jeśli naprawdę zależy nam na utworzeniu **statycznej** tablicy o wymiarach N*M, to musimy jawnie utworzyć nowe podlisty symulujące wiersze tablicy, np. używając sugerowanej wcześniej konstrukcji:

```
t3 = [0] * N
t3 = [ [0] * N for i in range(M) ]
```

lub zwięźle, w jednej linijce:

```
t3 = [[0]*N for i in range(M)]
```

Tuple (czasem zwane krotkami)

Tak zwane tuple w Pythonie budzą często zakłopotanie u osób podejmujących się nauki tego języka. Można je opisać jako pewną konwencję grupowania danych, zbliżoną do tej znanej z dziedziny baz danych, gdzie występuje pojęcie tzw. *rekordu* (w Polsce używany bywa także termin *krotka*). Ale pythonowa tupla, w przeciwieństwie do bazodanowego rekordu, nie może być modyfikowana, więc ta analogia jest nieco chybiona.

Definicję tupli łatwo rozpoznasz po nawiasach zwykłych () okalających listę wartości, np. `res=(False, True)`. Z powodu podobieństwa wizualnego wiele osób myli tuple z listami!

Aby ułatwić zrozumienie tego pojęcia, podsumowałem w tabeli 5.2 cechy charakterystyczne tupli w Pythonie.

TABELA 5.2. Tuple w Pythonie — cechy charakterystyczne

CECHA	KOMENTARZ
Uporządkowanie	Elementy w tupli znajdują się na określonych pozycjach i można do nich sięgnąć przez tzw. indeks, czyli numer porządkowy (pamiętaj, że numeracja zaczyna się od 0, a nie od 1). Podobnie jak dla list, możliwe jest używanie indeksów ujemnych (np. -1 to ostatni element listy) oraz wycinków (zakresy podawane z dwukropkiem, np. 2:4).
Duplikaty	Dozwolone jest tworzenie tupli zawierającej te same wartości.
Modyfikowalność	Tupli nie można modyfikować.
Stała długość	Po wstępnej inicjacji nie można już dokładać nowych elementów (patrz poprzednia cecha) — nie można zatem używać metod takich jak np. `append()` lub `insert()`.
Dowolna zawartość	Tuple mogą zawierać listę wartości dowolnych typów prostych (znaków, liczb, napisów) oraz nawet obiekty złożone (w zasadzie referencje do nich).
Symbol rozpoznawczy	Nawiasy proste: ().

Okazuje się też, że lista operacji dozwolonych dla tupli jest dość uboga. Ta klasa oferuje raptem dwie metody:

- `count(x)` — zwraca liczbę *wystąpień* elementu x;
- `index(x)` — zwraca *indeks pozycji* szukanego elementu x

oraz oferuje znane z pythonowych list operatory: [] (dostęp) i + (sklejenie, konkatenacja — z dwóch tupli możesz utworzyć nową).

Przykład:

```
dni = ( 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31)
print (dni.count(31)) # Wypisze: 7
print (dni.index(30)) # Wypisze: 3
```

Przypatrz się kilku sposobom deklarowania i użycia tupli (fragment skryptu *tuple.py*):

```
tupla1 = (1, 1, 2, 3, 5, 8, 13)          # Najpopularniejsza notacja
print(len(tupla1))                       # Wypisze: 7, tj. długość tupli
tupla1[3]=100                            # Błąd, tupli nie można modyfikować! (*)
# Alternatywna notacja, z użyciem konstruktora klasy 'tuple':
tupla2=tuple ( (1, 1, 2, 3, 5, 8, 13) )
tupla3=("Dr Who", "Dawid Tennant",   10) # Patrz uwaga (**)
tupla4=("Dr Who", "Jodie Whittaker", 13)
tupla3_plus_tupla4=tupla3+tupla4
print(tupla4)                # Wypisze: (' Dr Who', 'Jodie Whittaker', 13)
print(tupla3_plus_tupla4)    # Wypisze: ('Dr Who', 'Dawid Tennant', 10, 'Dr Who', 'Jodie
                             # Whittaker', 13)
print(tupla4[1])             # Wypisze: Jodie Whittaker
```

- Linia (*) jest błędna. Jeśli uruchomisz skrypt, to Python zgłosi błąd! (W związku z tym usuń ją z kodu lub oznacz ten wiersz znakiem komentarza #).
- Ad (**): *https://www.examinerlive.co.uk/news/here-full-list-dr-who-13339904.*

Zapis `tupla0=(1, )` spowoduje utworzenie obiektu klasy `tuple` zawierającego *pojedynczą* wartość (przecinek na końcu to nie jest błąd). Wbrew pozorom poprawny składniowo zapis `tupla0=(1)` spowodowałby utworzenie w Pythonie zwykłej zmiennej liczbowej klasy `int` (nawiasy zostałyby zignorowane jako zbędne).

Modyfikacja tupli

Wcześniej napisałem i wręcz podkreślałem, że tupli nie można modyfikować. To prawda. Przykładowo, jeśli gdzieś w kodzie zdefiniujesz np. `t=("pi", "razy", "drzwi=")`, to nie da się już w żaden sposób dopisać czwartego elementu, np. zawierającego wynik obliczenia intrygującego wyrażenia „pi razy drzwi", ani zamienić „pi" na „epsilon" (przykładowo).

Jest jednak **drobny wyjątek od tej reguły** — zasada niezmienności tupli nie dotyczy "włożonych" do niej obiektów złożonych. Okazuje się, że można je modyfikować

Tak naprawdę nie łamiemy tu jednak żadnej reguły, gdyż tupla zawiera nie dane właściwego obiektu, ale wyłącznie *referencję* do niego. Zatem jeśli nie spróbujesz podmienić samej referencji (co i tak by się nie powiodło), ciągle możesz modyfikować dane wskazywanego przez nią obiektu!

Zapewne moje wyjaśnienia brzmią enigmatycznie i lepiej je zrozumiesz, patrząc na poniższy przykład, w którym dwuelementowa `tupla5` zawiera *na drugiej pozycji* pewną listę:

```
print("Ilustracja niebezpośredniego modyfikowania zawartości tupli")
nazwy_skrocone=['pon', 'wt', 'śr', 'czw', 'pt', 'sob'] # Ups, na tej liście brakuje niedzieli!
tupla5=("dni", nazwy_skrocone)
print("Tupla oryginalna, dwuelementowa:", tupla5)
# Modyfikujemy zawartość listy nazwy_skrocone (sama tupla5 jest nienaruszona)
tupla5[1].append("nd") # Dodajemy element na koniec listy nazwy_skrocone
print("Tupla po dodaniu 'nd' do listy wskazywanej przez drugi element tupli:\n", tupla5)
```

Jeśli uruchomisz skrypt *tuple.py*, to ten fragment kodu wyświetli następujący wynik:

```
Ilustracja niebezpośredniego modyfikowania zawartości tupli
Tupla oryginalna, dwuelementowa: ('dni', ['pon', 'wt', 'śr', 'czw', 'pt', 'sob'])
Tupla po dodaniu 'nd' do listy wskazywanej przez drugi element tupli:
 ('dni', ['pon', 'wt', 'śr', 'czw', 'pt', 'sob', 'nd'])
```

Kontynuując ciekawostki, dodajmy, że tuplę można też zmodyfikować przez skonwertowanie jej zawartości na listę, którą można poddać dowolnym manipulacjom, a następnie przypisać z powrotem do pierwotnej zmiennej.

Przykład:

```
print("Modyfikacja tupli przez konwersję na listę")
dni_robocze=('pon', 'wt', 'śr', 'czw', 'pt')
print("Dni robocze", dni_robocze)          # Wypisze: Dni robocze ('pon', 'wt', 'śr', 'czw', 'pt')
dni_robocze_lista=list(dni_robocze)        # Tworzymy listę na podstawie tupli
dni_robocze_lista.append('sob')            # Dodajemy 'sob' do listy
dni_robocze=tuple(dni_robocze_lista)       # Tworzymy tuplę na podstawie listy
print("Dni robocze", dni_robocze)          # Wypisze: Dni robocze ('pon', 'wt', 'śr', 'czw', 'pt', 'sob')
```

Oczywiście nie jest to prawdziwe modyfikowanie tupli, ale swego rodzaju obejście problemu poprzez ponowne zbudowanie pewnego obiektu pod jego pierwotną nazwą (zupełnie jak odbudowa po wojnie Zamku Królewskiego w Warszawie, podczona z dołożeniem do niego elementów, które przecież nie istniały przed jego rzeniem przez niemieckiego agresora).

Zastosowania programistyczne

Można zadać sobie pytanie: po co w ogóle w Pythonie funkcjonuje taki dziwoląg jak tupla, będąca niejako ubogim bratem listy? Okazuje się, że wbrew pozorom programiści dość często używają tych struktur:

- Tuple służą do definiowania **niezmiennych zestawów danych**, zapewniając większą stabilność i czytelność kodu
- Dostęp do elementów zawartych w tupli jest znacznie szybszy od np. list, gdyż jest to statyczna struktura danych. Używając tupli, zwiększamy zatem **wydajność aplikacji**.
- Tuple umożliwiają tzw. **przypisania wielokrotne**.

Poniżej pokazuję przykład użycia tupli jako **statycznej listy referencyjnej**, zawierającej pewne dozwolone wartości służące do weryfikowania danych wprowadzanych przez użytkownika (kontynuacja skryptu *tuple.py*):

```
dozwolone_waluty=('PLN', 'EUR', 'USD') # Tupla
print ("Dozwolone waluty to:", dozwolone_waluty)
s=input("Podaj walutę: ")                    # Program poprosi użytkownika o wpisanie danych
if s in dozwolone_waluty:
    print("Poprawna waluta")
else:
    print("Nieznany kod waluty")
```

Przydadzą się także **przypisania wielokrotne** pozwalające pobrać wartości (np. pewne stałe) z tupli i użyć ich do inicjacji zwykłych *zmiennych*, np.:

```
pacjent_domyslny=("John Doe", 55) # To jest tupla
# Zmienne 'nazwisko' i 'wiek' są zainicjowane wartościami pobranymi z tupli
(nazwisko, wiek)= pacjent_domyslny
print (f"Dane przyjętej osoby: Nazwisko: {nazwisko}, wiek={wiek}")
```

(Python wypisze: „Dane przyjętej osoby: Nazwisko: John Doe, wiek=55").

Tuple pozwalają na czytelne zwracanie wielu wartości naraz z funkcji Pythona. Oto prosty przykład:

```
def sumaNaturalna(x, y):
    if x > 0 and y > 0:
        return True, x+y
    else:
        return False, -1 # Umowna sygnalizacja błędów

wynik = sumaNaturalna(2,-3)

if wynik[0] == True:          # Dekodujemy wyniki zwracane przez sumaNaturalna()
    print("Wszystko OK, suma=", wynik[1])
else:
    print("Błąd, potrafię sumować tylko liczby naturalne") # (*)
```

Powyższy kod oczywiście wykona instrukcję z linijki oznaczonej (*).

Na tym kończymy rozgrzewkę i zapraszam do kolejnego rozdziału opisującego techniki modelowania nowych struktur danych przy wykorzystaniu koncepcji programowania obiektowego.