원소에 대해 수행될 문장은 print 문인데 이때 %s 라는 표현이 나옵니다. print 함수에서는 문자열이 출력될 위치에 %s 로 나타내고 실제로 출력될 문자열은 % 기호 뒤에 변수명을 쓰면 됩니다.

```
>>> for company in interest_stocks:
        print("%s: Buy 10" % company)
```

```
Naver: Buy 10
Samsung: Buy 10
SK Hynix: Buy 10
>>>
```

print 함수의 사용법은 C 언어나 자바와 비슷하므로 이전에 프로그래밍해본 경험이 있는 분들은 쉽게 이해하실 것입니다. 그러나 프로그래밍 경험이 없는 분들은 처음에는 매우 어색하고 복잡하게 느껴지실 겁니다. 만일 다음과 같이 'print("company Buy 10")'이라고 작성하면 리스트 안에 있는 관심 종목명이 출력되는 것이 아니라 'company Buy 10'이라는 문자열만 화면에 출력됩니다.

```
>>> interest_stocks = ["Naver", "Samsung", "SK Hynix"]
>>> for company in interest_stocks:
```

```
        print("company: Buy 10")
```

```
company: Buy 10
company: Buy 10
company: Buy 10
>>>
```

4-3) for 와 튜플

파이썬의 튜플은 리스트와 유사하게 데이터를 순서대로 저장할 수 있는 자료구조입니다. 다만 리스트와 달리 튜플은 한번 만들어지면 수정할 수 없습니다. for 문에서 리스트를 사용하는 것과 비슷하게 튜플도 사용할 수 있습니다. 다음 코드는 앞서 작성한 코드에서 리스트 대신 튜플을 사용한 것입니다.

```
>>> interest_stocks = ("Naver", "Samsung", "SK Hynix")
>>> for company in interest_stocks:
        print("%s: Buy 10" % company)
```

```
Naver: Buy 10
Samsung: Buy 10
SK Hynix: Buy 10
>>>
```

앞에서 이야기한 것처럼 튜플은 수정할 수 없기 때문에 리스트보다 빠릅니다. 따라서 관심 종목을 변경할 필요가 없는 경우에는 이를 튜플로 관리하고 파이썬의 for 문을 사용하는 것이 좋고, 관심 종목을 변경할 필요가 있는 경우에는 데이터를 리스트로 관리하는 것이 좋습니다.

4-4) for 와 딕셔너리

3 장에서 배운 자료구조 중 리스트와 튜플은 데이터를 순서대로 저장하는 자료구조입니다. 이와 달리 딕셔너리는 키-값 쌍을 저장하는 구조로서 데이터를 입력한 순서대로 저장하지 않습니다. 따라서 딕셔너리에 대해 for 문을 사용하려면 조금 특별한 방법이 필요합니다. 그렇다면 어떨 때 for 문과 딕셔너리를 함께 사용하면 좋을까요?

앞의 예시에서는 관심 종목에 대해 동일하게 10 주씩 주식을 매수했습니다. 그런데 이번에는 종목마다 다른 수량만큼 주식을 매수하고자 합니다. 프로그램으로 주식을 매수하기 전에 종목별로 얼마만큼 매수할 것인지를 미리 표현해야 한다면 딕셔너리를 통해 다음과 같이 쉽게 데이터를 표현할 수 있습니다. 여전히 관심 종목은 'Naver', 'Samsung', 'SK Hynix'이지만 매수 수량은 종목마다 다른 것을 확인할 수 있습니다.

```
>>> interest_stocks = {"Naver":10, "Samsung":5, "SK Hynix":30}
>>>
```

interest_stocks 라는 딕셔너리에 있는 키-값 쌍에 대해 매수 동작을 반복적으로 수행하려면 for 문을 사용하면 됩니다. 다만 딕셔너리의 한 원소에는 키와 값이 있기 때문에 for 문 다음에 있는 변수가 한 개가 아니라 두 개를 적어야 합니다. 그리고 딕셔너리 이름을 적은 다음 .items( )를 붙여야 합니다.

```
>>> for company, stock_num in interest_stocks.items():
        print("%s: Buy %s" % (company, stock_num))
```

```
SK Hynix: Buy 30
Naver: Buy 10
Samsung: Buy 5
>>>
```

위 코드에서 화면 출력을 수행하는 코드를 보면 company 와 stock_num 이라는 두 변수가 바인딩하는 값을 출력하기 위해 두 개의 %s 를 사용했고 % 기호 다음에 실제 변수명을 지정할 때는 (company, stock_num)과 같이 튜플로 표현한 것을 볼 수 있습니다.

3 장에서는 딕셔너리의 키-값 쌍만을 구하기 위해 keys()와 values()이라는 메서드를 사용했습니다. 딕셔너리의 keys() 메서드를 이용하면 다음과 같이 코드를 작성할 수도 있습니다. 앞에서 설명한 것처럼 interest_stock.keys()의 반환값은 dict_keys(['SK Hynix', 'Naver', 'Samsung'])인데, 이것은 리스트는 아니지만 이를 명시적으로 리스트로 변환할 필요는 없습니다. 파이썬의 for 문은 dict_keys 타입에 대해서도 제대로 동작하기 때문이죠.

```
>>> interest_stocks = {"Naver":10, "Samsung":5, "SK Hynix":30}
>>> for company in interest_stocks.keys():
        print("%s: Buy %s" % (company, interest_stocks[company]))

SK Hynix: Buy 30
Naver: Buy 10
Samsung: Buy 5
>>>
```

5) while 문

4.4 절에서는 반복적인 작업을 파이썬의 for 문을 통해 처리했습니다. 파이썬에는 for 문 말고도 동일하게 반복문을 작성할 수 있는 while 문이 있습니다. for 문으로도 충분히 반복문을 작성할 수 있는데 왜 while 문이라는 것도 있을까요? 일반적으로 파이썬의 for 문은 반복 횟수가 미리 정해져 있거나 리스트, 튜플, 사전과 같은 파이썬 자료구조와 함께 사용됩니다.

반면 while 문은 반복해야 할 횟수가 특별히 정해지지 않고 어떤 조건을 충족하는 동안만 실행될 때 주로 사용합니다. 당연한 말이지만 for 문을 사용하나 while 문을 사용하나 결과는 동일합니다. 다만 반복을 수행하는 구조를 보고 for 문을 사용하는 것이 프로그래밍하기 더 쉬워 보이면 for 문으로 작성하는 것이고, while 문이 더 쉬워 보이면 while 문으로 작성하면 됩니다.

먼저 for 문에서 했던 것처럼 0 부터 10 까지의 숫자를 while 문을 사용해 화면에 출력해 보겠습니다. 다음 코드를 보면 먼저 i 라는 변수가 초깃값으로 0 을 바인딩합니다. while 문에는 조건문(i <= 10)이 있는데, 현재 i 가 바인딩하는 값이 10 보다 작거나 같으므로 해당 조건을 만족합니다. 조건문이 참이므로 while 문에 속하는 문장을 순서대로 실행합니다. 즉, print(i)라는 문장이 먼저 실행되고 현재 i 라는 변수가 바인딩하고 있는 값인 0 이 화면에 출력됩니다. 그리고 'i = i+1'이라는 문장이 수행되어 i 가 바인딩하는 값이 0 에서 1 로 바뀝니다.

```
>>> i = 0
>>> while i <= 10:
        print(i)
        i = i+1
```

```
0
1
2
3
4
5
6
```

```
7
8
9
10
>>>
```

while 문에 속하는 모든 문장이 수행된 경우 다시 while 문의 조건문으로 이동한 후 조건식을 다시 판단합니다. 이번에는 i 라는 변수가 1 을 바인딩하고 있지만(앞에서 1 증가했으므로) 여전히 10 보다는 작거나 같은 값입니다. 즉, 여전히 조건식은 참이 됩니다. 따라서 while 문에 속하는 문장을 차례로 수행합니다. 이러한 방식으로 i 라는 변수가 바인딩하는 값이 10 일 때까지 while 문 내부의 문장을 순서대로 수행하고 i 가 11 이 될 때 더는 조건식을 만족하지 않아 while 문을 빠져나오게 됩니다.

위 코드에서 i 라는 변수는 초깃값으로 0 을 바인딩하고 있었습니다. 그러나 while 문에서 조건문을 만족할 때마다 값이 1 씩 증가했습니다. 정말로 i 값이 증가했는지 확인해 봅시다. 다음 코드와 같이 파이썬 IDLE 에서 i 값을 확인해보면 i 값이 11 로 증가한 것을 확인할 수 있습니다. 앞서 설명한 것처럼 i 값이 11 이 되면 더는 while 문의 조건식(i <= 10)을 만족하지 않기 때문에 while 문을 빠져나오게 된 것이죠.

```
>>> i
11
>>>
```

5-1) while 문을 이용한 상한가 계산

while 문에 좀 더 익숙해지기 위해 주식과 관련된 간단한 코드를 작성해보겠습니다. '위키북스'의 주식 1 주를 보유하고 있고, 주가가 10,000 원이었는데 5 일 연속으로 상한가를 기록했다고 가정해봅시다. 첫 번째 상한가 후의 가격은 다음과 같이 계산할 수 있을 것입니다.

```
>>> 10000 + 10000 * 0.3
13000.0
>>>
```

보다시피 10,000 원에서 30% 오른 가격을 더해줌으로써 상한가를 계산한 것입니다. 계산 결과는 13,000 원입니다. 이 금액에서 두 번째 상한가가 발생하면 다시 30%가 더 오르기 때문에 다음과 같이 계산할 수 있을 것입니다.

```
>>> 13000 + 13000 * 0.3
16900.0
>>>
```

두 번의 상한가를 기록하게 되면 10,000 원에서 16,900 원이 됩니다. 문제는 두 번의 상한가를 계산하는 것도 점점 복잡해지고 있다는 것입니다. 아직도 남아있는 세 번의 상한가 계산을 하자니 머리가 아파집니다.

위와 같은 상황에서 while 문을 이용하면 다섯 번의 상한가 후의 가격도 쉽게 계산할 수 있습니다. 각자 while 문을 이용해 어떻게 프로그래밍할 수 있을지 간단히 생각해본 후 다음 코드를 보시기 바랍니다.

```
>>> wikibooks = 10000
>>> day = 1
>>> while day < 6:
        wikibooks = wikibooks + wikibooks * 0.3
        day = day + 1

>>> wikibooks
```

```
37129.3
>>>
```

어떤가요? 여러분이 생각한 것과 비슷하게 코드가 작성됐나요? 저자는 먼저 wikibooks 라는 변수에 첫 번째 상한가가 일어나기 전날의 종가인 10,000 원을 바인딩해 뒀습니다. 그리고 5 번의 상한가를 세기 위해 날짜를 저장하는 변수인 day 에 초깃값으로 1 을 바인딩했습니다.

현재 day 는 1 을 바인딩하고 있으므로 'day < 6'의 조건을 만족합니다. 조건문을 만족하기 때문에 while 문 안쪽에 들여쓰기된 코드가 순서대로 수행됩니다. 따라서 먼저 wikibooks 가 바인딩하는 값에 0.3 을 곱한 값이 계산되고 이 값이 현재 wikibooks 가 바인딩하고 있는 값에 더해집니다. 이렇게 최종 계산된 값을 다시 wikibooks 라는 변수가 바인딩하게 됩니다. 따라서 wikibooks 는 13,000 원을 바인딩하게 됩니다. 그런 후 day 라는 변수의 값이 1 증가해서 2 가 됩니다.

이러한 동작은 day 값이 5 일 때까지 반복되므로 총 5 번의 상한가가 발생한 금액을 wikibooks 라는 변수가 바인딩하게 됩니다. 위 코드를 보면 10,000 원에서 5 일 연속 상한가가 발생하면 약 37,000 원 정도가 되는 것을 확인할 수 있습니다.

while 문의 동작 원리를 정리하면 다음과 같습니다.

1) **while** 문의 조건 확인

2) 조건을 만족하면 **while** 문 내부의 코드를 차례로 수행하고, 조건을 만족하지 않으면 **while** 문의 다음 문장을 수행

3) 2)의 과정에서 **while** 문 내부의 코드를 차례로 수행한 경우 다시 **while** 문의 조건 확인으로 이동

5-2) while 과 if

이번에는 파이썬의 반복문 중 하나인 while 문과 분기문인 if 문을 함께 사용하는 것에 대해 알아보겠습니다. 이러한 코드는 반복적인 작업을 수행하다가 특정 조건일 때는 다르게 처리하고자 할 때 많이 사용합니다. 예를 들어, 앞에서는 while 문을 이용해 0 부터 10 까지의 숫자를 화면에 출력해봤습니다. while 문과 If 문을 함께 사용하면 0 부터 10 까지의 숫자 중 홀수만 출력하는 프로그램을 구현할 수 있습니다.

먼저 0 부터 10 까지의 숫자에 대해 반복해야 하므로 while 문을 사용하겠습니다. while 문에서 조건을 만족하는 경우 while 문 내에서 들여쓰기된 문장을 순서대로 수행하게 되는데, 이때 홀수인 경우에만 화면에 숫자를 출력하면 됩니다. 그런데 어떻게 숫자가 홀수인지 또는 짝수인지 확인할 수 있을까요?

프로그래밍할 때 어떤 숫자가 홀수인지 짝수인지 확인하는 일반적인 방법은 숫자를 2 로 나눈 후 나머지를 보는 것입니다. 예를 들어, 1 을 2 로 나눈 나머지는 1 이 되므로 홀수입니다. 2 는 2 로 나누면 나머지는 0 이 되므로 짝수입니다. 3 을 2 로 나누면 나머지는 1 이 되므로 홀수입니다.

다음 코드는 while 문과 if 문을 사용하여 0 부터 10 까지의 숫자 중에서 홀수만 화면에 출력하는 프로그램을 구현한 것입니다. 전반적으로 저자가 설명한 방식으로 구현돼 있으며, 한 가지 특이한 점은 num 이라는 변수가 바인딩하고 있는 값을 1 만큼 증가시킬 때 'num = num + 1' 대신 'num += 1'이라고 표현했다는 것입니다. 사실 두 표현은 의미가 같은데, 'num += 1'과 같은 표현도 많이 사용되기 때문에 눈으로 익혀두고 자주자주 사용해 보기 바랍니다.

```
>>> num = 0
>>> while num <= 10:
        if num % 2 == 1:
            print(num)
        num += 1
```

```
1
3
5
7
9
```

```
>>>
```

5-3) break 와 continue

for 나 while 문을 이용해 어떤 작업을 반복적으로 수행하다가 특정 조건일 때 반복문 자체를 빠져나와야 할 때가 있습니다. 예를 들어, 주식 프로그램을 만들었다면 해당 프로그램은 온종일 켜져 있다가 증시 거래 시간이 지나면 동작을 멈춰야 하는 것들이 이 경우에 해당합니다.

먼저 프로그램을 계속해서 실행하게 하려면 앞에서 배운 while 문을 이용해 계속해서 반복하게 해야 합니다. while 문에서 계속 반복하게 하려면 while 문의 조건이 항상 참이 되도록 만들면 됩니다. 그러면 while 문 내의 문장을 실행하고 다시 조건을 판단할 때 항상 참이 되므로 while 문 내의 코드가 무한히 실행되는 것입니다.

이처럼 무한히 실행되는 반복문을 무한 루프(infinite loop)라고 하며, 간단하게는 다음 코드와 같이 작성할 수 있습니다. 이 코드를 실행하면 파이썬 IDLE 에 계속해서 'Find stocks'가 출력됩니다. 'Find stocks'가 무한히 출력되도록 프로그래밍했기 때문에 무한히 출력되는 것입니다. 프로그램을 강제로 종료하려면 파이썬 IDLE 에서 Ctrl + C 키를 눌러 중단해야 합니다.

```
>>> while 1:
        print("Find stocks")
```

파이썬의 반복문(while 문 또는 for 문)에서 빠져나오려면 break 이라는 키워드를 사용하면 됩니다. 위 코드에서 무한 루프는 원래 무한히 반복해야 하는데, 여기에 break 이라는 키워드를 넣어주면 break 를 만나는 순간 해당 위치에서 반복문을 빠져나오게 됩니다.

다음 코드를 실행해 보면 먼저 while 문의 조건식을 판단하게 되고 조건식에 1 이라는 숫자만 있으므로 참으로 판단됩니다. 참고로 거짓이 되려면 0 이라는 값이 있어야 하고 0 이 아닌 다른 숫자가 있으면 모두 참으로 판단합니다.

while 문의 조건이 참이므로 while 문 내의 코드가 실행되겠죠? 따라서 먼저 print("Find stocks")라는 문장이 실행되고, 그 결과 화면에 'Find stocks'라는 문자열이 출력됩니다. 그다음 break 문이 실행되는데 앞서 설명한 것처럼 break 라는 키워드는 루프를 빠져나가게 하므로 더는 코드가 실행되지 않고 프로그램은 종료됩니다.

```
>>> while 1:
        print(“Find stocks”)
        break
```

```
Find stocks
```

```
>>>
```

while 문과 break 를 이용해 간단한 프로그램을 하나 더 작성해보겠습니다. 이번에도 화면에 0 부터 10 까지의 숫자를 출력하는데, 이번에는 while 문에서 무한 루프 조건('while 1:')을 사용해보기 바랍니다. 다음 코드를 보면 while 문 내에서 if 문을 이용해 현재 숫자가 10 이면 break 를 통해 while 문 바깥으로 나가도록 프로그래밍된 것을 볼 수 있습니다.

```
>>> num = 0
>>> while 1:
        print(num)
        if num == 10:
            break
        num += 1
```

```
0
1
2
3
4
5
6
7
8
9
10
>>>
```

while 문이나 for 문을 사용하는 반복문에서 break 를 사용하면 반복문 전체를 빠져나오게 됩니다. 그런데 반복문 전체를 빠져나오는 것이 아니라 해당 조건만 건너뛰고 싶을 때는 어떻게 하면 될까요? 예를 들어, 화면에 1 부터 10 까지 출력하는데 5 인 경우에만 출력하지 않고 건너뛰는 것처럼 말이죠. 이럴 때 사용할 수 있는 키워드가 바로 continue 입니다.

다음 코드를 참조하면 while 문 내에서 먼저 num 값을 증가하게 했습니다. 그리고 증가한 num 값이 5 와 같으면 continue 가 실행되도록 프로그래밍돼 있습니다. 파이썬 프로그램이 실행되다가 continue 를 만나면 그 아래의 코드를 수행하지 않고 while 문의 조건을 판단하는 곳으로 점프하게 됩니다. 따라서 다음 코드를 실행하면 num 값이 5 일 때는 화면에 5 가 출력되지 않은 채로 다시 while 문의 조건식으로 이동하게 되고 그다음 'num += 1'에 의해 num 값이 5 에서 6 으로 증가하게 됩니다.

```
>>> num = 0
>>> while num < 10:
        num += 1
        if num == 5:
            continue
        print(num)

1
2
3
4
6
7
8
9
10
>>>
```

물론 1 부터 10 까지의 숫자를 출력하는데 5 만 출력하지 않아야 한다고 할 때 위 코드와 같이 continue 문을 사용해야만 하는 것은 아닙니다. 단순히 if 문을 통해 num 값이 5 가 아닐 때만 출력하도록 하는 것이 구현하기 더 쉽습니다. 다만 복잡한 프로그램을 작성하다 보면 반복문의 특정 조건에서만 코드를 실행하지 않고 다음으로 이동하고 싶을 때가 있는데 이럴 때 continue 를 사용하면 된다는 정도만 기억하면 됩니다.

6) 중첩 루프

이번 절에서는 중첩 루프에 대해 배워 보겠습니다. 여기서 '루프'라는 용어는 반복을 의미하고 '중첩'이라는 것은 여러 개가 겹치는 것을 의미입니다. 즉, 반복문 여러 개가 겹쳐 있는 구조를 중첩 루프라고 합니다. 보통 두 개의 반복문이 겹쳐 있는 '이중 루프'와 세 개의 반복문이 겹쳐 있는 '삼중 루프'를 가장 많이 사용합니다.

다음은 반복문 두 개가 겹쳐 있는 '이중 루프'의 예입니다. 반복문은 for 키워드를 사용했고 for 문 내부에서 조건을 만족할 때 수행되는 문장에는 pass 키워드를 사용했습니다. 참고로 파이썬의 pass 키워드는 아무것도 수행하지 않음을 의미합니다.

```
>>> for i in [1, 2, 3, 4]:
        for j in [1, 2, 3, 4]:
            pass

>>>
```

파이썬 반복문에는 실행될 문장이 최소한 하나라도 있어야 문법 오류가 나지 않으므로 pass 를 사용해 문법 오류가 발생하는 것을 방지한 것입니다. 물론 pass 대신 print("")와 같은 구문을 넣어도 프로그램의 동작을 간단히 테스트해볼 수도 있습니다.

그렇다면 어떤 경우에 두 개의 반복문을 겹쳐서 사용하는 것일까요? 그림 4.3 은 저자가 거주하는 아파트의 각 세대를 간단히 표시해 본 것입니다. 1 층에는 101 호, 102 호, 103 호, 104 호가 있고 2 층에는 201 호, 202 호, 203 호, 204 호가 있습니다. 지금까지 배운 반복문으로는 101 호, 102 호, 103 호, 104 에 대해 반복적인 일을 수행할 수 있습니다. 그런데 그림 4.3 은 1 층에만 세대가 있는 것이 아니라 2 층, 3 층, 4 층에도 각 세대가 있습니다. 우리가 자주 사용하는 엑셀도 데이터가 그림 4.3 과 같은 형태로 저장되는데 이러한 구조를 2 차원 데이터라고 표현합니다. 이러한 2 차원 데이터를 다룰 때 필요한 것이 바로 이중 루프입니다.

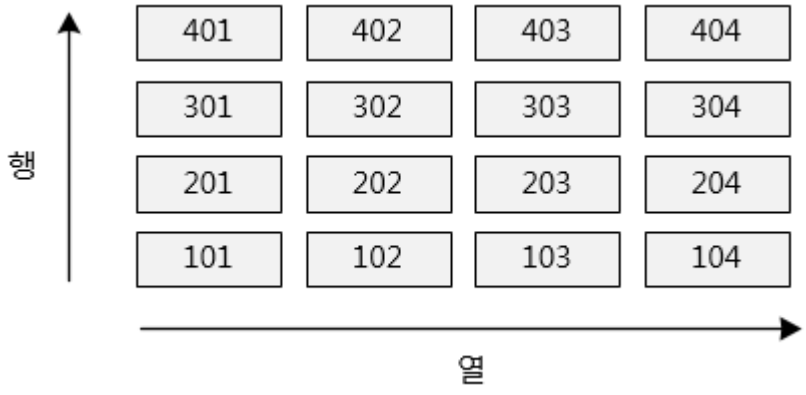

그림 4.3 2 차원 데이터의 표현

그림 4.3 의 각 세대에 대해 신문을 자동으로 배달하는 로봇을 만든다고 가정해 봅시다. 지금까지 배운 반복문을 사용한다면 다음과 같이 로봇에 프로그래밍해야 할 것입니다.

```
1 층에 가서 1 층의 각 세대에 신문 배달
2 층에 가서 2 층의 각 세대에 신문 배달
3 층에 가서 3 층의 각 세대에 신문 배달
4 층에 가서 4 층의 각 세대에 신문 배달
```

1 층의 각 세대에 신문을 배달하는 것을 파이썬의 반복문을 사용하면 다음과 같이 표현할 수 있습니다. 이 코드를 복사해서 리스트의 값만 바꿔주면 2 층, 3 층, 4 층에도 신문을 배달할 수 있습니다. 그러나 이 방식은 각 층에 있는 세대에 대해서는 반복문을 사용했지만 층별로는 여전히 반복문을 사용하지 못했기 때문에 코드가 중복되고 길어진다는 단점이 있습니다.

```
>>> for i in [101, 102, 103, 104]:
        print("Newspaper delivery: ", i)
```

```
Newspaper delivery:  101
Newspaper delivery:  102
Newspaper delivery:  103
Newspaper delivery:  104
>>>
```

이번에는 이중 루프를 이용해 코드를 작성해 보겠습니다. 이를 위해서는 먼저 그림 4.3 과 같은 2 차원 데이터를 파이썬의 2 차원 리스트로 표현해야 합니다. 3 장에서 배운 1 차원 리스트는 [ ] 기호 사이에 데이터가 쉼표로 구분되어 저장되는 형태였습니다. 2 차원 리스트는 리스트 안에 정수형, 문자열, 실수형과 같은 데이터가 존재하는 것이 아니라 리스트 타입이 존재하는 구조를 의미합니다. 예를 들어, 그림 4.3 은 다음과 같이 2 차원 리스트로 표현할 수 있습니다.

```
>>> apart = [[101, 102, 103, 104],[201, 202, 203, 204],[301, 302, 303, 304], [401, 402, 403,
404]]
>>>
```

2 차원 리스트도 1 차원 리스트와 마찬가지로 인덱싱할 수 있습니다. apart[0]으로 apart 리스트의 첫 번째 원소에 접근할 수 있는데, 리스트의 첫 번째 원소 역시 리스트입니다. 다음 코드를 보면 2 차원 리스트에 대한 인덱싱을 통해 그림 4.3 의 각 층의 세대 정보가 저장된 리스트에 접근할 수 있습니다. apart[0]에 위치한 값이 정말로 리스트 타입인지 확인하기 위해 type() 함수를 사용해보면 정말로 리스트 타입인 것을 확인할 수 있습니다.

```
>>> apart[0]
[101, 102, 103, 104]
>>> apart[1]
[201, 202, 203, 204]
>>> apart[2]
[301, 302, 302, 303]
>>> apart[3]
[401, 402, 403, 404]
>>> type(apart[0])
<class 'list'>
>>>
```

2 차원 리스트의 경우 인덱싱을 한번 하면 리스트 안쪽의 리스트에 접근할 수 있습니다. 따라서 한 번 더 인덱싱하면 접근 한 리스트 안의 데이터에 접근할 수 있게 됩니다. 예를 들어, 다음 코드를 보면 apart[0]으로 인덱싱했을 때 1 층의 세대 정보가 담긴 리스트에 접근했고 연속해서 인덱싱함으로써 각 세대의 정보를 구할 수 있습니다.

```
>>> apart[0][0]
101
>>> apart[0][1]
102
>>>
```

2 차원 구조의 데이터에 대해 2 차원 리스트로 표현했으니 이중 루프를 이용해 신문 배달을 해보겠습니다. 다음 코드를 보면 먼저 바깥쪽 for 문을 통해 각 층에 대한 리스트를 구합니다. 바깥쪽 for 문이 동작하는 동안 floor 변수는 [101, 102, 103, 104], [201, 202, 203, 204], [301, 302, 303, 304], [401, 402, 403, 404]라는 리스트를 순서대로 바인딩하게 됩니다. 이를 그림으로 나타내면 그림 4.4 와 같습니다. 그림 4.4 는 바깥쪽 for 문이 첫 번째로 실행될 때 floor 변수가 2 차원 리스트의 첫 번째 원소인 [101, 102, 103, 104] 리스트를 바인딩하는 것을 나타냅니다.

```
>>> for floor in apart:
        for room in floor:
            print("Newspaper derivery: ", room)
```

```
Newspaper derivery:  101
Newspaper derivery:  102
Newspaper derivery:  103
Newspaper derivery:  104
Newspaper derivery:  201
Newspaper derivery:  202
Newspaper derivery:  203
Newspaper derivery:  204
Newspaper derivery:  301
Newspaper derivery:  302
Newspaper derivery:  302
Newspaper derivery:  303
Newspaper derivery:  401
Newspaper derivery:  402
Newspaper derivery:  403
```

```
Newspaper derivery:  404
>>>
```

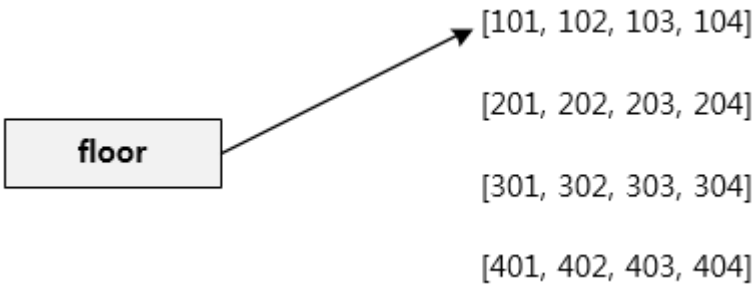

그림 4.4 2 차원 리스트 바인딩

for 문은 반복문이 실행될 때 for 문에서 들여쓰기된 문장을 실행하는데, 이중 루프에서는 for 문 안쪽에서 실행할 문장이 역시 또 다른 for 문입니다. 따라서 'for room in floor'라는 안쪽 for 문이 실행됩니다.

바깥쪽 for 문이 처음 실행될 때 floor 변수는 [101, 102, 103, 104]를 바인딩하고 있으므로 안쪽 for 문이 동작하면 room 변수는 101, 102, 103, 104 라는 값을 순서대로 바인딩하게 될 것입니다. 안쪽 for 문이 처음으로 실행될 때 room 은 101 이라는 값을 바인딩한 상태로 for 문 안쪽의 문장을 실행하게 됩니다. 따라서 화면에는 101 호에 대한 신문 배달이 출력될 것입니다. 그다음 역시 안쪽 for 문에서 102 호에 대한 신문 배달이 이뤄지고, 이러한 작업은 104 호까지 계속됩니다.

104 호까지 신문 배달이 끝나면 안쪽 for 문은 모두 수행이 완료됐기 때문에 바깥쪽 for 문 기준으로는 두 번째 반복이 시작됩니다. 따라서 floor 변수가 이번에는 [201, 202, 203, 204]를 바인딩한 상태에서 다시 안쪽 for 문을 수행하게 됩니다. 따라서 room 이라는 변수는 이번에는 201, 202, 203, 204 를 차례대로 바인딩하게 되며 이 값이 순서대로 출력됩니다.

1) 연습문제

문제 4-1

아래와 같은 패턴의 별(*)을 출력하는 프로그램을 작성해 보세요. 참고로 `print('*', end='')`와 같이 print 함수를 사용하면 줄바꿈 없이 화면에 출력할 수 있습니다.

```
*****
```

문제 4-2

아래와 같은 패턴의 별(*)을 출력하는 프로그램을 작성해보세요. (힌트: 이중 루프 사용)

```
*****
*****
*****
*****
```

문제 4-3

아래와 같은 패턴의 별(*)을 출력하는 프로그램을 작성해보세요.

```
*
**
***
****
*****
```

문제 4-4

아래와 같은 패턴의 별(*)을 출력하는 프로그램을 작성해 보세요.

```
*****
****
***
```

```
**
*
```

문제 4-5

아래와 같은 패턴의 별(*)을 출력하는 프로그램을 작성해 보세요.

```
    *
   **
  ***
 ****
*****
```

문제 4-6

아래와 같은 패턴의 별(*)을 출력하는 프로그램을 작성해 보세요.

```
*****
 ****
  ***
   **
    *
```

문제 4-7

아래와 같은 패턴의 별(*)을 출력하는 프로그램을 작성해 보세요.

```
    *
   ***
  *****
 *******
```

```
*********
```

문제 4-8

아래와 같은 패턴의 별(*)을 출력하는 프로그램을 작성 해보세요.

```
*********
 *******
  *****
   ***
    *
```

문제 4-9

이중 루프를 이용해 신문 배달을 하는 프로그램을 작성하세요. 단, 아래에서 arrears 리스트는 신문 구독료가 미납된 세대에 대한 정보를 포함하고 있는데, 해당 세대에는 신문을 배달하지 않아야 합니다.

```
>>> apart = [[101, 102, 103, 104],[201, 202, 203, 204],[301, 302, 303, 304], [401, 402, 403, 404]]
>>> arrears = [101, 203, 301, 404]
>>>
```