<[파이썬의 유용한 도구들] 제1장: 변치 않는 짝꿍, 튜플과 짝짓기 마법사 zip!>

선생님: (단단한 투명 상자에 구슬 몇 개를 넣어 보여주며) 얘들아, 안녕! 우리가 리스트라는 마법 주머니는 안에 든 물건을 마음대로 넣고 빼고 바꿀 수 있었지? 그런데 때로는 "이 안에 든 내용물은 절대 바뀌면 안 돼!" 하고 약속하고 싶을 때가 있어. 그럴 때 사용하는 것이 바로 **튜플(Tuple)**이란다!

<1. 절대 변하지 않는 약속, 튜플(Tuple)!>

선생님: 튜플은 리스트와 아주 비슷하게 순서가 있는 데이터 묶음이지만, 가장 큰 특징은 **한번 만들어지면 절대 그 내용을 바꿀 수 없다(Immutable, 불변)**는 거야! 마치 이 투명 상자처럼, 내용물을 볼 수는 있지만 열어서 바꿀 수는 없는 것과 같지. 만들 때는 소괄호 ()를 사용한단다.

```
# 소괄호를 사용한 튜플 생성
my_tuple = (1, 2, 4)
print("나의 첫 튜플:", my_tuple)
print("튜플의 타입은?", type(my_tuple))

# my_tuple[0] = 100 # 이 코드를 실행하면? TypeError! "튜플은 내용물을 바꿀 수 없어!"
```

코딩새싹 🌱: 와! 정말 에러가 나네요! 그럼 튜플은 왜 쓰는 거예요? 리스트가 더 편한 거 아니에요?

선생님: 좋은 질문이야! 내용이 바뀌지 않는다는 건 때로는 아주 큰 장점이 된단다. 예를 들어, 프로그램에서 절대로 변하면 안 되는 중요한 설정값이나, 함수의 결과로 여러 값을 한꺼번에 안전하게 전달하고 싶을 때 유용하지.

궁금이 💡: 그럼 튜플도 리스트처럼 인덱싱이나 슬라이싱은 할 수 있나요?

선생님: 물론! 순서가 있는 친구니까 그건 가능해! 그리고 리스트로 변신시키는 마법도 있지.

```
print("튜플의 0번 인덱스:", my_tuple[0])     # 인덱싱 가능!
print("튜플의 1번부터 끝까지:", my_tuple[1:]) # 슬라이싱 가능!

# 만약 정말 튜플 내용을 바꿔야 한다면? 리스트로 변신시켰다가 다시 튜플로!
my_list_from_tuple = list(my_tuple) # 튜플을 리스트로 변신!
my_list_from_tuple[0] = 100
print("리스트로 바꿔서 수정한 결과:", my_list_from_tuple)
# my_tuple_again = tuple(my_list_from_tuple) # 다시 튜플로 만들 수도 있어!

# 튜플도 곱하기 연산은 가능해! (반복)
a_tuple = (1, 3, 5, 10)
print("튜플 a * 2 =", a_tuple * 2)
```

선생님: 그리고 함수가 여러 개의 결과물을 돌려줄 때, 파이썬은 사실 이 튜플로 묶어서 살짝 건네준단다!

```
def get_two_numbers():
  return 1, 3 # 사실 이건 (1, 3) 이라는 튜플을 돌려주는 거야!
```

```
x, y = get_two_numbers() # 튜플 (1, 3)을 x와 y에 자동으로 나눠 담기! (튜플 언패킹)
print(f"함수에서 받은 x는 {x}, y는 {y}")
```

<2. 짝꿍 만들기 마법사, zip() 함수!>

선생님: (지퍼가 달린 옷을 보여주며) 얘들아, 이 지퍼는 양쪽의 이빨들을 하나씩 착착 맞춰서 잠가주지? zip() 함수는 바로 이 지퍼처럼, 여러 개의 리스트나 튜플 같은 데이터 묶음들을 받아서, 같은 위치에 있는 것들끼리 짝을 지어 튜플로 묶어주는 마법사란다!

```
a_zip_list = [1, 2, 3]
b_zip_list = ['a', 'b', 'c']

# a_zip_list와 b_zip_list의 짝꿍들을 만들어줘!
zipped_data_iterator = zip(a_zip_list, b_zip_list)
print("zip 결과 (그냥 출력하면?):", zipped_data_iterator) # 어? 이상한 게 나왔네?
print("zip 결과를 튜플 리스트로 변신!", list(zipped_data_iterator)) # [(1,'a'), (2,'b'), (3,'c')]
```

코딩새싹 🌱: zip도 filter나 map처럼 바로 결과가 안 보이고 특별한 꾸러미를 주네요! list()로 감싸야 짝꿍들이 보여요!

선생님: 맞아! zip은 "짝꿍 만들 준비 완료!" 꾸러미를 주고, 우리가 그걸 리스트나 튜플로 만들어달라고 하거나, for문으로 하나씩 꺼내 써야 한단다.

```
names = ["왕새우", "박태준", "오정민"]
scores = [15, 95, 100]

# 이름과 점수를 짝지어서 출력해볼까? (이때도 튜플 언패킹이 쓰이지!)
for name, score in zip(names, scores):
  print(f"{name}의 수학성적은 {score}점")
```

궁금이 💡: 만약 zip에 넣는 리스트들의 길이가 서로 다르면 어떻게 돼요?

선생님: 아주 중요한 질문이야! zip 마법사는 가장 짧은 리스트의 길이에 맞춰서 짝을 짓고, 긴 리스트의 남은 부분은 그냥 무시해버린단다. 마치 짧은 쪽 지퍼 이빨이 다 끝나면 더 이상 못 올라가는 것처럼!

```
fruits = ['apple', 'banana', 'cherry', 'strawberry'] # 4개
prices = [1200, 500, 2500] # 3개
counts = [10, 20, 30]      # 3개

# 3개의 리스트를 짝지으면, 가장 짧은 prices와 counts의 길이(3)에 맞춰서 짝이 지어져.
fruit_details = zip(fruits, prices, counts)
print("\n길이가 다른 리스트 zip 결과:", list(fruit_details))
# [('apple', 1200, 10), ('banana', 500, 20), ('cherry', 2500, 30)]
# 'strawberry'는 짝이 없어서 무시당했네!
```

선생님: zip 없이 for문과 enumerate로 비슷한 효과를 내려면 꽤 복잡하겠지? (사용자가 제공한 for문 예시를 보여주며) zip이 얼마나 편리한 마법인지 알 수 있겠지!

<3. 마법 주문의 유연성: 함수의 기본값 재료!>

선생님: 우리가 함수라는 마법 주문을 만들 때, "이 재료는 꼭 필요하지만, 저 재료는 안 넣으면 그냥 이걸로 써!" 하고 미리 **기본값**을 정해줄 수 있단다.

```
# b와 c 재료에는 기본값을 정해두자!
def plus_with_defaults(a, b=10, c=5):
  print(f"a={a}, b={b}, c={c}")
  return a - b * c

print("\n--- 기본값 재료 마법 ---")
# a 재료만 넣으면, b와 c는 기본값(10, 5)을 사용!
print("plus_with_defaults(2) 결과:", plus_with_defaults(2)) # a=2, b=10, c=5 -> 2 - 10*5 = -48

# b 재료 값을 바꿔서 넣어주면?
print("plus_with_defaults(2, b=3) 결과:", plus_with_defaults(2, b=3)) # a=2, b=3, c=5 -> 2 - 3*5 = -13
```

코딩새싹 🌱: 와! b=10 이렇게 써두니까, 함수를 부를 때 b를 안 넣어줘도 알아서 10으로 계산해요! 정말 편리한데요!

궁금이 💡: 그럼 *args랑은 뭐가 다른 거예요? *args도 여러 개를 받는 거였잖아요!

선생님: 좋은 질문이야!

- ***args**: (마법 주머니 파라미터) 정해지지 않은 '개수'의 재료들을 하나의 튜플로 묶어서 받을 때 사용해. 재료의 이름은 중요하지 않고 순서대로 들어오지.
- **기본값 재료**: 함수에 필요한 재료의 '종류와 순서'는 정해져 있지만, 특정 재료를 생략했을 때 대신 사용할 값을 미리 정해두는 거야. 재료 이름을 지정해서 값을 넣어줄 수도 있고!

선생님: (작은 보석함과 지퍼가 달린 파우치를 보여주며) 자, 오늘 우리는 절대 변하지 않는 보석함 같은 '튜플', 여러 가닥을 하나로 착! 묶어주는 마법 지퍼 'zip 함수', 그리고 필요에 따라 재료를 생략할 수 있는 '함수의 기본값'까지 배웠어! 이런 유용한 도구들을 잘 활용하면 파이썬 코드가 더욱 깔끔하고 강력해질 거란다! 오늘 탐험도 정말 멋졌어! ✨